6 Heapsort

6.1 Heaps

6.1-1

> What are the minimum and maximum numbers of elements in a heap of height h?

At least 2^h and at most $2^{h+1} - 1$. Can be seen because a complete binary tree of depth $h-1$ has $\sum_{i=0}^{h-1} 2^i = 2^h - 1$ elements, and the number of elements in a heap of depth h is between the number for a complete binary tree of depth $h-1$ exclusive and the number in a complete binary tree of depth h inclusive.

6.1-2

> Show that an n-element heap has height $\lfloor \lg n \rfloor$.

Write $n = 2^m - 1 + k$ where m is as large as possible. Then the heap consists of a complete binary tree of height $m-1$, along with k additional leaves along the bottom. The height of the root is the length of the longest simple path to one of these k leaves, which must have length m. It is clear from the way we defined m that $m = \lfloor \lg n \rfloor$.

6.1-3

> Show that in any subtree of a max-heap, the root of the subtree contains the largest value occuring anywhere in the subtree.

If the largest element in the subtree were somewhere other than the root, it has a parent that is in the subtree. So, it is larger than it's parent, so, the heap property is violated at the parent of the maximum element in the subtree.

6.1-4

> Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

In any of the leaves, that is, elements with index $\lfloor n/2 \rfloor + k$, where $k \geq 1$ (see exercise 6.1-7), that is, in the second half of the heap array.

6.1-5

> Is an array that is in sorted order a min-heap?

Yes. For any index i, both $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ are larger and thus the elements indexed by them are greater or equal to $A[i]$ (because the array is sorted.)

6.1-6

> Is the array with values $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ a max-heap?

No. Since $\text{PARENT}(7)$ is 6 in the array. This violates the max-heap property.

6.1-7

> Show that, with the array representation for sorting an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

Let's take the left child of the node indexed by $\lfloor n/2 \rfloor + 1$.

$$\begin{aligned} \text{LEFT}(\lfloor n/2 \rfloor + 1) &= 2(\lfloor n/2 \rfloor + 1) \\ &> 2(n/2 - 1) + 2 \\ &= n - 2 + 2 \\ &= n. \end{aligned}$$

Since the index of the left child is larger than the number of elements in the heap, the node doesn't have childrens and thus is a leaf. Same goes for all nodes with larger indices.

Note that if we take element indexed by $\lfloor n/2 \rfloor$, it will not be a leaf. In case of even number of nodes, it will have a left child with index n and in the case of odd number of nodes, it will have a left child with index $n-1$ and a right child with index n.

This makes the number of leaves in a heap of size n equal to $\lceil n/2 \rceil$.

6.2 Maintaining the heap property

6.2-1

> Using figure 6.2 as a model, illustrate the operation of $\text{MAX-HEAPIFY}(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

$$\begin{aligned} &\langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle \\ &\langle 27, 17, 10, 16, 13, 3, 1, 5, 7, 12, 4, 8, 9, 0 \rangle \\ &\langle 27, 17, 10, 16, 13, 9, 1, 5, 7, 12, 4, 8, 3, 0 \rangle \end{aligned}$$

6.2-2

> Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure $\text{MIN-HEAPIFY}(A, i)$, which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

```
MIN-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] < A[i]
        smallest = l
    else smallest = i
    if r ≤ A.heap-size and A[r] < A[smallest]
        smallest = r
    if smallest != i
        exchange A[i] with A[smallest]
        MIN-HEAPIFY(A, smallest)
```

The running time is the same. Actually, the algorithm is the same with the exceptions of two comparisons and some names.

6.2-3

> What is the effect of calling $\text{MAX-HEAPIFY}(A, i)$ when the element $A[i]$ is larger than its children?

No effect. The comparisons are carried out, $A[i]$ is found to be largest and the procedure just returns.

6.2-4

> What is the effect of calling $\text{MAX-HEAPIFY}(A, i)$ for $i > A.\text{heap-size}/2$?

No effect. In that case, it is a leaf. Both LEFT and RIGHT return values that fail the comparison with the heap size and i is stored in largest. Afterwards the procedure just returns.

6.2-5

> The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

```
MAX-HEAPIFY(A, i)
    while true
        l = LEFT(i)
        r = RIGHT(i)
        if l ≤ A.heap-size and A[l] > A[i]
            largest = l
        else largest = i
        if r ≤ A.heap-size and A[r] > A[largest]
            largest = r
        if largest == i
            return
        exchange A[i] with A[largest]
        i = largest
```

6.2-6

> Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$. (Hint: For a heap with n nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

Consider the heap resulting from A where $A[1] = 1$ and $A[i] = 2$ for $2 \leq i \leq n$. Since 1 is the smallest element of the heap, it must be swapped through each level of the heap until it is a leaf node. Since the heap has height $\lfloor \lg n \rfloor$, MAX-HEAPIFY has worst-case time $\Omega(\lg n)$.

6.3 Building a heap

6.3-1

> Using figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

$$\begin{aligned} &\langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle \\ &\langle 5, 3, 17, 22, 84, 19, 6, 10, 9 \rangle \\ &\langle 5, 3, 19, 22, 84, 17, 6, 10, 9 \rangle \\ &\langle 5, 84, 19, 22, 3, 17, 6, 10, 9 \rangle \\ &\langle 84, 5, 19, 22, 3, 17, 6, 10, 9 \rangle \\ &\langle 84, 22, 19, 5, 3, 17, 6, 10, 9 \rangle \\ &\langle 84, 22, 19, 10, 3, 17, 6, 5, 9 \rangle \end{aligned}$$

6.3-2

> Why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from $\lfloor A.\text{length}/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor A.\text{length}/2 \rfloor$?

Otherwise we won't be allowed to call MAX-HEAPIFY, since it will fail the condition of having the subtrees be max-heaps. That is, if we start with 1, there is no guarantee that $A[2]$ and $A[3]$ are roots of max-heaps.

6.3-3

> Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

From 6.1-7, we know that the leaves of a heap are the nodes indexed by

$$\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n.$$

Note that those elements corresponds to the second half of the heap array (plus the middle element if n is odd). Thus, the number of leaves in any heap of size n is $\lceil n/2 \rceil$. Let's prove by induction. Let n_h denote the number of nodes at height h. The upper bound holds for the base since $n_0 = \lceil n/2 \rceil$ is exactly the number of leaves in a heap of size n.

Now assume it holds for $h-1$. We have prove that it also holds for h. Note that if n_{h-1} is even each node at height h has exactly two children, which implies $n_h = n_{h-1}/2 = \lfloor n_{h-1}/2 \rfloor$. If n_{h-1} is odd, one node at height h has one child and the remaining has two children, which also implies $n_h = \lfloor n_{h-1}/2 \rfloor + 1 = \lceil n_{h-1}/2 \rceil$. Thus, we have

$$\begin{aligned} n_h &= \left\lceil \frac{n_{h-1}}{2} \right\rceil \\ &\leq \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil \right\rceil \\ &= \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^h} \right\rceil \right\rceil \\ &= \left\lceil \frac{n}{2^{h+1}} \right\rceil, \end{aligned}$$

which implies that it holds for h.

6.4 The heapsort algorithm

6.4-1

> Using figure 6.4 as a model, illustrate the operation of HEAPSORT on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

$$\begin{aligned} &\langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle \\ &\langle 5, 13, 20, 25, 7, 17, 2, 8, 4 \rangle \\ &\langle 5, 25, 20, 13, 7, 17, 2, 8, 4 \rangle \\ &\langle 25, 5, 20, 13, 7, 17, 2, 8, 4 \rangle \\ &\langle 25, 13, 20, 5, 7, 17, 2, 8, 4 \rangle \\ &\langle 25, 13, 20, 8, 7, 17, 2, 5, 4 \rangle \\ &\langle 4, 13, 20, 8, 7, 17, 2, 5, 25 \rangle \\ &\langle 20, 13, 4, 8, 7, 17, 2, 5, 25 \rangle \\ &\langle 20, 13, 17, 8, 7, 4, 2, 5, 25 \rangle \\ &\langle 5, 13, 17, 8, 7, 4, 2, 20, 25 \rangle \\ &\langle 17, 13, 5, 8, 7, 4, 2, 20, 25 \rangle \\ &\langle 2, 13, 5, 8, 7, 4, 17, 20, 25 \rangle \\ &\langle 13, 2, 5, 8, 7, 4, 17, 20, 25 \rangle \\ &\langle 13, 8, 5, 2, 7, 4, 17, 20, 25 \rangle \\ &\langle 4, 8, 5, 2, 7, 13, 17, 20, 25 \rangle \\ &\langle 8, 4, 5, 2, 7, 13, 17, 20, 25 \rangle \\ &\langle 8, 7, 5, 2, 4, 13, 17, 20, 25 \rangle \\ &\langle 4, 7, 5, 2, 8, 13, 17, 20, 25 \rangle \\ &\langle 7, 4, 5, 2, 8, 13, 17, 20, 25 \rangle \\ &\langle 2, 4, 5, 7, 8, 13, 17, 20, 25 \rangle \\ &\langle 5, 4, 2, 7, 8, 13, 17, 20, 25 \rangle \\ &\langle 2, 4, 5, 7, 8, 13, 17, 20, 25 \rangle \\ &\langle 4, 2, 5, 7, 8, 13, 17, 20, 25 \rangle \\ &\langle 2, 4, 5, 7, 8, 13, 17, 20, 25 \rangle \end{aligned}$$

6.4-2

> Argue the correctness of HEAPSORT using the following loop invariant:
>
> At the start of each iteration of the **for** loop of lines 2-5, the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$, and the subarray $A[i+1..n]$ contains the $n-i$ largest elements of $A[1..n]$, sorted.

Initialization: The subarray $A[i+1..n]$ is empty, thus the invariant holds.

Maintenance: $A[1]$ is the largest element in $A[1..i]$ and it is smaller than the elements in $A[i+1..n]$. When we put it in the ith position, then $A[i..n]$ contains the largest elements, sorted. Decreasing the heap size and calling MAX-HEAPIFY turns $A[1..i-1]$ into a max-heap. Decrementing i sets up the invariant for the next iteration.

Termination: After the loop $i = 1$. This means that $A[2..n]$ is sorted and $A[1]$ is the smallest element in the array, which makes the array sorted.

6.4-3

> What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? What about decreasing order?

Both of them are $\Theta(n \lg n)$.

If the array is sorted in increasing order, the algorithm will need to convert it to a heap that will take $O(n)$. Afterwards, however, there are $n - 1$ calls to MAX-HEAPIFY and each one will perform the full $\lg k$ operations. Since:

$$\sum_{k=1}^{n-1} \lg k = \lg((n-1)!) = \Theta(n \lg n).$$

Same goes for decreasing order. BUILD-MAX-HEAP will be faster (by a constant factor), but the computation time will be dominated by the loop in HEAPSORT, which is $\Theta(n \lg n)$.

6.4-4

> Show that the worst-case running time of HEAPSORT is $\Omega(n \lg n)$.

This is essentially the first part of exercise 6.4-3. Whenever we have an array that is already sorted, we take linear time to convert it to a max-heap and then $n \lg n$ time to sort it.

6.4-5 *

> Show that when all elements are distinct, the best-case running time of HEAPSORT is $\Omega(n \lg n)$.

This proved to be quite tricky. My initial solution was wrong. Also, heapsort appeared in 1964, but the lower bound was proved by Schaffer and Sedgewick in 1992. It's evil to put this an exercise.

Let's assume that the heap is a full binary tree with $n = 2^k - 1$. There are 2^{k-1} leaves and $2^{k-1} - 1$ inner nodes.

Let's look at sorting the first 2^{k-1} elements of the heap. Let's consider their arrangement in the heap and color the leaves to be red and the inner nodes to be blue. The colored nodes are a subtree of the heap (otherwise there would be a contradiction). Since there are 2^{k-1} colored nodes, at most 2^{k-2} are red, which means that at least $2^{k-2} - 1$ are blue.

While the red nodes can jump directly to the root, the blue nodes need to travel up before they get removed. Let's count the number of swaps to move the blue nodes to the root. The minimal case of swaps is when

1. there are $2^{k-2} - 1$ blue nodes and
2. they are arranged in a binary tree.

If there are d such blue nodes, then there would be $i = \lg d$ levels, each containing 2^i nodes with length i. Thus the number of swaps is,

$$\sum_{i=0}^{\lg d} i2^i = 2 + (\lg d - 2)2^{\lg d} = \Omega(d \lg d).$$

And now for a lazy (but cute) trick. We've figured out a tight bound on sorting half of the heap. We have the following recurrence:

$$T(n) = T(n/2) + \Omega(n \lg n).$$

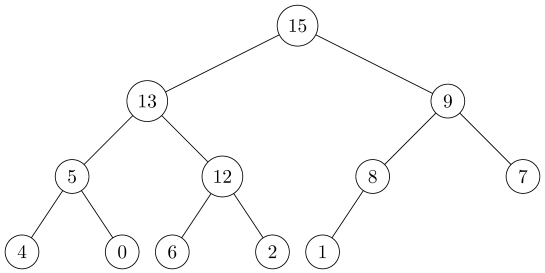
Applying the master method, we get that $T(n) = \Omega(n \lg n)$.

6.5 Priority queues

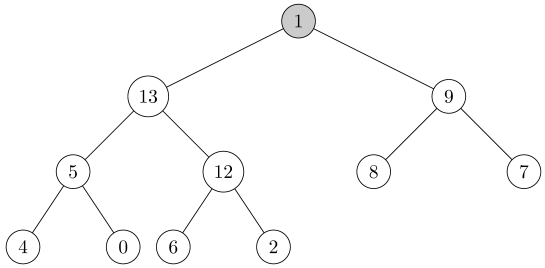
6.5-1

> Illustrate the operation HEAP-EXTRACT-MAX on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

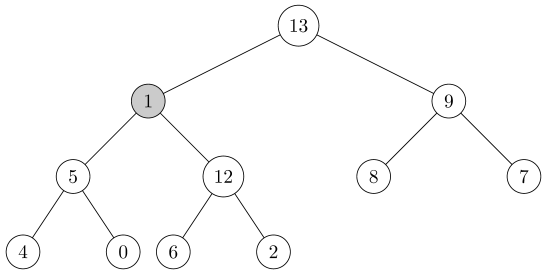
1. Original heap.



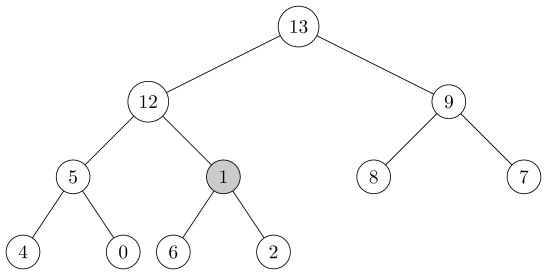
2. Extract the max node 15, then move 1 to the top of the heap.



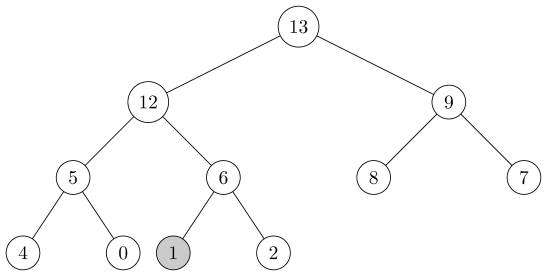
3. Since $13 > 9 > 1$, swap 1 and 13.



4. Since $12 > 5 > 1$, swap 1 and 12.



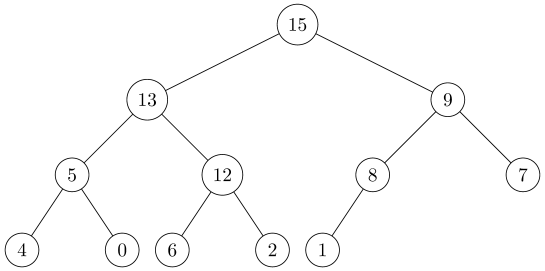
5. Since $6 > 2 > 1$, swap 1 and 6.



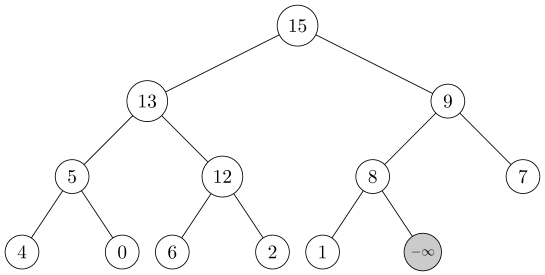
6.5-2

> Illustrate the operation of MAX-HEAP-INSERT(A, 10) on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

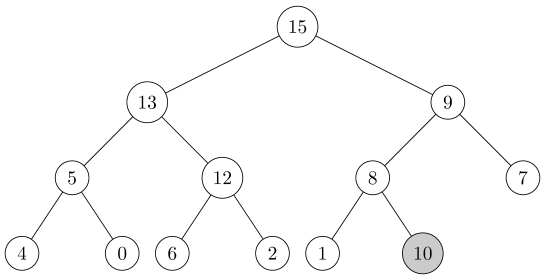
1. Original heap.



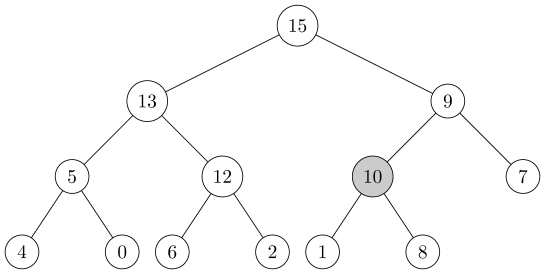
2. Since MAX-HEAP-INSERT(A, 10) is called, we append a node assigned value $-\infty$.



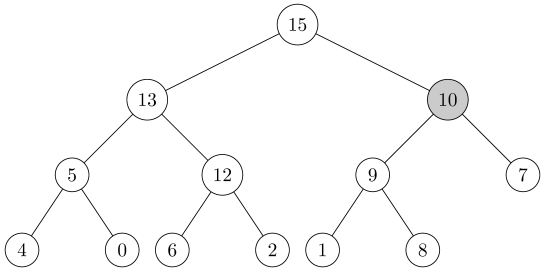
3. Update the key value of the new node.



4. Since the parent key is smaller than 10, the nodes are swapped.



5. Since the parent key is smaller than 10, the nodes are swapped.



6.5-3

> Write pseudocode for the procedures HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY, and MIN-HEAP-INSERT that implement a min-priority queue with a min-heap.

```
HEAP-MINIMUM(A)
    return A[1]
```

```
HEAP-EXTRACT-MIN(A)
    if A.heap-size < 1
        error "heap underflow"
    min = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    MIN-HEAPIFY(A, 1)
    return min
```

```
HEAP-DECREASE-KEY(A, i, key)
    if key > A[i]
        error "new key is larger than current key"
    A[i] = key
    while i > 1 and A[PARENT(i)] > A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

```
MIN-HEAP-INSERT(A, key)
    A.heap-size = A.heap-size + 1
    A[A.heap-size] = ∞
    HEAP-DECREASE-KEY(A, A.heap-size, key)
```

6.5-4

> Why do we bother setting the key of the inserted node to $-\infty$ in line 2 of MAX-HEAP-INSERT when the next thing we do is increase its key to the desired value?

In order to pass the guard clause. Otherwise we have to drop the check if $\text{key} < A[i]$.

6.5-5

> Argue the correctness of HEAP-INCREASE-KEY using the following loop invariant:
>
> At the start of each iteration of the **while** loop of lines 4-6, the subarray $A[1..A.\text{heap-size}]$ satisfies the max-heap property, except that there may be one violation: $A[i]$ may be larger than $A[\text{PARENT}(i)]$.
>
> You may assume that the subarray $A[1..A.\text{heap-size}]$ satisfies the max-heap property at the time HEAP-INCREASE-KEY is called.

Initialization: A is a heap except that $A[i]$ might be larger that it's parent, because it has been modified. $A[i]$ is larger than its children, because otherwise the guard clause would fail and the loop will not be entered (the new value is larger than the old value and the old value is larger than the children).

Maintenance: When we exchange $A[i]$ with its parent, the max-heap property is satisfied except that now $A[\text{PARENT}(i)]$ might be larger than its parent. Changing i to its parent maintains the invariant.

Termination: The loop terminates whenever the heap is exhausted or the max-heap property for $A[i]$ and its parent is preserved. At the loop termination, A is a max-heap.

6.5-6

> Each exchange operation on line 5 of HEAP-INCREASE-KEY typically requires three assignments. Show how to use the idea of the inner loop of INSERTION-SORT to reduce the three assignments down to just one assignment.

```
HEAP-INCREASE-KEY(A, i, key)
    if key < A[i]
        error "new key is smaller than current key"
    while i > 1 and A[PARENT(i)] < key
        A[i] = A[PARENT(i)]
        i = PARENT(i)
    A[i] = key
```

6.5-7

> Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue. (Queues and stacks are defined in section 10.1).

Both are simple. For a stack we keep adding elements in increasing priority, while in a queue we add them in decreasing priority. For the stack we can set the new priority to $\text{HEAP-MAXIMUM}(A) + 1$. For the queue we need to keep track of it and decrease it on every insertion.

Both are not very efficient. Furthermore, if the priority can overflow or underflow, so will eventually need to reassign priorities.

6.5-8

> The operation $\text{HEAP-DELETE}(A, i)$ deletes the item in node i from heap A. Give an implementation of HEAP-DELETE that runs in $O(\lg n)$ time for an n-element max-heap.

```
HEAP-DELETE(A, i)
    if A[i] > A[A.heap-size]
        A[i] = A[A.heap-size]
        MAX-HEAPIFY(A, i)
    else
        HEAP-INCREASE-KEY(A, i, A[A.heap-size])
    A.heap-size = A.heap-size - 1
```

Note: The following algorithm is wrong. For example, given an array $A = [15, 7, 9, 1, 2, 3, 8]$ which is a max-heap, and if we delete $A[5] = 2$, then it will fail.

```
HEAP-DELETE(A, i)
    A[i] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, i)
```

- before:

```
      15
     /   \
    7     9
   / \   / \
  1   2 3   8
```

- after (which is wrong since $8 > 7$ violates the max-heap property):

```
      15
     /   \
    7     9
   / \   /
  1   8 3
```

6.5-9

> Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (Hint: Use a min-heap for k-way merging.)

We take one element of each list and put it in a min-heap. Along with each element we have to track which list we took it from. When merging, we take the minimum element from the heap and insert another element off the list it came from (unless the list is empty). We continue until we empty the heap.

We have n steps and at each step we're doing an insertion into the heap, which is $\lg k$.

Suppose that sorted lists on input are all nonempty, we have the following pseudocode.

```
def MERGE-SORTED-LISTS(lists)
    n = lists.length
    // Take the lowest element from each of lists together with an index of
the list and make list of such pairs.
    // Pairs are of "type" (element-value, index-of-list)
    let lowest-from-each be an empty array
    for i = 1 to n
        add (lists[i][0], i) to lowest-from-each
        delete lists[i][0]
    // This makes min-heap from list lowest-from-each.
    // We are sssuming that pairs of "type" (element-value, index-of-list)
are compared according to the values of elements.
    A = MIN-HEAP(lowest-from-each)
    let merged-lists be an empty array
    while not !min-heap.EMPTY()
        element-value, index-of-list = HEAP-EXTRACT-MIN(A)
        add element-value to merged-lists
        if lists[index-of-list].length > 0
            MIN-HEAP-INSERT(A, (lists[index-of-list][0], index-of-list))
            delete lists[index-of-list][0]
    return merged-lists
```

Problem 6-1 Building a heap using insertion

> We can build a heap by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the following variation of the BUILD-MAX-HEAP procedure:
>
> ```
> BUILD-MAX-HEAP'(A)
>     A.heap-size = 1
>     for i = 2 to A.length
>         MAX-HEAP-INSERT(A, A[i])
> ```
>
> **a.** Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP′ always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.
>
> **b.** Show that in the worst case, BUILD-MAX-HEAP′ requires $\Theta(n \lg n)$ time to build a n-element heap.

a. Consider the following counterexample.

- Input array $A = \langle 1, 2, 3 \rangle$:
- BUILD-MAX-HEAP(A): $A = \langle 3, 2, 1 \rangle$.
- BUILD-MAX-HEAP′(A): $A = \langle 3, 1, 2 \rangle$.

b. Each insert step takes at most $O(\lg n)$, since we are doing it n times, we get a bound on the runtime of $O(n \lg n)$.

Problem 6.2 Maintaining the heap property

> A d-***ary heap*** is like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children.
>
> **a.** How would you represent a d-ary heap in an array?
>
> **b.** What is the height of a d-ary heap of n elements in terms of n and d?
>
> **c.** Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.
>
> **d.** Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n.
>
> **e.** Give an efficient implementation of $\text{INCREASE-KEY}(A, i, k)$, which flags an error if $k < A[i]$, but otherwise sets $A[i] = k$ and then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d and n.

a. We can use those two following functions to retrieve parent of i-th element and j-th child of i-th element.

```
d-ARY-PARENT(i)
    return floor((i - 2) / d + 1)
```

```
d-ARY-CHILD(i, j)
    return d(i - 1) + j + 1
```

Obviously $1 \le j \le d$. You can verify those functions checking that

$$\text{d-ARY-PARENT}(\text{d-ARY-CHILD}(i, j)) = i.$$

Also easy to see is that binary heap is special type of d-ary heap where $d = 2$, if you substitute d with 2, then you will see that they match functions PARENT, LEFT and RIGHT mentioned in book.

b. Since each node has d children, the height of a d-ary heap with n nodes is $\Theta(\log_d n)$.

c. d-ARY-HEAP-EXTRACT-MAX(A) consists of constant time operations, followed by a call to d-ARY-MAX-HEAPIFY(A, i).

The number of times this recursively calls itself is bounded by the height of the d-ary heap, so the running time is $O(d \log_d n)$.

```
d-ARY-HEAP-EXTRACT-MAX(A)
    if A.heap-size < 1
        error "heap under flow"
    max = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    d-ARY-MAX-HEAPIFY(A, 1)
    return max
```

```
d-ARY-MAX-HEAPIFY(A, i)
    largest = i
    for k = 1 to d
        if d-ARY-CHILD(k, i) ≤ A.heap-size and A[d-ARY-CHILD(k, i)] > A[i]
            if A[d-ARY-CHILD(k, i)] > A[largest]
                largest = d-ARY-CHILD(k, i)
    if largest != i
        exchange A[i] with A[largest]
        d-ARY-MAX-HEAPIFY(A, largest)
```

d. The runtime is $O(\log_d n)$ since the **while** loop runs at most as many times as the height of the d-ary array.

```
d-ARY-MAX-HEAP-INSERT(A, key)
    A.heap-size = A.heap-size + 1
    A[A.heap-size] = key
    i = A.heap-size
    while i > 1 and A[d-ARY-PARENT(i)] < A[i]
        exchange A[i] with A[d-ARY-PARENT(i)]
        i = d-ARY-PARENT(i)
```

e. The runtime is $O(\log_d n)$ since the **while** loop runs at most as many times as the height of the d-ary array.

```
d-ARY-INCREASE-KEY(A, i, key)
    if key < A[i]
        error "new key is smaller than current key"
    A[i] = key
    while i > 1 and A[d-ARY-PARENT(i)] < A[i]
        exchange A[i] with A[d-ARY-PARENT(i)]
        i = d-ARY-PARENT(i)
```

Problem 6.3 Building a heap

An $m \times n$ Young tableau is an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be ∞, which we treat as nonexistent elements. Thus, a Young tableau can be used to hold $r \le mn$ finite numbers.

a. Draw 4×4 tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

b. Argue that an $m \times n$ Young tableau Y is empty if $Y[1, 1] = \infty$. Argue that Y is full (contains mn elements) if $Y[m, n] < \infty$.

c. Give an algorithm to implement EXTRACT-MIN on a nonempty $m \times n$ Young tableau that runs in $O(m + n)$ time. Your algorithm should use a recursive subroutine that solves an $m \times n$ problem by recursively solving either an $(m - 1) \times n$ or an $m \times (n - 1)$ subproblem. (Hint: Think about MAX-HEAPIFY.) Define $T(p)$ where $p = m + n$, to be the maximum running time of EXTRACT-MIN on any $m \times n$ Young tableau. Give and solve a recurrence relation for $T(p)$ that yields the $O(m + n)$ time bound.

d. Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m + n)$ time.

e. Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau to sort n^2 numbers in $O(n^3)$ time.

f. Give an $O(m + n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.

a.

$$\begin{matrix} 2 & 3 & 12 & 14 \\ 4 & 8 & 16 & \infty \\ 5 & 9 & \infty & \infty \\ \infty & \infty & \infty & \infty \end{matrix}$$

b. If the top left element is ∞, then all the elements on the first row need to be ∞. But if this is the case, all other elements need to be ∞ because they are larger than the first element on their column.

If the bottom right element is smaller than ∞, all the elements on the bottom row need to be smaller than ∞. But so are the other elements in the tableau, because each is smaller than the bottom element of its column.

c. The $A[1, 1]$ is the smallest element. We store it, so we can return it later and then replace it with ∞. This breaks the Young tableau property and we need to perform a procedure, similar to MAX-HEAPIFY, to restore it.

We compare $A[i, j]$ with each of its neighbours and exchange it with the smallest. This restores the property for $A[i, j]$ but reduces the problem to either $A[i, j + 1]$ or $A[i + 1, j]$. We terminate when $A[i, j]$ is smaller than its neighbours.

The relation in question is

$$T(p) = T(p - 1) + O(1) = T(p - 2) + O(1) + O(1) = \cdots = O(p).$$

d. The algorithm is very similar to the previous, except that we start with the bottom right element of the tableau and move it upwards and leftwards to the correct position. The asymptotic analysis is the same.

e. We can sort by starting with an empty tableau and inserting all the n^2 elements in it. Each insertion is $O(n + n) = O(n)$. The complexity is $n^2 O(n) = O(n^3)$. Afterwards we can take them one by one and put them back in the original array which has the same complexity. In total, its $O(n^3)$.

We can also do it in place if we allow for "partial" tableaus where only a portion of the top rows (and a portion of the last of them) is in the tableau. Then we can build the tableau in place and then start putting each minimal element to the end. This would be asymptotically equal, but use constant memory. It would also sort the array in reverse.

f. We start from the lower-left corner. We check the current element current with the one we're looking for key and move up if current $>$ key and right if current $<$ key. We declare success if current $=$ key and otherwise terminate if we walk off the tableau.

7 Quicksort

7.1 Description of quicksort

7.1-1

Using figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.

$$\begin{gathered} \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle \\ \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle \\ \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle \\ \langle 9, 19, 13, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle \\ \langle 9, 5, 13, 19, 12, 8, 7, 4, 21, 2, 6, 11 \rangle \\ \langle 9, 5, 13, 19, 12, 8, 7, 4, 21, 2, 6, 11 \rangle \\ \langle 9, 5, 8, 19, 12, 13, 7, 4, 21, 2, 6, 11 \rangle \\ \langle 9, 5, 8, 7, 12, 13, 19, 4, 21, 2, 6, 11 \rangle \\ \langle 9, 5, 8, 7, 4, 13, 19, 12, 21, 2, 6, 11 \rangle \\ \langle 9, 5, 8, 7, 4, 13, 19, 12, 21, 2, 6, 11 \rangle \\ \langle 9, 5, 8, 7, 4, 2, 19, 12, 21, 13, 6, 11 \rangle \\ \langle 9, 5, 8, 7, 4, 2, 6, 12, 21, 13, 19, 11 \rangle \\ \langle 9, 5, 8, 7, 4, 2, 6, 11, 21, 13, 19, 12 \rangle \end{gathered}$$

7.1-2

What value of q does PARTITION return when all elements in the array $A[p..r]$ have the same value? Modify PARTITION so that $q = \lfloor (p + r)/2 \rfloor$ when all elements in the array $A[p..r]$ have the same value.

It returns r.

We can modify PARTITION by counting the number of comparisons in which $A[j] = A[r]$ and then subtracting half that number from the pivot index.

7.1-3

Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

There is a for statement whose body executes $r - 1 - p = \Theta(n)$ times. In the worst case every time the body of the if is executed, but it takes constant time and so does the code outside of the loop. Thus the running time is $\Theta(n)$.

7.1-4

How would you modify QUICKSORT to sort into nonincreasing order?

We only need to flip the condition on line 4.

7.2 Performance of quicksort

7.2-1

Use the substitution method to prove that the recurrence $T(n) = T(n - 1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$, as claimed at the beginning of section 7.2.

We represent $\Theta(n)$ as $c_2 n$ and we guess that $T(n) \le c_1 n^2$,

$$\begin{aligned} T(n) &= T(n - 1) + c_2 n \\ &\le c_1 (n - 1)^2 + c_2 n \\ &= c_1 n^2 - 2c_1 n + c_1 + c_2 n \qquad (2c_1 > c_2, n \ge c_1/(2c_1 - c_2)) \\ &\le c_1 n^2. \end{aligned}$$

7.2-2

What is the running time of QUICKSORT when all elements of the array A have the same value?

It is $\Theta(n^2)$, since one of the partitions is always empty (see exercise 7.1-2.)

7.2-3

Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

If the array is already sorted in decreasing order, then, the pivot element is less than all the other elements. The partition step takes $\Theta(n)$ time, and then leaves you with a subproblem of size $n - 1$ and a subproblem of size 0. This gives us the recurrence considered in 7.2-1. Which we showed has a solution that is $\Theta(n^2)$.

7.2-4

Banks often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check numbers. People usually write checks in order by check number, and merchants usually cash the with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Argue that the procedure INSERTION-SORT would tend to beat the procedure QUICKSORT on this problem.

The more sorted the array is, the less work insertion sort will do. Namely, INSERTION-SORT is $\Theta(n + d)$, where d is the number of inversions in the array. In the example above the number of inversions tends to be small so insertion sort will be close to linear.

On the other hand, if PARTITION does pick a pivot that does not participate in an inversion, it will produce an empty partition. Since there is a small number of inversions, QUICKSORT is very likely to produce empty partitions.

7.2-5

Suppose that the splits at every level of quicksort are in proportion $1 - \alpha$ to α, where $0 < \alpha \le 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately $-\lg n / \lg \alpha$ and the maximum depth is approximately $-\lg n / \lg(1 - \alpha)$. (Don't worry about integer round-off.)

The minimum depth corresponds to repeatedly taking the smaller subproblem, that is, the branch whose size is proportional to α. Then, this will fall to 1 in k steps where $1 \approx \alpha^k n$. Therefore, $k \approx \log_\alpha 1/n = -\frac{\lg n}{\lg \alpha}$. The longest depth corresponds to always taking the larger subproblem. we then have an identical expression, replacing α with $1 - \alpha$.

7.2-6 ⋆

Argue that for any constant $0 < \alpha \le 1/2$, the probability is approximately $1 - 2\alpha$ that on a random input array, PARTITION produces a split more balanced than $1 - \alpha$ to α.

In order to produce a worse split than $1 - \alpha$ to α, PARTITION must pick a pivot that will be either within the smallest αn elements or the largest αn elements. The probability of either is (approximately) $\alpha n/n = \alpha$ and the probability of both is 2α. Thus, the probability of having a better partition is the complement, $1 - 2\alpha$.

7.3 A randomized version of quicksort

7.3-1

Why do we analyze the expected running time of a randomized algorithm and not its worst-case running time?

We analyze the expected run time because it represents the more typical time cost. Also, we are doing the expected run time over the possible randomness used during computation because it can't be produced adversarially, unlike when doing expected run time over all possible inputs to the algorithm.

7.3-2

When RANDOMIZED-QUICKSORT runs, how many calls are made to the random number generator RANDOM in the worst case? How about in the best case? Give your answer in terms of Θ-notation.

In the worst case, the number of calls to RANDOM is

$$T(n) = T(n - 1) + 1 = n = \Theta(n).$$

As for the best case,

$$T(n) = 2T(n/2) + 1 = \Theta(n).$$

This is not too surprising, because each third element (at least) gets picked as pivot.

7.4 Analysis of quicksort

7.4-1

Show that in the recurrence

$$\begin{aligned} T(n) &= \max_{0 \le q \le n-1} (T(q) + T(n - q - 1)) + \Theta(n), \\ T(n) &= \Omega(n^2). \end{aligned}$$

We guess $T(n) \ge cn^2 - 2n$,

$$\begin{aligned} T(n) &= \max_{0 \le q \le n-1} (T(q) + T(n - q - 1)) + \Theta(n) \\ &\ge \max_{0 \le q \le n-1} (cq^2 - 2q + c(n - q - 1)^2 - 2n - 2q - 1) + \Theta(n) \\ &\ge c \max_{0 \le q \le n-1} (q^2 + (n - q - 1)^2 - (2n + 4q + 1)/c) + \Theta(n) \\ &\ge cn^2 - c(2n - 1) + \Theta(n) \\ &\ge cn^2 - 2cn + 2c \qquad (c \le 1) \\ &\ge cn^2 - 2n. \end{aligned}$$

7.4-2

Show that quicksort's best-case running time is $\Omega(n \lg n)$.

We'll use the substitution method to show that the best-case running time is $\Omega(n \lg n)$. Let $T(n)$ be the best-case time for the procedure QUICKSORT on an input of size n. We have

$$T(n) = \min_{1 \le q \le n-1} (T(q) + T(n - q - 1)) + \Theta(n).$$

Suppose that $T(n) \ge c(n \lg n + 2n)$ for some constant c. Substituting this guess into the recurrence gives

$$\begin{aligned} T(n) &\ge \min_{1 \le q \le n-1} (cq \lg q + 2cq + c(n - q - 1) \lg(n - q - 1) + 2c(n - q - 1)) + \Theta(n) \\ &= (cn/2) \lg(n/2) + cn + c(n/2 - 1) \lg(n/2 - 1) + cn - 2c + \Theta(n) \\ &\ge (cn/2) \lg n - cn/2 + c(n/2 - 1)(\lg n - 2) + 2cn - 2c\Theta(n) \\ &= (cn/2) \lg n - cn/2 + (cn/2) \lg n - cn - c \lg n + 2c + 2cn - 2c\Theta(n) \\ &= cn \lg n + cn/2 - c \lg n + 2c - 2c\Theta(n). \end{aligned}$$

Taking a derivative with respect to q shows that the minimum is obtained when $q = n/2$. Taking c large enough to dominate the $-\lg n + 2 - 2c + \Theta(n)$ term makes this greater than $cn \lg n$, proving the bound.

7.4-3

> Show that the expression $q^2 + (n-q-1)^2$ achieves a maximum over $q = 0, 1, \ldots, n-1$ when $q = 0$ and $q = n-1$.

$$f(q) = q^2 + (n-q-1)^2$$
$$f'(q) = 2q - 2(n-q-1) = 4q - 2n + 2$$
$$f''(q) = 4.$$

$f'(q) = 0$ when $q = \frac{1}{2}n - \frac{1}{2}$. $f'(q)$ is also continious. $\forall q : f''(q) > 0$, which means that $f'(q)$ is negative left of $f'(q) = 0$ and positive right of it, which means that this is a local minima. In this case, $f(q)$ is decreasing in the beginning of the interval and increasing in the end, which means that those two points are the only candidates for a maximum in the interval.

$$f(0) = (n-1)^2$$
$$f(n-1) = (n-1)^2 + 0^2.$$

7.4-4

> Show that RANDOMIZED-QUICKSORT's expected running time is $\Omega(n \lg n)$.

We use the same reasoning for the expected number of comparisons, we just take in a different direction.

$$\begin{aligned} E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} && (k \geq 1) \\ &\geq \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{2k} \\ &\geq \sum_{i=1}^{n-1} \Omega(\lg n) \\ &= \Omega(n \lg n). \end{aligned}$$

Using the master method, we get the solution $\Theta(n \lg n)$.

7.4-5

> We can improve the running time of quicksort in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. Upon calling quicksort on a subarray with fewer than k elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process. Argue that this sorting algorithm runs in $O(nk + n \lg(n/k))$ expected time. How should we pick k, both in theory and practice?

In the quicksort part of the proposed algorithm, the recursion stops at level $\lg(n/k)$, which makes the expected running time $O(n \lg(n/k))$. However, this leaves n/k non-sorted, non - intersecting subarrays of (maximum) length k.

Because of the nature of the insertion sort algorithm, it will first sort fully one such subarray before consider the next one. Thus, it has the same complexity as sorting each of those arrays, that is $\frac{n}{k}O(k^2) = O(nk)$.

In theory, if we ignore the constant factors, we need to solve

$$\begin{aligned} & n \lg n \geq nk + n \lg n/k \\ \Rightarrow & \lg n \geq k + \lg n - \lg k \\ \Rightarrow & \lg k \geq k. \end{aligned}$$

Which is not possible.

If we add the constant factors, we get

$$\begin{aligned} & c_q n \lg n \geq c_i nk + c_q n \lg(n/k) \\ \Rightarrow & c_q \lg n \geq c_i k + c_q \lg n - c_q \lg k \\ \Rightarrow & \lg k \geq \frac{c_i}{c_q} k. \end{aligned}$$

Which indicates that there might be a good candidate. Furthermore, the lower-order terms should be taken into consideration too.

In practice, k should be chosed by experiment.

7.4-6 *

> Consider modifying the PARTITION procedure by randomly picking three elements from array A and partitioning about their median (the middle value of the three elements). Approximate the probability of getting at worst an α-to-$(1-\alpha)$ split, as a function of α in the range $0 < \alpha < 1$.

First, for simplicity's sake, let's assume that we can pick the same element twice. Let's also assume that $0 < \alpha \leq 1/2$.

In order to get such a split, two out of three elements need to be in the smallest αn elements. The probability of having one is $\alpha n/n = \alpha$. The probability of having exactly two is $\alpha^2 - \alpha^3$. There are three ways in which two elements can be in the smallest αn and one way in which all three can be in the smallest αn so the probability of getting such a median is $3\alpha^2 - 2\alpha^3$. We will get the same split if the median is in the largest αn. Since the two events are mutually exclusive, the probability is

$$\Pr\{\text{OK split}\} = 6\alpha^2 - 4\alpha^3 = 2\alpha^2(3 - 2\alpha).$$

Problem 7-1 Hoare partition correctness

> The version of PARTITION given in this chapter is not the original partitioning algorithm. Here is the original partition algorithm, which is due to C.A.R. Hoare:

```
HOARE-PARTITION(A, p, r)
    x = A[p]
    i = p - 1
    j = r + 1
    while true
        repeat
            j = j - 1
        until A[j] ≤ x
        repeat
            i = i + 1
        until A[i] ≥ x
        if i < j
            exchange A[i] with A[j]
        else return j
```

a. Demonstrate the operation of HOARE-PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$, showing the values of the array and auxiliary values after each iteration of the **while** loop in lines 4-13.

The next three questions ask you to give a careful argument that the procedure HOARE-PARTITION is correct. Assuming that the subarray $A[p..r]$ contains at least two elements, prove the following:

b. The indices i and j are such that we never access an element of A outside the subarray $A[p..r]$.

c. When HOARE-PARTITION terminates, it returns a value j such that $p \leq j < r$.

d. Every element of $A[p..j]$ is less than or equal to every element of $A[j+1..r]$ when HOARE-PARTITION terminates.

The PARTITION procedure in section 7.1 separates the pivot value (originally in $A[r]$) from the two partitions it forms. The HOARE-PARTITION procedure, on the other hand, always places the pivot value (originally in $A[p]$) into one of the two paritittions $A[p..j]$ and $A[j+1..r]$. Since $p \leq j < r$, this split is always nontrivial.

e. Rewrite the QUICKSORT procedure to use HOARE-PARTITION.

a. After the end of the loop, the variables have the following values: $x = 13$, $j = 9$ and $i = 10$.

b. Because when HOARE-PARTITION is running, $p \leq i < j \leq r$ will always hold, i, j won't access any element of A outside the subarray $A[p..r]$.

c. When $i \geq j$, HOARE-PARTITION terminates, so $p \leq j < r$.

d. When HOARE-PARTITION terminates, $A[p..j] \leq x \leq A[j+1..r]$.

e.

```
HOARE-QUICKSORT(A, p, r)
    if p < r
        q = HOARE-PARTITION(A, p, r)
        HOARE-QUICKSORT(A, p, q)
        HOARE-QUICKSORT(A, q + 1, r)
```

Problem 7-2 Quicksort with equal element values

> The analysis of the expected running time of randomized quicksort in section 7.4.2 assumes that all element values are distinct. In this problem. we examine what happens when they are not.
>
> **a.** Suppose that all element values are equal. What would be randomized quick-sort's running time in this case?
>
> **b.** The PARTITION procedure returns an index q such that each element of $A[p..q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1..r]$ is greater than $A[q]$. Modify the PARTITION procedure to produce a procedure PARTITION$'(A, p, r)$ which permutes the elements of $A[p..r]$ and returns two indices q and t where $p \leq q \leq t \leq r$, such that
>
> - all elements of $A[q..t]$ are equal,
> - each element of $A[p..q-1]$ is less than $A[q]$, and
> - each element of $A[t+1..r]$ is greater than $A[q]$.
>
> Like PARTITION, your PARTITION$'$ procedure should take $\Theta(r-p)$ time.
>
> **c.** Modify the RANDOMIZED-QUICKSORT procedure to call PARTITION$'$, and name the new procedure RANDOMIZED-QUICKSORT$'$. Then modify the QUICKSORT procedure to produce a procedure QUICKSORT$'(p, r)$ that calls RANDOMIZED-PARTITION$'$ and recurses only on partitions of elements not know to be equal to each other.
>
> **d.** Using QUICKSORT$'$, how would you adjust the analysis of section 7.4.2 to avoid the assumption that all elements are distinct?

a. Since all elements are equal, RANDOMIZED-QUICKSORT always returns $q = r$. We have recurrence $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.

b.

```
PARTITION'(A, p, r)
    x = A[p]
    low = p
    high = p
    for j = p + 1 to r
        if A[j] < x
            y = A[j]
            A[j] = A[high + 1]
            A[high + 1] = A[low]
            A[low] = y
            low = low + 1
            high = high + 1
        else if A[j] == x
            exchange A[high + 1] with A[j]
            high = high + 1
    return (low, high)
```

c.

```
QUICKSORT'(A, p, r)
    if p < r
        (low, high) = RANDOMIZED-PARTITION'(A, p, r)
        QUICKSORT'(A, p, low - 1)
        QUICKSORT'(A, high + 1, r)
```

d. Since we don't recurse on elements equal to the pivot, the subproblem sizes with QUICKSORT$'$ are no larger than the subproblem sizes with QUICKSORT when all elements are distinct.

Problem 7-3 Alternative quicksort analysis

> An alternative analysis of the running time of randomized quicksort focuses on the expected running time of each individual recursive call to RANDOMIZED-QUICKSORT, rather than on the number of comparisons performed.
>
> **a.** Argue that, given an array of size n, the probability that any particular element is chosen as the pivot is $1/n$. Use this to define indicator random variables
>
> $$X_i = I\{\text{ith smallest element is chosen as the pivot}\}.$$
>
> What is $E[X_i]$?
>
> **b.** Let $T(n)$ be a random variable denoting the running time of quicksort on an array of size n. Argue that
>
> $$E[T(n)] = E\left[\sum_{q=1}^{n} X_q (T(q-1) + T(n-q) + \Theta(n))\right]. \tag{7.5}$$
>
> **c.** Show that we can rewrite equation (7.5) as
>
> $$E[T(n)] = \frac{2}{n} \sum_{q=2}^{n-1} E[T(q)] + \Theta(n). \tag{7.6}$$

> **d.** Show that
>
> $$\sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2. \tag{7.7}$$
>
> (Hint: Split the summation into two parts, one for $k = 2, 3, \ldots, \lceil n/2 \rceil - 1$ and one for $k = \lceil n/2 \rceil, \ldots, n-1$.)
>
> **e.** Using the bound from equation (7.7), show that the recurrence in equation (7.6) has the solution $E[T(n)] = \Theta(n \lg n)$. (Hint: Show, by substitution, that $E[T(n)] \leq an \lg n$ for sufficiently large n and for some positive constant a.)

a. Since the pivot is selected as a random element in the array, which has size n, the probabilities of any particular element being selected are all equal, and add to one, so, are all $\frac{1}{n}$. As such, $E[X_i] = \Pr\{i \text{ smallest is picked}\} = \frac{1}{n}$.

b. We can apply linearity of expectation over all of the events X_i. Suppose we have a particular X_i be true, then, we will have one of the sub arrays be length $i-1$, and the other be $n-i$, and will of course still need linear time to run the partition procedure. This corresponds exactly to the summand in equation (7.5).

c.

$$\begin{aligned} & E\left[\sum_{q=1}^{n} X_q (T(q-1) + T(n-q) + \Theta(n))\right] \\ &= \sum_{q=1}^{n} E[X_q (T(q-1) + T(n-q) + \Theta(n))] \\ &= \sum_{q=1}^{n} (T(q-1) + T(n-q) + \Theta(n))/n \\ &= \Theta(n) + \frac{1}{n} \sum_{q=1}^{n} (T(q-1) + T(n-1)) \\ &= \Theta(n) + \frac{1}{n} \left(\sum_{q=1}^{n} T(q-1) + \sum_{q=1}^{n} T(n-q) \right) \\ &= \Theta(n) + \frac{1}{n} \left(\sum_{q=1}^{n} T(q-1) + \sum_{q=1}^{n} T(q-1) \right) \\ &= \Theta(n) + \frac{2}{n} \sum_{q=1}^{n} T(q-1) \\ &= \Theta(n) + \frac{2}{n} \sum_{q=0}^{n-1} T(q) \\ &= \Theta(n) + \frac{2}{n} \sum_{q=2}^{n-1} T(q). \end{aligned}$$

d. We will prove this inequality in a different way than suggested by the hint. If we let $f(k) = k \lg k$ treated as a continuous function, then $f'(k) = \lg k + 1$. Note now that the summation written out is the left hand approximation of the integral of $f(k)$ from 2 to n with step size 1. By integration by parts, the anti-derivative of $k \lg k$ is

$$\frac{1}{\lg 2}(\frac{k^2}{2}\ln k - \frac{k^2}{4}).$$

So, plugging in the bounds and subtracting, we get $\frac{n^2 \lg n}{2} - \frac{n^2}{4 \ln 2} - 1$. Since f has a positive derivative over the entire interval that the integral is being evaluated over, the left hand rule provides a underapproximation of the integral, so, we have that

$$\begin{aligned}\sum_{k=2}^{n-1} k \lg k &\leq \frac{n^2 \lg n}{2} - \frac{n^2}{4 \ln 2} - 1 \\ &\leq \frac{n^2 \lg n}{2} - \frac{n^2}{8},\end{aligned}$$

where the last inequality uses the fact that $\ln 2 > 1/2$.

e. Assume by induction that $T(q) \leq q \lg(q) + \Theta(n)$. Combining (7.6) and (7.7), we have

$$\begin{aligned}E[T(n)] &= \frac{2}{n}\sum_{q=2}^{n-1} E[T(q)] + \Theta(n) \\ &\leq \frac{2}{n}\sum_{q=2}^{n-1}(q \lg q + \Theta(n)) + \Theta(n) \\ &\leq \frac{2}{n}\sum_{q=2}^{n-1} q \lg q + \frac{2}{n}\Theta(n) + \Theta(n) \\ &\leq \frac{2}{n}(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2) + \Theta(n) \\ &= n \lg n - \frac{1}{4}n + \Theta(n) \\ &= n \lg n + \Theta(n).\end{aligned}$$

Problem 7-4 Stack depth for quicksort

The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the left subarray and then it recursively sorts the right subarray. The second recursive call in QUICKSORT is not really necessary; we can avoid it by using an iterative control structure. This technique, called ***tail recursion***, is provided automatically by good compilers. Consider the following version of quicksort, which simulates tail recursion:

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
    while p < r
        // Partition and sort left subarray.
        q = PARTITION(A, p, r)
        TAIL-RECURSIVE-QUICKSORT(A, p, q - 1)
        p = q + 1
```

a. Argue that TAIL-RECURSIVE-QUICKSORT$(A, 1, A.\text{length})$ correctly sorts the array A.

Compilers usually execute recursive procedures by using a ***stack*** that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. Upon calling a procedure, its information is ***pushed*** onto the stack; when it terminates, its information is ***popped***. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The ***stack depth*** is the maximum amount of stack space used at any time during a computation.

b. Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is $\Theta(n)$ on an n-element input array.

c. Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

a. The book proved that QUICKSORT correctly sorts the array A. TAIL-RECURSIVE-QUICKSORT differs from QUICKSORT in only the last line of the loop.

It is clear that the conditions starting the second iteration of the **while** loop in TAIL-RECURSIVE-QUICKSORT are identical to the conditions starting the second recursive call in QUICKSORT. Therefore, TAIL-RECURSIVE-QUICKSORT effectively performs the sort in the same manner as QUICKSORT. Therefore, TAIL-RECURSIVE-QUICKSORT must correctly sort the array A.

b. The stack depth will be $\Theta(n)$ if the input array is already sorted. The right subarray will always have size 0 so there will be $n - 1$ recursive calls before the **while**-condition $p < r$ is violated.

c.

```
MODIFIED-TAIL-RECURSIVE-QUICKSORT(A, p, r)
    while p < r
        q = PARTITION(A, p, r)
        if q < floor((p + r) / 2)
            MODIFIED-TAIL-RECURSIVE-QUICKSORT(A, p, q - 1)
            p = q + 1
        else
            MODIFIED-TAIL-RECURSIVE-QUICKSORT(A, q + 1, r)
            r = q - 1
```

Problem 7-5 Median-of-3 partition

One way to improve the RANDOMIZED-QUICKSORT procedure is to partition around a pivot that is chosen more carefully than by picking a random element from the subarray. One common approach is the ***median-of-3*** method: choose the pivot as the median (middle element) of a set of 3 elements randomly selected from the subarray. (See exercise 7.4-6.) For this problem, let us assume that the elements of the input array $A[1..n]$ are distinct and that $n \geq 3$. We denote the sorted output array by $A'[1..n]$. Using the median-of-3 method to choose the pivot element x, define $p_i = \Pr\{x = A'[i]\}$.

a. Give an exact formula for p_i as a function of n and i for $i = 2, 3, \ldots, n-1$. (Note that $p_1 = p_n = 0$.)

b. By what amount have we increased the likelihood of choosing the pivot as $x = A'[\lfloor (n+1)/2 \rfloor]$, the median of $A[1..n]$, compared with the ordinary implementation? Assume that $n \to \infty$, and give the limiting ratio of these probabilities.

c. If we define a "good" split to mean choosing the pivot as $x = A'[i]$, where $n/3 \leq i \leq 2n/3$, by what amount have we increased the likelihood of getting a good split compared with the ordinary implementation? (Hint: Approximate the sum by an integral.)

d. Argue that in the $\Omega(n \lg n)$ running time of quicksort, the median-of-3 method affects only the constant factor.

a. p_i is the probability that a randomly selected subset of size three has the $A'[i]$ as it's middle element. There are 6 possible orderings of the three elements selected. So, suppose that S' is the set of three elements selected.

We will compute the probability that the second element of S' is $A'[i]$ among all possible 3-sets we can pick, since there are exactly six ordered 3-sets corresponding to each 3-set, these probabilities will be equal. We will compute the probability that $S'[2] = A[i]$. For any such S', we would need to select the first element from $[i-1]$ and the third from $i+1, \ldots, n$. So, there are $(i-1)(n-i)$ such 3-sets. The total number of 3-sets is $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$. So,

$$p_i = \frac{6(n-i)(i-1)}{n(n-1)(n-2)}.$$

b. If we let $i = \lfloor \frac{n+1}{2} \rfloor$, the previous result gets us an increase of

$$\frac{6(\lfloor \frac{n-1}{2} \rfloor)(n - \lfloor \frac{n+1}{2} \rfloor)}{n(n-1)(n-2)} - \frac{1}{n}$$

in the limit n going to infinity, we get

$$\lim_{n\to\infty} \frac{\frac{6(\lfloor \frac{n+1}{2} \rfloor)(n-\lfloor \frac{n+1}{2} \rfloor)}{n(n-1)(n-2)}}{\frac{1}{n}} = \frac{3}{2}.$$

c. To save the messiness, suppose n is a multiple of 3. We will approximate the sum as an integral, so,

$$\begin{aligned}\sum_{i=n/3}^{2n/3} &\approx \int_{n/3}^{2n/3} \frac{6(-x^2 + nx + x - n)}{n(n-1)(n-2)} dx \\ &= \frac{6(-7n^3/81 + 3n^3/18 + 3n^2/18 - n^2/3)}{n(n-1)(n-2)},\end{aligned}$$

which, in the limit n goes to infinity, is $\frac{13}{27}$ which is a constant that $> \frac{1}{3}$ as it was in the original randomized quicksort implementation.

d. Even though we always choose the middle element as the pivot (which is the best case), the height of the recursion tree will be $\Theta(\lg n)$. Therefore, the running time is still $\Omega(n \lg n)$.

Problem 7-6 Fuzzy sorting of intervals

Consider the problem in which we do not know the numbers exactly. Instead, for each number, we know an interval on the real line to which it belongs. That is, we are given n closed intervals of the form $[a_i, b_i]$, where $a_i \leq b_i$. We wish to ***fuzzy-sort*** these intervals, i.e., to produce a permutation $\langle i_1, i_2, \ldots, i_n \rangle$ of the intervals such that for $j = 1, 2, \ldots, n$, there exists $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \leq c_2 \leq \cdots \leq c_n$.

a. Design a randomized algorithm for fuzzy-sorting n intervals. Your algorithm should have the general structure of an algorithm that quicksorts the left endpoints (the a_i values), but it should take advantage of overlapping intervals to improve the running time. (As the intervals overlap more and more, the problem of fuzzy-sorting the intervals becoes progressively easier. Your algorithm should take advantage of such overlapping, to the extend that it exists.)

b. Argue that your algorithm runs in expected time $\Theta(n \lg n)$ in general, but runs in expected time $\Theta(n)$ when all of the intervals overlap (i.e., when there exists a value x such that $x \in [a_i, b_i]$ for all i). Your algorithm should not be checking for this case explicitly; rather, its performance should naturally improve as the amount of overlap increases.

a. With randomly selected left endpoint for the pivot, we could trivially perform fuzzy sorting by quicksorting the left endpoints, a_i's. This would achieve the worst-case expected running time of $\Theta(n \lg n)$. We definitely can do better by exploit the characteristic that we don't have to sort overlapping intervals. That is, for two overlapping intervals, $[a_i, b_i]$ and $[a_j, b_j]$. In such situations, we can always choose $\{c_i, c_j\}$ (within the intersection of these intervals) such that $c_i \leq c_j$ or $c_j \leq c_i$.

Since overlapping intervals do not require sorting, we can improve the expected running time by modifying quicksort to identify overlaps:

```
FIND-INTERSECTION(A, p, r)
    rand = RANDOM(p, r)
    exchange A[rand] with A[r]
    a = A[r].a
    b = A[r].b
    for i = p to r - 1
        if A[i].a ≤ b and A[i].b ≥ a
            if A[i].a > a
                a = A[i].a
            if A[i].b < b
                b = A[i].b
    return (a, b)
```

On lines 2 through 3 of FIND-INTERSECTION, we select a random *pivot interval* as the initial region of overlap $[a, b]$. There are two situations:

- If the intervals are all disjoint, then the estimated region of overlap will be this randomly-selected interval;
- otherwise, on lines 6 through 11, we loop through all intervals in arrays A (except the endpoint which is the initial pivot interval). At each iteration, we determine if the current interval overlaps the current estimated region of overlap. If it does, we update the estimated region of overlap as $[a, b] = [a_i, b_i] \cap [a, b]$.

FIND-INTERSECTION has a worst-case running time $\Theta(n)$ since we evaluate the intersection from index 1 to $A.\text{length}$ of the array.

We can extend the QUICKSORT to allow fuzzy sorting using FIND-INTERSECTION.

First, partition the input array into "left", "middle", and "right" subarrays. The "middle" subarray elements overlap the interval $[a, b]$ found by FIND-INTERSECTION. As a result, they can appear in any order in the output.

We recursively call FUZZY-SORT on the "left" and "right" subarrays to produce a fuzzy sorted array in-place. The following pseudocode implements these basic operations. One can run FUZZY-SORT$(A, 1, A.\text{length})$ to fuzzy-sort an array.

The first and last elements in a subarray are indexed by p and r, respectively. The index of the first and last intervals in the "middle" region are indexed by q and t, respectively.

```
FUZZY-SORT(A, p, r)
    if p < r
        (a, b) = FIND-INTERSECTION(A, p, r)
        t = PARTITION-RIGHT(A, a, p, r)
        q = PARTITION-LEFT(A, b, p, t)
        FUZZY-SORT(A, p, q - 1)
        FUZZY-SORT(A, t + 1, r)
```

We need to determine how to partition the input arrays into "left", "middle", and "right" subarrays in-place.

First, we PARTITION-RIGHT the entire array from p to r using a pivot value equal to the left endpoint a found by FIND-INTERSECTION, such that $a_i \leq a$.

Then, we PARTITION-LEFT the subarray from p to t using a pivot value equal to the right endpoint b found by FIND-INTERSECTION, such that $b_i < b$.

```
PARTITION-RIGHT(A, a, p, r)
    i = p - 1
    for j = p to r - 1
        if A[j].a ≤ a
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```

```
PARTITION-LEFT(A, b, p, t)
    i = p - 1
    for j = p to t - 1
        if A[j].b < b
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[t]
    return i + 1
```

The FUZZY-SORT is similar to the randomized quicksort presented in the textbook. In fact, PARTITION-RIGHT and PARTITION-LEFT are nearly identical to the PARTITION procedure on page 171. The primary difference is the value of the pivot used to sort the intervals.

b. We expect FUZZY-SORT to have a worst-case running time $\Theta(n \lg n)$ for a set of input intervals which do not overlap each other. First, notice that lines 2 through 3 of FIND-INTERSECTION select a *random interval* as the initial pivot interval. Recall that if the intervals are disjoint, then $[a, b]$ will simply be this initial interval.

Since for this example there are no overlaps, the "middle" region created by lines 4 and 5 of FUZZY-SORT will only contain the initially-selected interval. In general, line 3 is $\Theta(n)$. Fortunately, since the pivot interval $[a, b]$ is randomly-selected, the expected sizes of the "left" and "right" subarrays are both $\lfloor \frac{n}{2} \rfloor$. In conclusion, the reccurrence of the running time is

$$\begin{aligned}T(n) &\leq 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n).\end{aligned}$$

The FIND-INTERSECTION will always return a non-empty region of overlap $[a, b]$ containing x if the intervals all overlap at x. For this situation, every interval will be within the "middle" region since the "left" and "right" subarrays will be empty, lines 6 and 7 of FUZZY-SORT are $\Theta(1)$. As a result, there is no recursion and

the running time of FUZZY-SORT is determined by the $\Theta(n)$ running time required to find the region of overlap. Therfore, if the input intervals all overlap at a point, then the expected worst-case running time is $\Theta(n)$.

8 Sorting in Linear Time

8.1 Lower bounds for sorting

8.1-1

> What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

For a permutation $a_1 \le a_2 \le \ldots \le a_n$, there are $n-1$ pairs of relative ordering, thus the smallest possible depth is $n-1$.

8.1-2

> Obtain asymptotically tight bounds on $\lg(n!)$ without using Stirling's approximation. Instead, evaluate the summation $\sum_{k=1}^{n} \lg k$ using techniques from Section A.2.

$$\begin{aligned}\sum_{k=1}^{n} \lg k &\le \sum_{k=1}^{n} \lg n \\ &= n \lg n.\end{aligned}$$

$$\begin{aligned}\sum_{k=1}^{n} \lg k &= \sum_{k=2}^{n/2} \lg k + \sum_{k=n/2}^{n} \lg k \\ &\ge \sum_{k=1}^{n/2} 1 + \sum_{k=n/2}^{n} \lg n/2 \\ &= \frac{n}{2} + \frac{n}{2}(\lg n - 1) \\ &= \frac{n}{2} \lg n.\end{aligned}$$

8.1-3

> Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

Consider a decision tree of height h with r reachable leaves corresponding to a comparison sort on n elements. From **Theorem 8.1**, We have $n!/2 \le n! \le r \le 2^h$. By taking logarithms, we have

$$h \ge \lg(n!/2) = \lg(n!) - 1 = \Theta(n \lg n) - 1 = \Theta(n \lg n).$$

From the equation above, there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n.

Consider the $1/n$ of inputs of length n condition. we have $(1/n)n! \le n! \le r \le 2^h$. By taking logarithms, we have

$$h \ge \lg(n!/n) = \lg(n!) - \lg n = \Theta(n \lg n) - \lg n = \Theta(n \lg n).$$

From the equation above, there is no comparison sort whose running time is linear for $1/n$ of the $n!$ inputs of length n.

Consider the $1/2^n$ of inputs of length n condition. we have $(1/2^n)n! \le n! \le r \le 2^h$. By taking logarithms, we have

$$h \ge \lg(n!/2^n) = \lg(n!) - n = \Theta(n \lg n) - n = \Theta(n \lg n).$$

From the equation above, there is no comparison sort whose running time is linear for $1/2^n$ of the $n!$ inputs of length n.

8.1-4

> Suppose that you are given a sequence of n elements to sort. The input sequence consists of n/k subsequences, each containing k elements. The elements in a given subsequence are all smaller than the elements in the succeeding subsequence and larger than the elements in the preceding subsequence. Thus, all that is needed to sort the whole sequence of length n is to sort the k elements in each of the n/k subsequences. Show an $\Omega(n \lg k)$ lower bound on the number of comparisons needed to solve this variant of the sorting problem. (Hint: It is not rigorous to simply combine the lower bounds for the individual subsequences.)

Assume that we need to construct a binary decision tree to represent comparisons. Since length of each subsequece is k, there are $(k!)^{n/k}$ possible output permutations. To compute the height h of the decision tree, we must have $(k!)^{n/k} \le 2^h$. Taking logs on both sides, we know that

$$h \ge \frac{n}{k} \times \lg(k!) \ge \frac{n}{k} \times \left(\frac{k \ln k - k}{\ln 2}\right) = \frac{n \ln k - n}{\ln 2} = \Omega(n \lg k).$$

8.2 Counting sort

8.2-1

> Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the array $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

We have that $C = \langle 2, 4, 6, 8, 9, 9, 11 \rangle$. Then, after successive iterations of the loop on lines 10-12, we have

$$\begin{aligned}B &= \langle , , , , , 2, , , , , \rangle, \\ B &= \langle , , , , , 2, 3, , , \rangle, \\ B &= \langle , , 1, , 2, 3, , , \rangle\end{aligned}$$

and at the end,

$$B = \langle 0, 0, 1, 1, 2, 2, 3, 3, 4, 6, 6 \rangle.$$

8.2-2

> Prove that COUNTING-SORT is stable.

Suppose positions i and j with $i < j$ both contain some element k. We consider lines 10 through 12 of COUNTING-SORT, where we construct the output array. Since $j > i$, the loop will examine $A[j]$ before examining $A[i]$. When it does so, the algorithm correctly places $A[j]$ in position $m = C[k]$ of B. Since $C[k]$ is decremented in line 12, and is never again incremented, we are guaranteed that when the **for** loop examines $A[i]$ we will have $C[k] < m$. Therefore $A[i]$ will be placed in an earlier position of the output array, proving stability.

8.2-3

> Suppose that we were to rewrite the **for** loop header in line 10 of the COUNTING-SORT as
>
> ```
> 10  for j = 1 to A.length
> ```
>
> Show that the algorithm still works properly. Is the modified algorithm stable?

The algorithm still works correctly. The order that elements are taken out of C and put into B doesn't affect the placement of elements with the same key. It will still fill the interval $(C[k-1], C[k]]$ with elements of key k. The question of whether it is stable or not is not well phrased. In order for stability to make sense, we would need to be sorting items which have information other than their key, and the sort as written is just for integers, which don't. We could think of extending this algorithm by placing the elements of A into a collection of elements for each cell in array C. Then, if we use a FIFO collection, the modification of line 10 will make it stable, if we use LILO, it will be anti-stable.

8.2-4

> Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a..b]$ in $O(1)$ time. Your algorithm should use $\Theta(n+k)$ preprocessing time.

The algorithm will begin by preprocessing exactly as COUNTING-SORT does in lines 1 through 9, so that $C[i]$ contains the number of elements less than or equal to i in the array. When queried about how many integers fall into a range $[a..b]$, simply compute $C[b] - C[a-1]$. This takes $O(1)$ times and yields the desired output.

8.3 Radix sort

8.3-1

> Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

0	1	2	3
COW	SE**A**	TA**B**	**B**AR
DOG	TE**A**	BA**R**	**B**IG
SEA	MO**B**	EA**R**	**B**OX
RUG	TA**B**	TA**R**	**C**OW
ROW	DO**G**	SE**A**	**D**IG
MOB	RU**G**	TE**A**	**D**OG
BOX	DI**G**	DI**G**	**E**AR
TAB	BI**G**	BI**G**	**F**OX
BAR	BA**R**	MO**B**	**M**OB
EAR	EA**R**	DO**G**	**N**OW
TAR	TA**R**	CO**W**	**R**OW
DIG	CO**W**	RO**W**	**R**UG
BIG	RO**W**	NO**W**	**S**EA
TEA	NO**W**	BO**X**	**T**AB
NOW	BO**X**	FO**X**	**T**AR
FOX	FO**X**	RU**G**	**T**EA

8.3-2

> Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any sorting algorithm stable. How much additional time and space does your scheme entail?

Insertion sort and merge sort are stable. Heapsort and quicksort are not.

To make any sorting algorithm stable we can preprocess, replacing each element of an array with an ordered pair. The first entry will be the value of the element, and the second value will be the index of the element.

For example, the array $[2, 1, 1, 3, 4, 4, 4]$ would become $[(2, 1), (1, 2), (1, 3), (3, 4), (4, 5), (4, 6), (4, 7)]$. We now interpret $(i, j) < (k, m)$ if $i < k$ or $i = k$ and $j < m$. Under this definition of less-than, the algorithm is guaranteed to be stable because each of our new elements is distinct and the index comparison ensures that if a repeat element appeared later in the original array, it must appear later in the sorted array. This doubles the space requirement, but the running time will be asymptotically unchanged.

8.3-3

> Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?

Loop invariant: At the beginning of the **for** loop, the array is sorted on the last $i-1$ digits.

Initialization: The array is trivially sorted on the last 0 digits.

Maintenance: Let's assume that the array is sorted on the last $i-1$ digits. After we sort on the ith digit, the array will be sorted on the last i digits. It is obvious that elements with different digit in the ith position are ordered accordingly; in the case of the same ith digit, we still get a correct order, because we're using a stable sort and the elements were already sorted on the last $i-1$ digits.

Termination: The loop terminates when $i = d + 1$. Since the invariant holds, we have the numbers sorted on d digits.

8.3-4

> Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

First run through the list of integers and convert each one to base n, then radix sort them. Each number will have at most $\log_n n^3 = 3$ digits so there will only need to be 3 passes. For each pass, there are n possible values which can be taken on, so we can use counting sort to sort each digit in $O(n)$ time.

8.3-5 ⋆

> In the first card-sorting algorithm in this section, exactly how many sorting passes are needed to sort d-digit decimal numbers in the worst case? How many piles of cards would an operator need to keep track of in the worst case?

Given n d-digit numbers in which each digit can take on up to k possible values, we'll perform $\Theta(k^d)$ passes and keep track of $\Theta(nk)$ piles in the worst case.

8.4 Bucket sort

8.4-1

> Using Figure 8.4 as a model, illustrate the operation of BUCKET-SORT on the array $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$.

R
0
1 .13.16
2 .20
3 .39
4 .42
5 .53
6 .64
7 .79.71
8 .89
9

$$A = \langle .13, .16, .20, .39, .42, .53, .64, .71, .79, .89 \rangle.$$

8.4-2

> Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

If all the keys fall in the same bucket and they happen to be in reverse order, we have to sort a single bucket with n items in reversed order with insertion sort. This is $\Theta(n^2)$.

We can use merge sort or heapsort to improve the worst-case running time. Insertion sort was chosen because it operates well on linked lists, which has optimal time and requires only constant extra space for short linked lists. If we use another sorting algorithm, we have to convert each list to an array, which might slow down the algorithm in practice.

8.4-3

> Let X be a random variable that is equal to the number of heads in two flips of a fair coin. What is $E[X^2]$? What is $E^2[X]$?

$$\begin{aligned} \square \ \& = 2 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} = 1 \\ \text E[X^2] \ \& = 4 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} = 1.5 \\ \square \ \& = \text E[X] \cdot \text E[X] = 1 \cdot 1 = 1. \end{aligned}$$

8.4-4 ⋆

> We are given n points in the unit circle, $p_i = (x_i, y_i)$, such that $0 < x_i^2 + y_i^2 \le 1$ for $i = 1, 2, \ldots, n$. Suppose that the points are uniformly distributed; that is, the probability of finding a point in any region of the circle is proportional to the area of that region. Design an algorithm with an average-case running time of $\Theta(n)$ to sort the n points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin. (Hint: Design the bucket sizes in BUCKET-SORT to reflect the uniform distribution of the points in the unit circle.)

Bucket sort by radius,

$$\pi r_i^2 = \frac{i}{n} \cdot \pi 1^2$$
$$r_i = \sqrt{\frac{i}{n}}.$$

8.4-5 ⋆

> A ***probability distribution function*** $P(x)$ for a random variable X is defined by $P(x) = \Pr\{X \le x\}$. Suppose that we draw a list of n random variables $X_1, X_2, \ldots, X_n$ from a continuous probability distribution function P that is computable in $O(1)$ time. Give an algorithm that sorts these numbers in linear average-case time.

Bucket sort by p_i, so we have n buckets: $[p_0, p_1), [p_1, p_2), \ldots, [p_{n-1}, p_n)$. Note that not all buckets are the same size, which is ok as to ensure linear run time, the inputs should on average be uniformly distributed amongst all buckets, of which the intervals defined with p_i will do so.

p_i is defined as follows:

$$P(p_i) = \frac{i}{n}.$$

Problem 8-1 Probabilistic lower bounds on comparison sorting

> In this problem, we prove a probabilistic $\Omega(n \lg n)$ lower bound on the running time of any deterministic or randomized comparison sort on n distinct input elements. We begin by examining a deterministic comparison sort A with decision tree T_A. We assume that every permutation of A's inputs is equally likely.
>
> **a.** Suppose that each leaf of T_A is labeled with the probability that it is reached given a random input. Prove that exactly $n!$ leaves are labeled $1/n!$ and that the rest are labeled 0.
>
> **b.** Let $D(T)$ denote the external path length of a decision tree T; that is, $D(T)$ is the sum of the depths of all the leaves of T. Let T be a decision tree with $k > 1$ leaves, and let LT and RT be the left and right subtrees of T. Show that $D(T) = D(LT) + D(RT) + k$.
>
> **c.** Let $d(k)$ be the minimum value of $D(T)$ over all decision trees T with $k > 1$ leaves. Show that $d(k) = \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$. (Hint: Consider a decision tree T with k leaves that achieves the minimum. Let i_0 be the number of leaves in LT and $k - i_0$ the number of leaves in RT.)
>
> **d.** Prove that for a given value of $k > 1$ and i in the range $1 \le i \le k - 1$, the function $i \lg i + (k - i) \lg(k - i)$ is minimized at $i = k/2$. Conclude that $d(k) = \Omega(k \lg k)$.
>
> **e.** Prove that $D(T_A) = \Omega(n! \lg(n!))$, and conclude that the average-case time to sort n elements is $\Omega(n \lg n)$.
>
> Now, consider a *randomized* comparison sort B. We can extend the decision-tree model to handle randomization by incorporating two kinds of nodes: ordinary comparison nodes and "randomization" nodes. A randomization node models a random choice of the form $\text{RANDOM}(1, r)$ made by algorithm B; the node has r children, each of which is equally likely to be chosen during an execution of the algorithm.
>
> **f.** Show that for any randomized comparison sort B, there exists a deterministic comparison sort A whose expected number of comparisons is no more than those made by B.

a. There are $n!$ possible permutations of the input array because the input elements are all distinct. Since each is equally likely, the distribution is uniformly supported on this set. So, each occurs with probability $\frac{1}{n!}$ and corresponds to a different leaf because the program needs to be able to distinguish between them.

b. The depths of particular elements of LT or RT are all one less than their depths when considered elements of T. In particular, this is true for the leaves of the two subtrees. Also, $\{LT, RT\}$ form a partition of all the leaves of T. Therefore, if we let $L(T)$ denote the leaves of T,

$$\begin{aligned} D(T) &= \sum_{\ell \in L(T)} D_T(\ell) \\ &= \sum_{\ell \in L(LT)} D_T(\ell) + \sum_{\ell \in L(RT)} D_T(\ell) \\ &= \sum_{\ell \in L(LT)} (D_{LT}(\ell) + 1) + \sum_{\ell \in L(RT)} (D_{RT}(\ell) + 1) \\ &= \sum_{\ell \in L(LT)} D_{LT}(\ell) + \sum_{\ell \in L(RT)} D_{RT}(\ell) + k \\ &= D(LT) + D(RT) + k. \end{aligned}$$

c. Suppose we have a T with k leaves so that $D(T) = d(k)$. Let i_0 be the number of leaves in LT. Then, $d(k) = D(T) = D(LT) + D(RT) + k$. However, we can pick LT and RT to minimize the external path length.

d. We treat i as a continuous variable, and take a derivative to find critical points. The given expression has the following as a derivative with respect to i

$$\frac{1}{\ln 2} + \lg i + \frac{1}{\ln 2} - \lg(k - i) = \frac{2}{\ln 2} + \lg\left(\frac{i}{k - i}\right),$$

which is 0 when we have $\frac{i}{k-i} = 2^{-\frac{2}{\ln 2}} = 2^{-\lg e^2} = e^{-2}$. Therefore, $(1 + e^{-2})i = k$, $i = \frac{k}{1+e^{-2}}$.

Since we are picking the two subtrees to be roughly equal size, the total depth will be order $\lg k$, with each level contributing k, so the total external path length is at least $k \lg k$.

e. Since before we that a tree with k leaves needs to have external length $k \lg k$, and that a sorting tree needs at least $n!$ trees, a sorting tree must have external tree length at least $n! \lg(n!)$. Since the average case run time is the depth of a leaf weighted by the probability of that leaf being the one that occurs, we have that the run time is at least $\frac{n! \lg(n!)}{n!} = \lg(n!) \in \Omega(n \lg n)$.

f. Since the expected runtime is the average over all possible results from the random bits, if every possible fixing of the randomness resulted in a higher runtime, the average would have to be higher as well.

Problem 8-2 Sorting in place in linear time

> Suppose that we have an array of n data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:
>
> 1. The algorithm runs in $O(n)$ time.
> 2. The algorithm is stable.
>
> 3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.
>
> **a.** Give an algorithm that satisfies criteria 1 and 2 above.
>
> **b.** Give an algorithm that satisfies criteria 1 and 3 above.
>
> **c.** Give an algorithm that satisfies criteria 2 and 3 above.
>
> **d.** Can you use any of your sorting algorithms from parts (a)–(c) as the sorting method used in line 2 of RADIX-SORT, so that RADIX-SORT sorts n records with b-bit keys in $O(bn)$ time? Explain how or why not.
>
> **e.** Suppose that the n records have keys in the range from 1 to k. Show how to modify counting sort so that it sorts the records in place in $O(n + k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable? (Hint: How would you do it for $k = 3$?)

a. Counting-Sort.

b. Quicksort-Partition.

c. Insertion-Sort.

d. (a) Yes. (b) No. (c) No.

e.

Thanks @Gutdub for providing the solution in this issue.

```
MODIFIED-COUNTING-SORT(A, k)
    let C[0..k] be a new array
    for i = 1 to k
        C[i] = 0
    for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
    for i = 2 to k
        C[i] = C[i] + C[i - 1]
    insert sentinel element NIL at the start of A
    B = C[0..k - 1]
    insert number 1 at the start of B
    // B now contains the "endpoints" for C
    for i = 2 to A.length
        while C[A[i]] != B[A[i]]
            key = A[i]
            exchange A[C[A[i]]] with A[i]
            while A[C[key]] == key // make sure that elements with the same
keys will not be swapped
                C[key] = C[key] - 1
    remove the sentinel element
    return A
```

In place (storage space is $\Theta(k)$) but not stable.

Problem 8-3 Sorting variable-length items

> **a.** You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over *all* the integers in the array is n. Show how to sort the array in $O(n)$ time.
>
> **b.** You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is n. Show how to sort the strings in $O(n)$ time.
>
> (Note that the desired order here is the standard alphabetical order; for example, $a < ab < b$.)

a. First, sort the integer according to their lengths by bucket sort, where we make a bucket for each possible number of digits. We sort each these uniform length sets of integers using radix sort. Then, we just concatenate the sorted lists obtained from each bucket.

b. Make a bucket for every letter in the alphabet, each containing the words that start with that letter. Then, forget about the first letter of each of the words in the bucket, concatenate the empty word (if it's in this new set of words) with the result of recursing on these words of length one less. Since each word is processed a number of times equal to it's length, the runtime will be linear in the total number of letters.

Problem 8-4 Water jugs

> Suppose that you are given n red and n blue water jugs, all of different shapes and sizes. All red jugs hold different amounts of water, as do the blue ones. Moreover, for every red jug, there is a blue jug that holds the same amount of water, and vice versa.
>
> Your task is to find a grouping of the jugs into pairs of red and blue jugs that hold the same amount of water. To do so, you may perform the following operation: pick a pair of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug. This operation will tell you whether the red or the blue jug can hold more water, or that they have the same volume. Assume that such a comparison takes one time unit. Your goal is to find an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.
>
> **a.** Describe a deterministic algorithm that uses $\Theta(n^2)$ comparisons to group the jugs into pairs.
>
> **b.** Prove a lower bound of $\Omega(n \lg n)$ for the number of comparisons that an algorithm solving this problem must make.
>
> **c.** Give a randomized algorithm whose expected number of comparisons is $O(n \lg n)$, and prove that this bound is correct. What is the worst-case number of comparisons for your algorithm?

a. Select a red jug. Compare it to blue jugs until you find one which matches. Set that pair aside, and repeat for the next red jug. This will use at most $\sum_{i=1}^{n-1} i = n(n-1)/2 = \Theta(n^2)$ comparisons.

b. We can imagine first lining up the red jugs in some order. Then a solution to this problem becomes a permutation of the blue jugs such that the ith blue jug is the same size as the ith red jug. As in section 8.1, we can make a decision tree which represents comparisons made between blue jugs and red jugs. An internal node represents a comparison between a specific pair of red and blue jugs, and a leaf node represents a permutation of the blue jugs based on the results of the comparison. We are interested in when one jug is greater than, less than, or equal in size to another jug, so the tree should have 3 children per node. Since there must be at least $n!$ leaf nodes, the decision tree must have height at least $\log_3(n!)$. Since a solution corresponds to a simple path from root to leaf, an algorithm must make at least $\Theta(n \lg n)$ comparisons to reach any leaf.

c. We use an algorithm analogous to randomized quicksort. Select a blue jug at random. Partition the red jugs into those which are smaller than the blue jug, and those which are larger. At some point in the comparisons, you will find the red jug which is of equal size. Once the red jugs have been divided by size, use the red jug of equal size to partition the blue jugs into those which are smaller and those which are larger. If k red jugs are smaller than the originally chosen jug, we need to solve the original problem on input of size $k - 1$ and size $n - k$, which we will do in the same manner. A subproblem of size 1 is trivially solved because if there is only one red jug and one blue jug, they must be the same size. The analysis of expected number of comparisons is exactly the same as that of RANDOMIZED-QUICKSORT given on pages 181-184. We are running the procedure twice so the expected number of comparisons is doubled, but this is absorbed by the big-O notation. In the worst case, we pick the largest jug each time, which results in $\sum_{i=2}^{n} i + i - 1 = n^2$ comparisons.

Problem 8-5 Average sorting

> Suppose that, instead of sorting an array, we just require that the elements increase on average. More precisely, we call an n-element array A ***k-sorted*** if, for all $i = 1, 2, \ldots, n - k$, the following holds:
>
> $$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \le \frac{\sum_{j=i+1}^{i+k} A[j]}{k}.$$
>
> **a.** What does it mean for an array to be 1-sorted?
>
> **b.** Give a permutation of the numbers $1, 2, \ldots, 10$ that is 2-sorted, but not sorted.
>
> **c.** Prove that an n-element array is k-sorted if and only if $A[i] \le A[i + k]$ for all $i = 1, 2, \ldots, n - k$.
>
> **d.** Give an algorithm that k-sorts an n-element array in $O(n \lg(n/k))$ time.
>
> We can also show a lower bound on the time to produce a k-sorted array, when k is a constant.

e. Show that we can sort a k-sorted array of length n in $O(n \lg k)$ time. (Hint: Use the solution to Exercise 6.5-9.)

f. Show that when k is a constant, k-sorting an n-element array requires $\Omega(n \lg n)$ time. (Hint: Use the solution to the previous part along with the lower bound on comparison sorts.)

a. Ordinary sorting

b. $2, 1, 4, 3, 6, 5, 8, 7, 10, 9$.

c.

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \le \frac{\sum_{j=i+1}^{i+k} A[j]}{k}$$
$$\sum_{j=i}^{i+k-1} A[j] \le \sum_{j=i+1}^{i+k} A[j]$$
$$A[i] \le A[i+k].$$

d. Shell-Sort, i.e., We split the n-element array into k part. For each part, we use Insertion-Sort (or Quicksort) to sort in $O(n/k \lg(n/k))$ time. Therefore, the total running time is $k \cdot O(n/k \lg(n/k)) = O(n \lg(n/k))$.

e. Using a heap, we can sort a k-sorted array of length n in $O(n \lg k)$ time. (The height of the heap is $\lg k$, the solution to Exercise 6.5-9.)

f. The lower bound of sorting each part is $\Omega(n/k \lg(n/k))$, so the total lower bound is $\Theta(n \lg(n/k))$. Since k is a constant, therefore $\Theta(n \lg(n/k)) = \Omega(n \lg n)$.

Problem 8-6 Lower bound on merging sorted lists

The problem of merging two sorted lists arises frequently. We have seen a procedure for it as the subroutine MERGE in Section 2.3.1. In this problem, we will prove a lower bound of $2n - 1$ on the worst-case number of comparisons required to merge two sorted lists, each containing n items.

First we will show a lower bound of $2n - o(n)$ comparisons by using a decision tree.

a. Given $2n$ numbers, compute the number of possible ways to divide them into two sorted lists, each with n numbers.

b. Using a decision tree and your answer to part (a), show that any algorithm that correctly merges two sorted lists must perform at least $2n - o(n)$ comparisons.

Now we will show a slightly tighter $2n - 1$ bound.

c. Show that if two elements are consecutive in the sorted order and from different lists, then they must be compared.

d. Use your answer to the previous part to show a lower bound of $2n - 1$ comparisons for merging two sorted lists.

a. There are $\binom{2n}{n}$ ways to divide $2n$ numbers into two sorted lists, each with n numbers.

b. Based on Exercise C.1.13,

$$\binom{2n}{n} \le 2^h$$
$$h \ge \lg \frac{(2n)!}{(n!)^2}$$
$$= \lg(2n!) - 2\lg(n!)$$
$$= \Theta(2n \lg 2n) - 2\Theta(n \lg n)$$
$$= \Theta(2n).$$

c. We have to know the order of the two consecutive elements.

d. Let list $A = 1, 3, 5, \ldots, 2n - 1$ and $B = 2, 4, 6, \ldots, 2n$. By part (c), we must compare 1 with 2, 2 with 3, 3 with 4, and so on up until we compare $2n - 1$ with $2n$. This amounts to a total of $2n - 1$ comparisons.

Problem 8-7 The 0-1 sorting lemma and columnsort

A ***compare-exchange*** operation on two array elements $A[i]$ and $A[j]$, where $i < j$, has the form

```
COMPARE-EXCHANGE(A, i, j)
    if A[i] > A[j]
        exchange A[i] with A[j]
```

After the compare-exchange operation, we know that $A[i] \le A[j]$.

An ***oblivious compare-exchange algorithm*** operates solely by a sequence of prespecified compare-exchange operations. The indices of the positions compared in the sequence must be determined in advance, and although they can depend on the number of elements being sorted, they cannot depend on the values being sorted, nor can they depend on the result of any prior compare-exchange operation. For example, here is insertion sort expressed as an oblivious compare-exchange algorithm:

```
INSERTION-SORT(A)
    for j = 2 to A.length
        for i = j - 1 downto 1
            COMPARE-EXCHANGE(A, i, i + 1)
```

The ***0-1 sorting lemma*** provides a powerful way to prove that an oblivious compare-exchange algorithm produces a sorted result. It states that if an oblivious compare-exchange algorithm correctly sorts all input sequences consisting of only 0s and 1s, then it correctly sorts all inputs containing arbitrary values.

You will prove the 0-1 sorting lemma by proving its contrapositive: if an oblivious compare-exchange algorithm fails to sort an input containing arbitrary values, then it fails to sort some 0-1 input. Assume that an oblivious compare-exchange algorithm X fails to correctly sort the array $A[1..n]$. Let $A[p]$ be the smallest value in A that algorithm X puts into the wrong location, and let $A[q]$ be the value that algorithm X moves to the location into which $A[p]$ should have gone. Define an array $B[1..n]$ of 0s and 1s as follows:

$$B[i] = \begin{cases} 0 & \text{if } A[i] \le A[p], \\ 1 & \text{if } A[i] > A[p]. \end{cases}$$

a. Argue that $A[q] > A[p]$, so that $B[p] = 0$ and $B[q] = 1$.

b. To complete the proof of the 0-1 sorting lemma, prove that algorithm X fails to sort array B correctly.

Now you will use the 0-1 sorting lemma to prove that a particular sorting algorithm works correctly. The algorithm, ***columnsort***, works on a rectangular array of n elements. The array has r rows and s columns (so that $n = rs$), subject to three restrictions:

- r must be even,
- s must be a divisor of r, and
- $r \ge 2s^2$.

When columnsort completes, the array is sorted in ***column-major order***: reading down the columns, from left to right, the elements monotonically increase.

Columnsort operates in eight steps, regardless of the value of n. The odd steps are all the same: sort each column individually. Each even step is a fixed permutation. Here are the steps:

1. Sort each column.
2. Transpose the array, but reshape it back to r rows and s columns. In other words, turn the leftmost column into the top r/s rows, in order; turn the next column into the next r/s rows, in order; and so on.
3. Sort each column.
4. Perform the inverse of the permutation performed in step 2.
5. Sort each column.
6. Shift the top half of each column into the bottom half of the same column, and shift the bottom half of each column into the top half of the next column to the right. Leave the top half of the leftmost column empty. Shift the bottom half of the last column into the top half of a new rightmost column, and leave the bottom half of this new column empty.
7. Sort each column.
8. Perform the inverse of the permutation performed in step 6.

Figure 8.5 shows an example of the steps of columnsort with $r = 6$ and $s = 3$. (Even though this example violates the requirement that $r \ge 2s^2$, it happens to work.)

c. Argue that we can treat columnsort as an oblivious compare-exchange algorithm, even if we do not know what sorting method the odd steps use.

Although it might seem hard to believe that columnsort actually sorts, you will use the 0-1 sorting lemma to prove that it does. The 0-1 sorting lemma applies because we can treat columnsort as an oblivious compare-exchange algorithm. A couple of definitions will help you apply the 0-1 sorting lemma. We say that an area of an array is ***clean*** if we know that it contains either all 0s or all 1s. Otherwise, the area might contain mixed 0s and 1s, and it is ***dirty***. From here on, assume that the input array contains only 0s and 1s, and that we can treat it as an array with r rows and s columns.

d. Prove that after steps 1–3, the array consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most s dirty rows between them.

e. Prove that after step 4, the array, read in column - major order, starts with a clean area of 0s, ends with a clean area of 1s, and has a dirty area of at most s^2 elements in the middle.

f. Prove that steps 5–8 produce a fully sorted 0-1 output. Conclude that columnsort correctly sorts all inputs containing arbitrary values.

g. Now suppose that s does not divide r. Prove that after steps 1–3, the array consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most $2s - 1$ dirty rows between them. How large must r be, compared with s, for columnsort to correctly sort when s does not divide r?

h. Suggest a simple change to step 1 that allows us to maintain the requirement that $r \ge 2s^2$ even when s does not divide r, and prove that with your change, columnsort correctly sorts.

(Removed)

9 Medians and Order Statistics

9.1 Minimum and maximum

9.1-1

Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. (Hint: Also find the smallest element.)

We can compare the elements in a tournament fashion - we split them into pairs, compare each pair and then proceed to compare the winners in the same fashion. We need to keep track of each "match" the potential winners have participated in.

We select a winner in $n - 1$ matches. At this point, we know that the second smallest element is one of the lgn elements that lost to the smallest - each of them is smaller than the ones it has been compared to, prior to losing. In another $\lceil \lg n \rceil - 1$ comparisons we can find the smallest element out of those.

9.1-2 ⋆

Prove the lower bound of $\lceil 3n/2 \rceil - 2$ comparisons in the worst case to find both the maximum and minimum of n numbers. (Hint: Consider how many numbers are potentially either the maximum or minimum, and investigate how a comparison affects these counts.)

If n is odd, there are

$$1 + \frac{3(n-3)}{2} + 2 = \frac{3n}{2} - \frac{3}{2}$$
$$= (\left\lfloor \frac{3n}{2} \right\rfloor - \frac{1}{2}) - \frac{3}{2}$$
$$= \left\lfloor \frac{3n}{2} \right\rfloor - 2$$

comparisons.

If n is even, there are

$$1 + \frac{3(n-2)}{2} = \frac{3n}{2} - 2$$
$$= \left\lceil \frac{3n}{2} \right\rceil - 2$$

comparisons.

9.2 Selection in expected linear time

9.2-1

Show that RANDOMIZED-SELECT never makes a recursive call to a 0-length array.

Calling a 0-length array would mean that the second and third arguments are equal. So, if the call is made on line 8, we would need that $p = q - 1$, which means that $q - p + 1 = 0$.

However, i is assumed to be a nonnegative number, and to be executing line 8, we would need that $i < k = q - p + 1 = 0$, a contradiction. The other possibility is that the bad recursive call occurs on line 9. This would mean that $q + 1 = r$. To be executing line 9, we need that $i > k = q - p + 1 = r - p$. This would be a nonsensical original call to the array though because we are asking for the ith element from an array of strictly less size.

9.2-2

Argue that the indicator random variable X_k and the value $T(\max(k-1, n-k))$ are independent.

The probability that X_k is equal to 1 is unchanged when we know the max of $k - 1$ and $n - k$. In other words, $\Pr\{X_k = a \mid \max(k-1, n-k) = m\} = \Pr\{X_k = a\}$ for $a = 0, 1$ and $m = k - 1, n - k$ so X_k and $\max(k-1, n-k)$ are independent.

By C.3-5, so are X_k and $T(\max(k-1, n-k))$.

9.2-3

Write an iterative version of RANDOMIZED-SELECT.

```
PARTITION(A, p, r)
    x = A[r]
    i = p
    for k = p - 1 to r
       if A[k] < x
           i = i + 1
           swap A[i] with A[k]
    i = i + 1
    swap A[i] with A[r]
    return i
```

```
RANDOMIZED-PARTITION(A, p, r)
    x = RANDOM(p - 1, r)
    swap A[x] with A[r]
    return PARTITION(A, p, r)
```

```
RANDOMIZED-SELECT(A, p, r, i)
    while true
        if p == r
            return A[p]
        q = RANDOMIZED-PARTITION(A, p, r)
        k = q - p + 1
        if i == k
            return A[q]
        if i < k
            r = q - 1
        else
            p = q + 1
            i = i - k
```

9.2-4

Suppose we use RANDOMIZED-SELECT to select the minimum element of the array $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

When the partition selected is always the maximum element of the array we get worst-case performance. In the example, the sequence would be $\langle 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 \rangle$.

9.3 Selection in worst-case linear time

9.3-1

In the algorithm SELECT, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7? Argue that SELECT does not run in linear time if groups of 3 are used.

It will still work if they are divided into groups of 7, because we will still know that the median of medians is less than at least 4 elements from half of the $\lceil n/7 \rceil$ groups, so, it is greater than roughly $4n/14$ of the elements.

Similarly, it is less than roughly $4n/14$ of the elements. So, we are never calling it recursively on more than $10n/14$ elements. $T(n) \leq T(n/7) + T(10n/14) + O(n)$. So, we can show by substitution this is linear.

We guess $T(n) < cn$ for $n < k$. Then, for $m \geq k$,

$$\begin{aligned} T(m) &\leq T(m/7) + T(10m/14) + O(m) \\ &\leq cm(1/7 + 10/14) + O(m), \end{aligned}$$

therefore, as long as we have that the constant hidden in the big-Oh notation is less than $c/7$, we have the desired result.

Suppose now that we use groups of size 3 instead. So, For similar reasons, we have that the recurrence we are able to get is $T(n) = T(\lceil n/3 \rceil) + T(4n/6) + O(n) \geq T(n/3) + T(2n/3) + O(n)$ So, we will show it is $\geq cn \lg n$.

$$\begin{aligned} T(m) &\geq c(m/3)\lg(m/3) + c(2m/3)\lg(2m/3) + O(m) \\ &\geq cm \lg m + O(m), \end{aligned}$$

therefore, we have that it grows more quickly than linear.

9.3-2

Analyze SELECT to show that if $n \geq 140$, then at least $\lceil n/4 \rceil$ elements are greater than the median-of-medians x and at least $\lceil n/4 \rceil$ elements are less than x.

$$\begin{aligned} \frac{3n}{10} - 6 &\geq \lceil \frac{n}{4} \rceil \\ \frac{3n}{10} - 6 &\geq \frac{n}{4} + 1 \\ 12n - 240 &\geq 10n + 40 \\ n &\geq 140. \end{aligned}$$

9.3-3

Show how quicksort can be made to run in $O(n \lg n)$ time in the worst case, assuming that all elements are distinct.

We can modify quicksort to run in worst case $n \lg n$ time by choosing our pivot element to be the exact median by using quick select. Then, we are guaranteed that our pivot will be good, and the time taken to find the median is on the same order of the rest of the partitioning.

9.3-4 ⋆

Suppose that an algorithm uses only comparisons to find the ith smallest element in a set of n elements. Show that it can also find the $i-1$ smaller elements and $n-i$ larger elements without performing additional comparisons.

Create a graph with n vertices and draw a directed edge from vertex i to vertex j if the ith and jth elements of the array are compared in the algorithm and we discover that $A[i] \geq A[j]$. Observe that $A[i]$ is one of the $i-1$ smaller elements if there exists a path from x to i in the graph, and $A[i]$ is one of the $n-i$ larger elements if there exists a path from i to x in the graph. Every vertex i must either lie on a path to or from x because otherwise the algorithm can't distinguish between $i \leq x$ and $i \geq x$. Moreover, if a vertex i lies on both a path to x and a path from x, then it must be such that $x \leq A[i] \leq x$, so $x = A[i]$. In this case, we can break ties arbitrarily.

9.3-5

Suppose that you have a "black-box" worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

To use it, just find the median, partition the array based on that median.

- If i is less than half the length of the original array, recurse on the first half.
- If i is half the length of the array, return the element coming from the median finding black box.
- If i is more than half the length of the array, subtract half the length of the array, and then recurse on the second half of the array.

9.3-6

The kth ***quantiles*** of an n-element set are the $k-1$ order statistics that divide the sorted set into k equal-sized sets (to within 1). Give an $O(n \lg k)$-time algorithm to list the kth quantiles of a set.

Pre-calculate the positions of the quantiles in $O(k)$, we use the $O(n)$ select algorithm to find the $\lfloor k/2 \rfloor$th position, after that the elements are divided into two sets by the pivot the $\lfloor k/2 \rfloor$th position, we do it recursively in the two sets to find other positions. Since the maximum depth is $\lceil \lg k \rceil$, the total running time is $O(n \lg k)$.

```
PARTITION(A, p, r)
    x = A[r]
    i = p
    for k = p to r
        if A[k] < x
            i = i + 1
            swap A[i] with A[k]
    i = i + 1
    swap a[i] with a[r]
    return i
```

```
RANDOMIZED-PARTITION(A, p, r)
    x = RANDOM(p, r)
    swap A[x] with A[r]
    return PARTITION(A, p, r)
```

```
RANDOMIZED-SELECT(A, p, r, i)
    while true
        if p == r
            return p, A[p]
        q = RANDOMIZED-PARTITION(A, p, r)
        k = q - p + 1
        if i == k
            return q, A[q]
        if i < k
            r = q
        else
            p = q + 1
            i = i - k
```

```
k-QUANTITLES-SUB(A, p, r, pos, f, e, quantiles)
    if f + 1 > e
        return
    mid = (f + e) / 2
    q, val = RANDOMIZED-SELECT(A, p, r, pos[mid])
    quantiles[mid] = val
    k = q - p + 1
    for i = mid + 1 to e
        pos[i] = pos[i] - k
    k-QUANTILES-SUB(A, q + 1, r, pos, mid + 1, e, quantiles)
```

```
k-QUANTITLES(A, k)
    num = A.size() / k
    mod = A.size() % k
    pos = num[1..k]
    for i = 1 to mod
        pos[i] = pos[i] + 1
    for i = 1 to k
        pos[i] = pos[i] + pos[i - 1]
    quantiles = [1..k]
    k-QUANTITLES-SUB(A, 0, A.length, pos, 0, pos.size(), quantiles)
    return quantiles
```

9.3-7

Describe an $O(n)$-time algorithm that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S.

Find the median in $O(n)$; create a new array, each element is the absolute value of the original value subtract the median; find the kth smallest number in $O(n)$, then the desired values are the elements whose absolute difference with the median is less than or equal to the kth smallest number in the new array.

9.3-8

Let $X[1..n]$ and $Y[1..n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\lg n)$-time algorithm to find the median of all $2n$ elements in arrays X and Y.

Without loss of generality, assume n is a power of 2.

```
MEDIAN(X, Y, n)
    if n == 1
        return min(X[1], Y[1])
    if X[n / 2] < Y[n / 2]
        return MEDIAN(X[n / 2 + 1..n], Y[1..n / 2], n / 2)
    return MEDIAN(X[1..n / 2], Y[n / 2 + 1..n], n / 2)
```

9.3-9

Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. The company wants to connect a spur pipeline from each well directly to the main pipeline along a shortest route (either north or south), as shown in Figure 9.2. Given the x- and y-coordinates of the wells, how should the professor pick the optimal location of the main pipeline, which would be the one that minimizes the total length of the spurs? Show how to determine the optimal location in linear time.

- If n is odd, we pick the y coordinate of the main pipeline to be equal to the median of all the y coordinates of the wells.
- If n is even, we pick the y coordinate of the pipeline to be anything between the y coordinates of the wells with y-coordinates which have order statistics $\lfloor (n+1)/2 \rfloor$ and the $\lceil (n+1)/2 \rceil$. These can all be found in linear time using the algorithm from this section.

Problem 9-1 Largest i numbers in sorted order

Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of n and i.

a. Sort the numbers, and list the i largest.

b. Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.

c. Use an order-statistic algorithm to find the ith largest number, partition around that number, and sort the i largest numbers.

a. The running time of sorting the numbers is $O(n \lg n)$, and the running time of listing the i largest is $O(i)$. Therefore, the total running time is $O(n \lg n + i)$.

b. The running time of building a max-priority queue (using a heap) from the numbers is $O(n)$, and the running time of each call EXTRACT-MAX is $O(\lg n)$. Therefore, the total running time is $O(n + i \lg n)$.

c. The running time of finding and partitioning around the ith largest number is $O(n)$, and the running time of sorting the i largest numbers is $O(i \lg i)$. Therefore, the total running time is $O(n + i \lg i)$.

Problem 9-2 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

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

For example, if the elements are $0.1, 0.35, 0.05, 0.1, 0.15, 0.05, 0.2$ and each element equals its weight (that is, $w_i = x_i$ for $i = 1, 2, \ldots, 7$), then the median is 0.1, but the weighted median is 0.2.

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

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

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.

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.

d. Argue that the weighted median is a best solution for the 1-dimensional postoffice location problem, in which points are simply real numbers and the distance between points a and b is $d(a, b) = |a - b|$.

e. Find the best solution for the 2-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 $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

a. Let m_k be the number of x_i smaller than x_k. When weights of $1/n$ are assigned to each x_i, we have $\sum_{x_i < x_k} w_i = m_k/n$ and $\sum_{x_i > x_k} w_i = (n - m_k - 1)/2$. The only value of m_k which makes these sums $< 1/2$ and $\leq 1/2$ respectively is when $\lceil n/2 \rceil - 1$, and this value x must be the median since it has equal numbers of x_i's which are larger and smaller than it.

b. First use mergesort to sort the x_i's by value in $O(n \lg n)$ time. Let S_i be the sum of the weights of the first i elements of this sorted array and note that it is $O(1)$ to update S_i. Compute $S_1, S_2, \ldots$ until you reach k such that $S_{k-1} < 1/2$ and $S_k \geq 1/2$. The weighted median is x_k.

c. We modify SELECT to do this in linear time. Let x be the median of medians. Compute $\sum_{x_i<x} w_i$ and $\sum_{x_i>x} w_i$ and check if either of these is larger than $1/2$. If not, stop. If so, recurse on the collection of smaller or larger elements known to contain the weighted median. This doesn't change the runtime, so it is $\Theta(n)$.

d. Let p be the minimizer, and suppose that p is not the weighted median. Let ϵ be small enough such that $\epsilon < \min_i(|p - p_i|)$, where we don't include k if $p = p_k$. If p_m is the weighted median and $p < p_m$, choose $\epsilon > 0$. Otherwise choose $\epsilon < 0$. Thus, we have

$$\sum_{i=1}^{n} w_i d(p+\epsilon, p_i) = \sum_{i=1}^{n} w_i d(p, p_i) + \epsilon\left(\sum_{p_i<p} w_i - \sum_{p_i>p} w_i\right) < \sum_{i=1}^{n} w_i d(p, p_i),$$

the difference in sums will take the opposite sign of epsilon.

e. Observe that

$$\sum_{i=1}^{n} w_i d(p, p_i) = \sum_{i=1}^{n} w_i |p_x - (p_i)_x| + \sum_{i=1}^{n} w_i |p_y - (p_i)_y|.$$

It will suffice to minimize each sum separately, which we can do since we choose p_x and p_y individually. By part (e), we simply take $p = (p_x, p_y)$ to be such that p_x is the weighted median of the x-coordinates of the p_i's and py is the weighted medain of the y-coordiantes of the p_i's.

Problem 9-3 Small order statistics

> We showed that the worst-case number $T(n)$ of comparisons used by SELECT to select the ith order statistic from n numbers satisfies $T(n) = \Theta(n)$, but the constant hidden by the Θ-notation is rather large. When i is small relative to n, we can implement a different procedure that uses SELECT as a subroutine but makes fewer comparisons in the worst case.
>
> **a.** Describe an algorithm that uses $U_i(n)$ comparisons to find the ith smallest of n elements, where
>
> $$U_i(n) = \begin{cases} T(n) & \text{if } i \geq n/2, \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) & \text{otherwise.} \end{cases}$$
>
> (Hint: Begin with $\lfloor n/2 \rfloor$ disjoint pairwise comparisons, and recurse on the set containing the smaller element from each pair.)
>
> **b.** Show that, if $i < n/2$, then $U_i(n) = n + O(T(2i)\lg(n/i))$.
>
> **c.** Show that if i is a constant less than $n/2$, then $U_i(n) = n + O(\lg n)$.
>
> **d.** Show that if $i = n/k$ for $k \geq 2$, then $U_i(n) = n + O(T(2n/k)\lg k)$.

(Removed)

Problem 9-4 Alternative analysis of randomized selection

> In this problem, we use indicator random variables to analyze the RANDOMIZED-SELECT procedure in a manner akin to our analysis of RANDOMIZED-QUICKSORT in Section 7.4.2.
>
> As in the quicksort analysis, we assume that all elements are distinct, and we rename the elements of the input array A as $z_1, z_2, \ldots, z_n$, where z_i is the ith smallest element. Thus, the call RANDOMIZED-SELECT$(A, 1, n, k)$ returns z_k.
>
> For $1 \leq i < j \leq n$, let
>
> $$X_{ijk} = I\{z_i \text{ is compared with } z_j \text{ sometime during the execution of the algorithm to find } z_k\}.$$
>
> **a.** Give an exact expression for $E[X_{ijk}]$. (Hint: Your expression may have different values, depending on the values of i, j, and k.)
>
> **b.** Let X_k denote the total number of comparisons between elements of array A when finding z_k. Show that
>
> $$E[X_k] \leq 2\left(\sum_{i=1}^{k}\sum_{j=k}^{n}\frac{1}{j-i+1} + \sum_{j=k+1}^{n}\frac{j-k-1}{j-k+1} + \sum_{i=1}^{k-2}\frac{k-i-1}{k-i+1}\right).$$
>
> **c.** Show that $E[X_k] \leq 4n$.
>
> **d.** Conclude that, assuming all elements of array A are distinct, RANDOMIZED-SELECT runs in expected time $O(n)$.

(Removed)