Homework 6
Chapter 11-13: Sorting

1 Part I: Sort Traces

1. Trace the sorts of the following dataset by hand using the six sorts listed below. Show the array each time it is altered.

   Dataset:

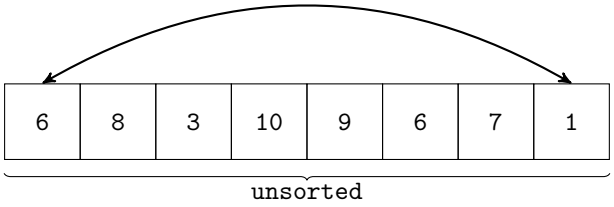
6	8	3	10	9	6	7	1

   Note: I skip steps in the following algorithms when no changes are made to the array

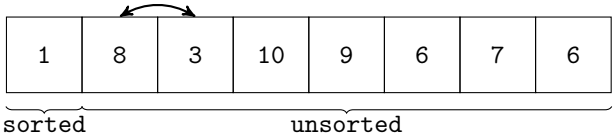
   Sorts to trace:

   (a) Selection sort

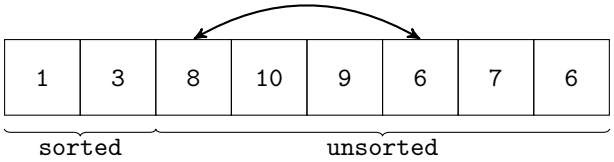
   i.

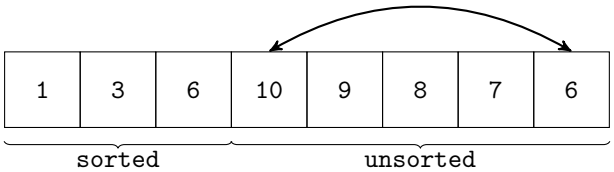
   ii.

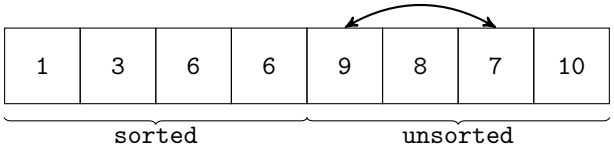
   iii.

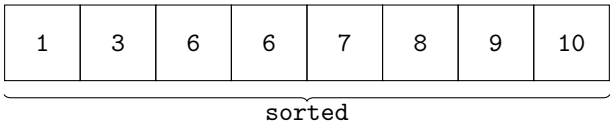
   iv.

   v.

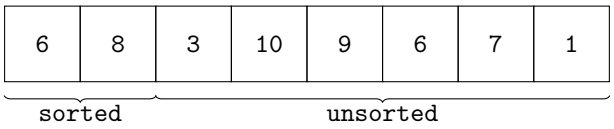
   

vi.

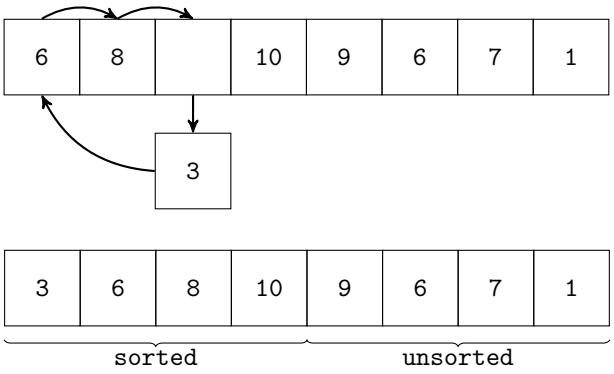


(b) Insertion sort

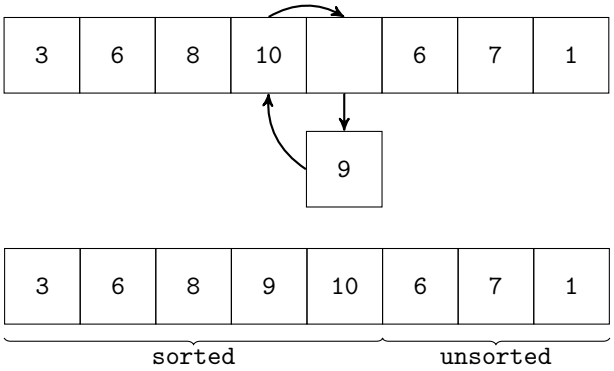
i.



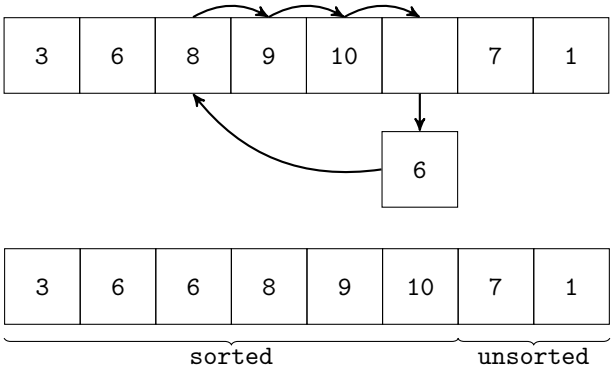
ii.



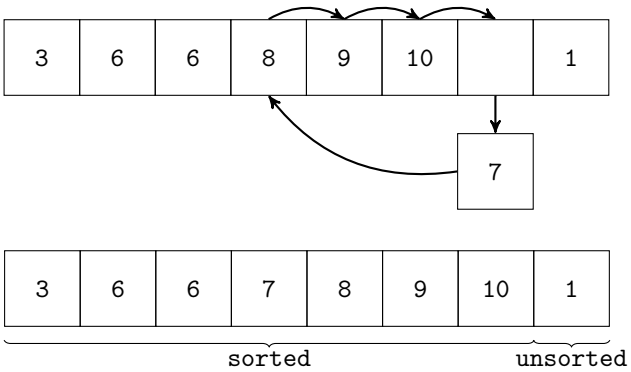
iii.



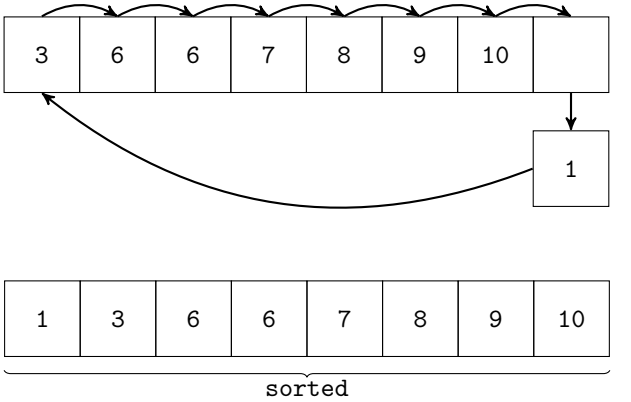
iv.



v.



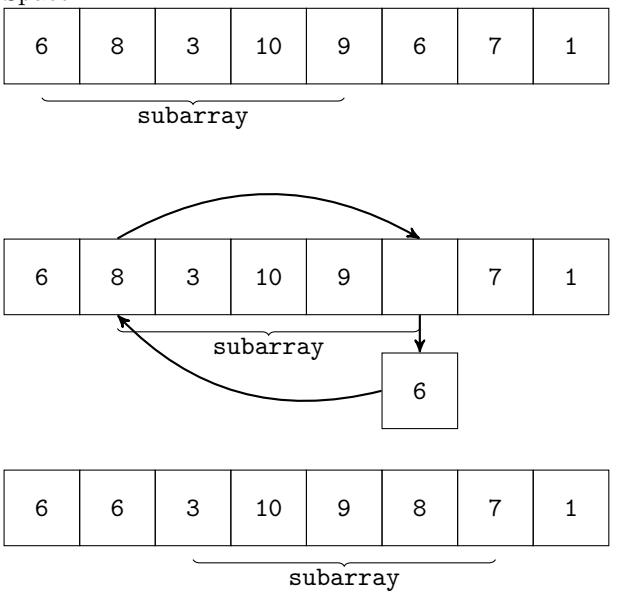
vi.

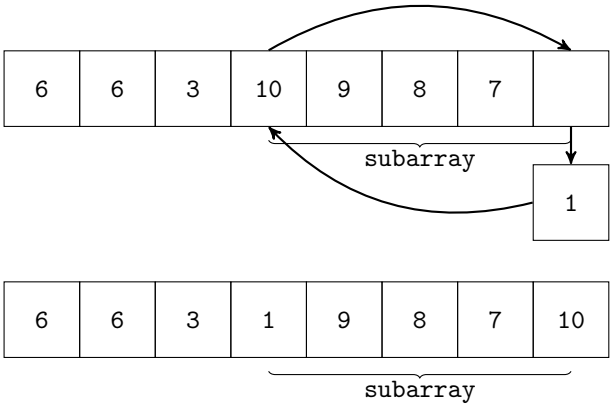


(c) Shell sort

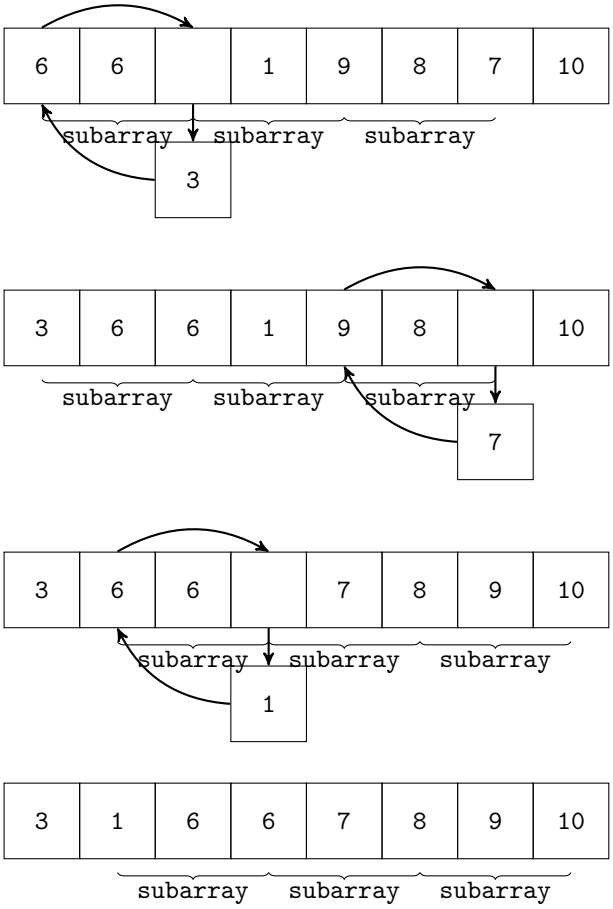
- In addition to showing the contents of the array, also list what the gap is for that part of the trace.

i. Space = 4

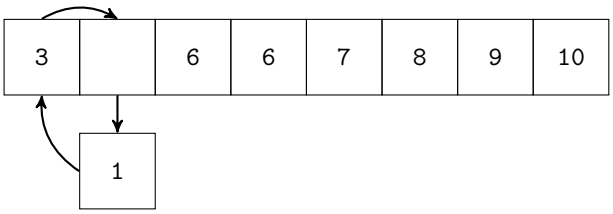




ii. Space = 2



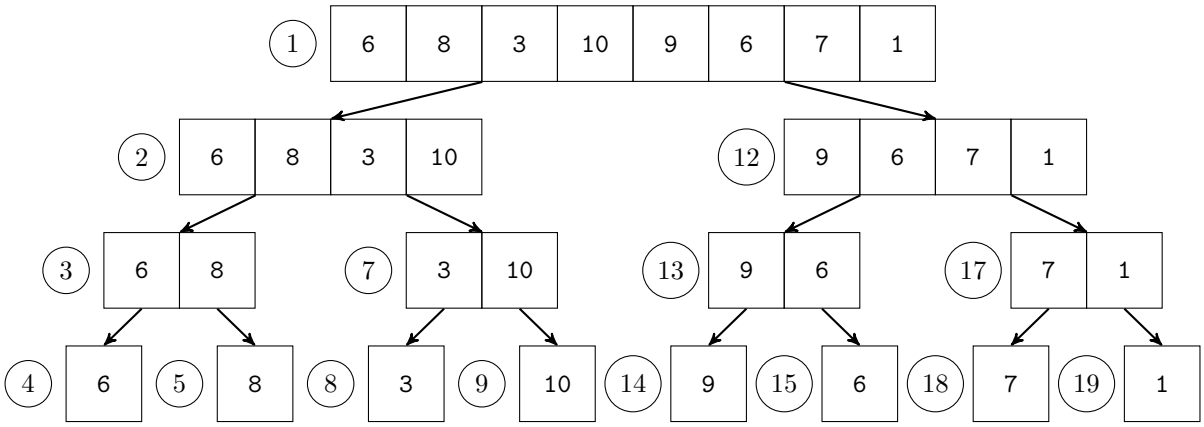
iii. Space = 1



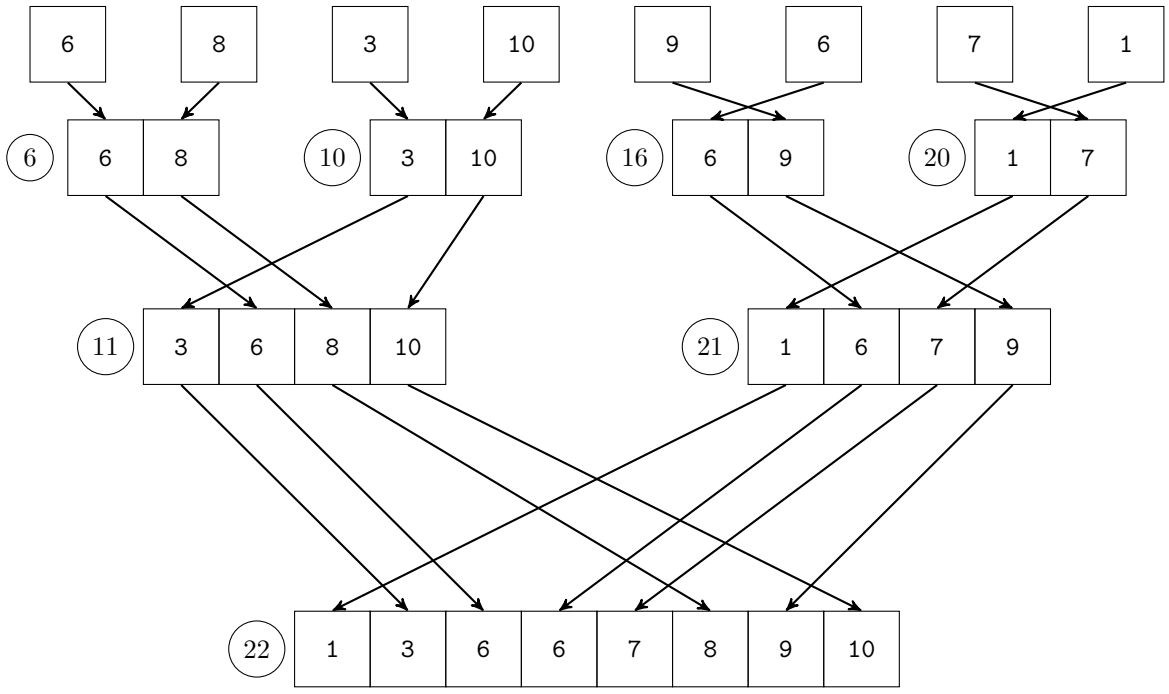
1	3	6	6	7	8	9	10

(d) Merge sort

i. Recursively split the array in half until we have two subarrays where they are both length = 1



ii. sort and merge



(e) Quick sort

- Use the code given in Section 18. I recommend writing out the variables to help with your trace (e.g., pivotIndex, pivot, indexFromLeft, etc.).
- You only need to show the results after the first partition step. In other words, show how the array is changed after the first call to the partition method (displayed in Section 17).
- Also submit what the parameters will be for the next two recursive calls to quickSort

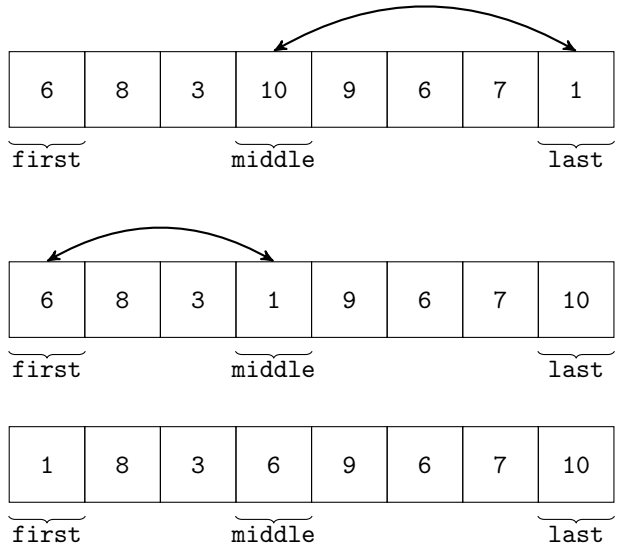
i. Dataset:

6	8	3	10	9	6	7	1

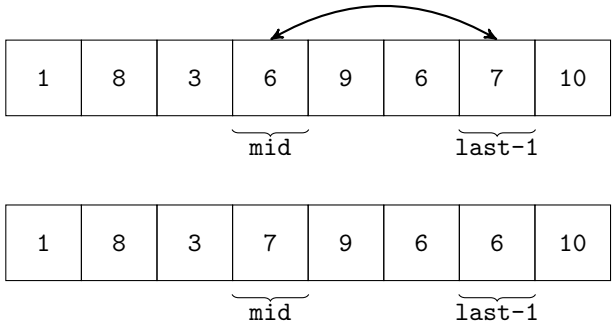
ii. `quickSort(array,0,array.length - 1)`

A. `pivotIndex = partition(array,0,7)`

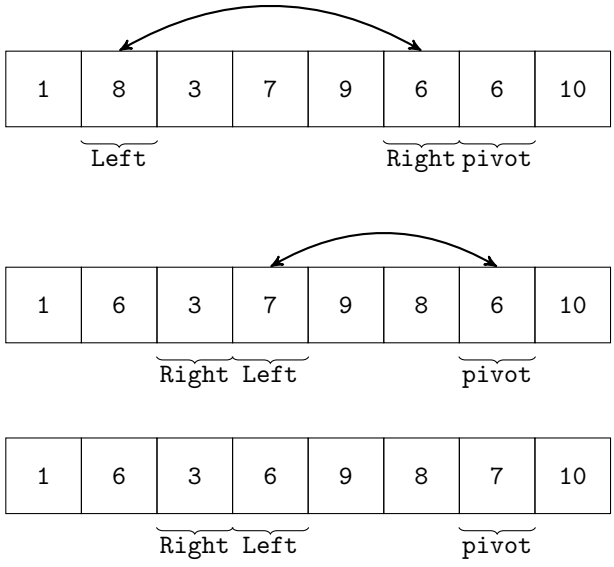
- `sortFirstMiddle(array,0,3,7)`



- `swap(array,mid,last-1)`



- `| small | pivot | large |`



`pivotIndex = 3`

B. `quickSort(array,0,3)`

C. `quickSort(array,4,7)`

(f) Radix sort

- Show the contents of buckets and the array after each pass.

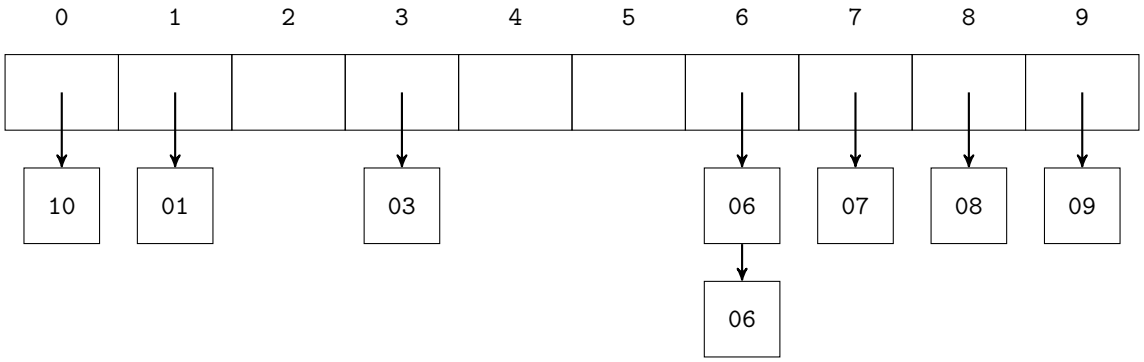
i. Dataset:

6	8	3	10	9	6	7	1

ii. Add padding

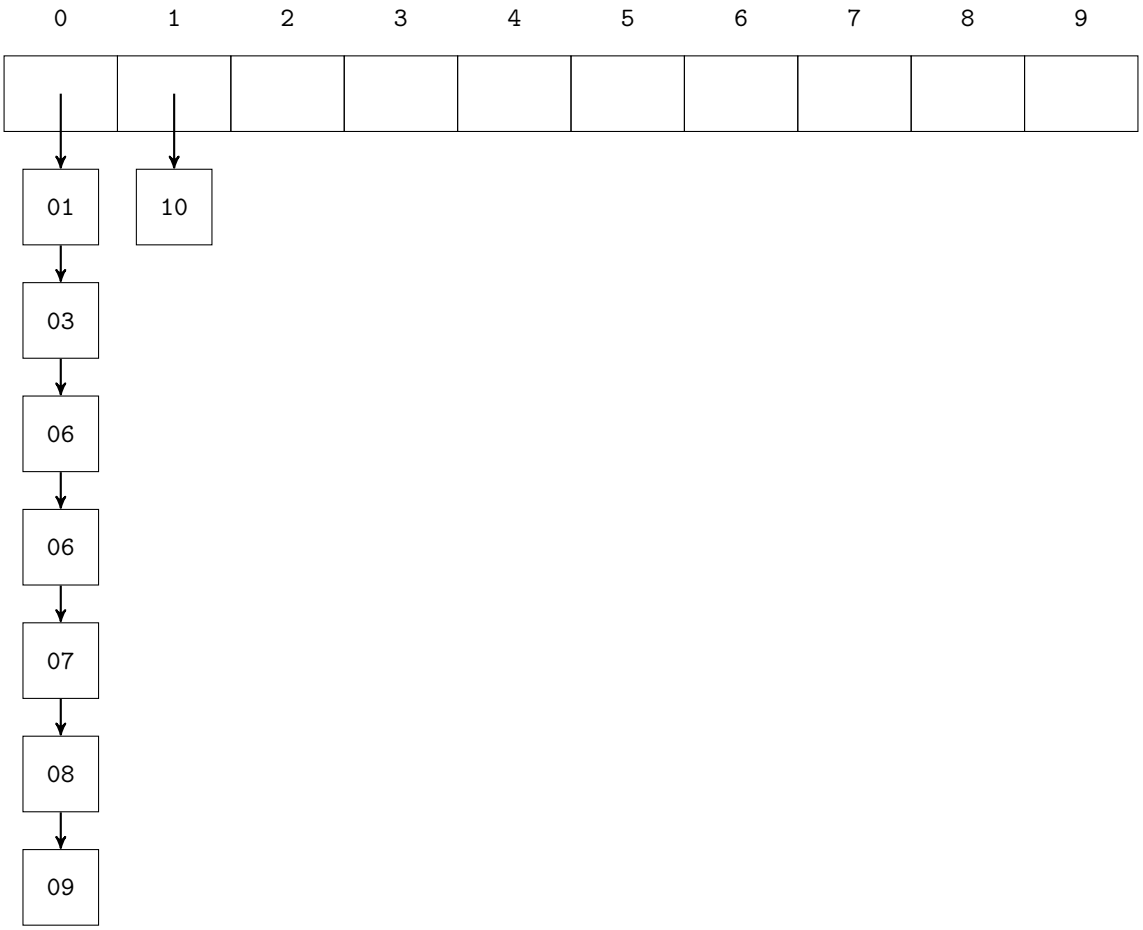
06	08	03	10	09	06	07	01

Distribute integers into buckets according to the rightmost digit



iii. Distribute integers into buckets according to the leftmost digit

10	01	03	06	06	07	08	09



iv. Sorted Dataset

01	03	06	06	07	08	09	10

v. Remove padding

1	3	6	6	7	8	9	10

1.1 Extra Credit

EC. Trace a radix sort on the following dataset. Specify what is required in terms of padding along with the trace of the sort. Words should be sorted into alphabetic, dictionary order.

Dataset:

cat	bird	dog	fly	bug	pug	man	fish

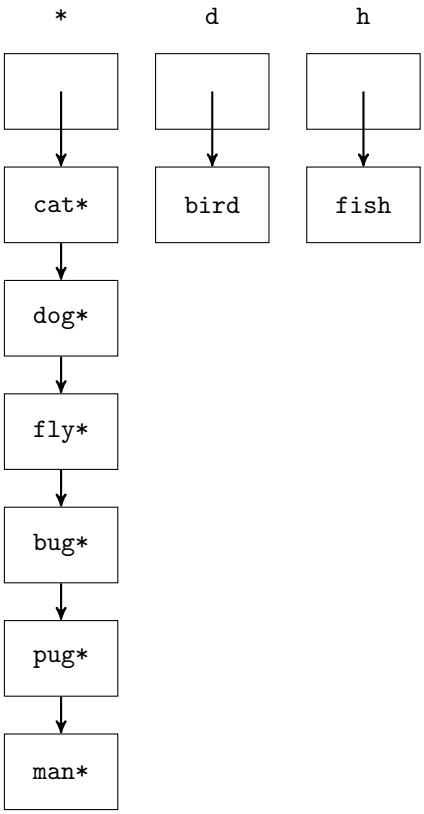
(a) Add some arbitrary non used character as padding. In this case I use '*' as the character for padding.

cat*	bird	dog*	fly*	bug*	pug*	man*	fish

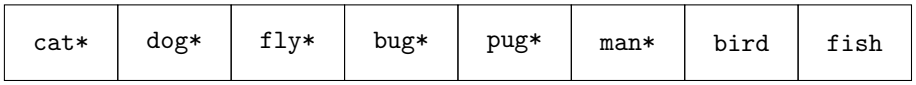
Create an array of length 27, 1 element for each letter in the lowercase alphabet plus 1 for the asterisk. Make

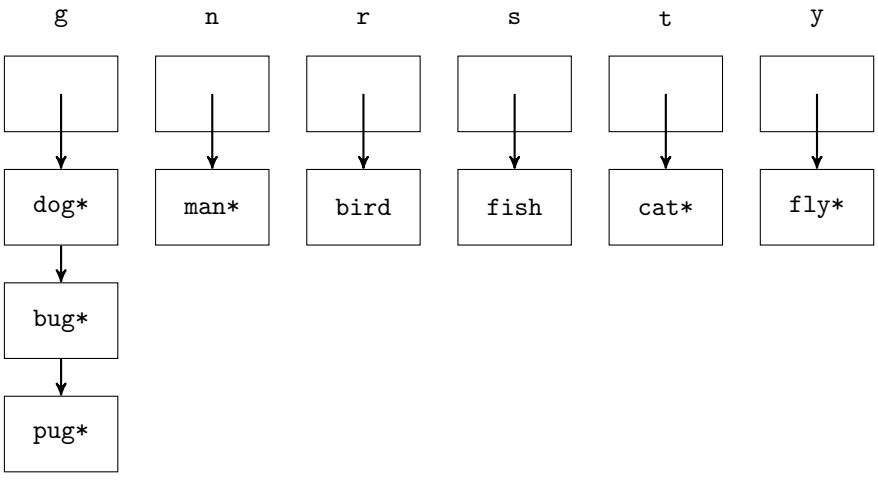
the asterisk the 0^{th} element. The 0^{th} element represents '*', the 1^{st} element represents 'a', the 2^{nd} element represents 'b',..., the 25^{th} element represents 'y', the 26^{th} element represents 'z'.

Distribute words into buckets according to the rightmost character.

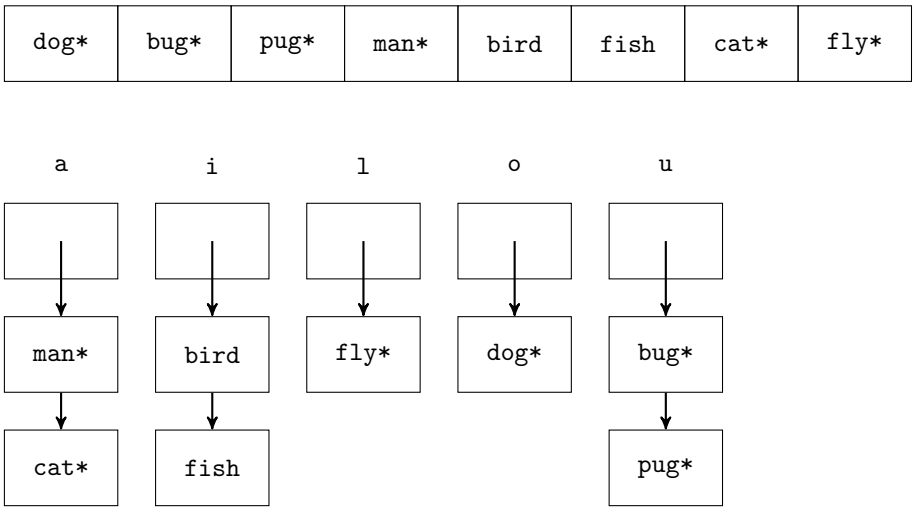


(b) Distribute the words into buckets according to the 2nd rightmost character

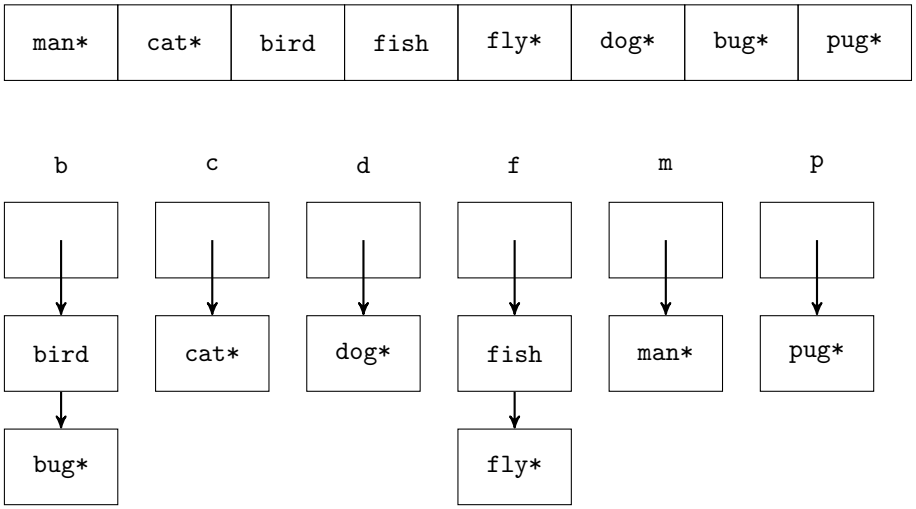




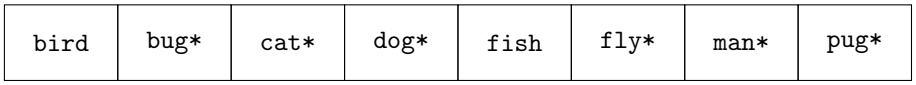
(c) Distribute the words into buckets according to the 3rd rightmost character



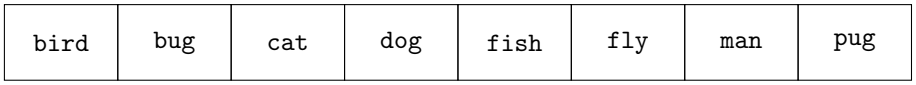
(d) Distribute the words into buckets according to the leftmost character



(e) Sorted Dataset



(f) Remove padding



2 Part II: Sorting Methods and Sorted Lists

11.11 Devise an algorithm that detects whether a given array is sorted into ascending order. Write a Java method that implements your algorithm. You can use your method to test whether a sort method has executed correctly.

- Write a complete method with this header: `public boolean isSorted(Comparable[] array)`

Refer to SortArray.java

11.15 Suppose you want to find the largest element in an unsorted array of n elements. Algorithm A searches the entire array sequentially and records the largest element seen so far. Algorithm B sorts the array into descending order and then reports the first element as the largest. Compare the time efficiency of the two approaches.

In Algorithm A, we have to sequentially iterate through all n elements. The number of operations required for each element is some constant, c. That means there are $c \cdot n$ operations. Therefore the time complexity of Algorithm A is $O(n)$.

In Algorithm B, the fastest sorting algorithm that we learned so far is Radix sort. Radix sort's best, average, and worst case time complexity is $O(n)$. Assuming that the data in the array is appropriate for Radix sort, sorting the data in descending order will have a time complexity of $O(n)$. Reporting the first element after the sort only takes a constant number of operations, so the time complexity will still be $O(n)$.

If Radix sort is not appropriate, the next fastest sorting algorithms are Quick sort and Merge sort. Quick sort's best and average case time complexity is $O(nlogn)$, and its worst case time complexity is $O(n^2)$. However the worst case is rare to occur. Merge sort's best, average, and worst case time complexity is $O(n \cdot \log n)$. In practice, Quick sort can be faster than Merge sort. Assume that we use either Quick sort or Merge sort, and assume that Quick sort only arrives at its best and average cases. The time complexity for sorting the array in descending order will be $O(n \cdot \log n)$. Again, reporting the first element after the sort only takes a constant number of operations, so the time complexity will still be $O(n \cdot \log n)$.

$O(n) < O(n \cdot \log n)$. It would seem that Algorithm A is the better choice if we only care about the largest element in the array. If we are going to search the array for more than just the largest element, it is best to use Algorithm B, as we only need to sort the data once, while the data remains unchanged.

13.2 As specified in this chapter, the sorted list can contain duplicate entries. Specify a sorted list of unique items. For example, `add` could return `true` if it added an entry to the list but return `false` if the entry is in the list already.

- Write a complete method with this header: `public boolean add(T newEntry)`. This method would go inside a `SortedList` class and replace the existing `add` method. This method differs from the current `add` method in that it will not allow the user to add duplicate items. Note: you can invoke any methods from the `SortedListInterface` class. Also, it's possible you would want to keep the existing implementation of `add` as a private method in your class to help with the current implementation. If you want to do this, you can just invoke `addPrivate` in your solution.

Refer to SortedLinkedList.java

13.12 Write a linked implementation of the sorted list method `contains`. Your search of the chain should end when it either locates the desired entry or passes the point at which the entry should have occurred.

- Write a complete method (`public boolean contains(T anEntry)`) that would go inside the `SortedList` class. For full credit, directly access the linked nodes (rather than just invoking existing methods). You can assume that `T` is `Comparable`.

Refer to SortedLinkedList.java

2.1 Extra Credit

13.14c Segment 12.2 of Chapter 12 described how to merge two sorted arrays into one sorted array. Add an operation to the ADT sorted list that merges two sorted lists. Implement the merge in three ways, as follows: Assume a linked implementation.

- Write an implementation-side method: `public void merge(SortedListInterface<T> sList)`
- For full credit, write an efficient implementation that directly access the underlying linked nodes. Writing a method that repeatedly invokes the add method will receive 0 points.
- Note that you can only invoke methods from `SortedListInterface` on the `sList` object
- Be sure to consider all cases for both lists!

Refer to SortedLinkedList.java