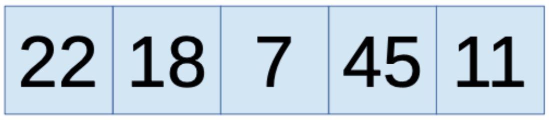
Exercise 13
Locality of Reference

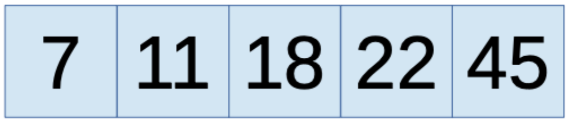
Arrays tend to exhibit good locality, especially if you access elements sequentially or access nearby elements one after another (i.e., nearby elements in the array accessed nearby in time). Linked structures like linked lists may not exhibit as much locality. Moving from one node to the next could involve moving to a completely different location in memory. This could hurt the performance of various types of cache (including the TLB) and could really hurt the performance of virtual memory if a program doesn't all fit in main memory.

For this exercise, we're going to compare array and linked list implementations of quicksort. This is a good algorithm to consider, since its basic operation, partitioning a sequence of values based on a chosen pivot, mostly accesses elements in order.

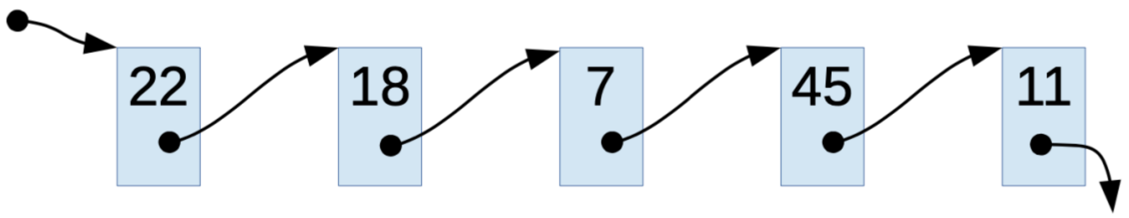
For an array implementation, as the sort moves values around in the array, we should still be accessing memory in order. If the list starts out of order:



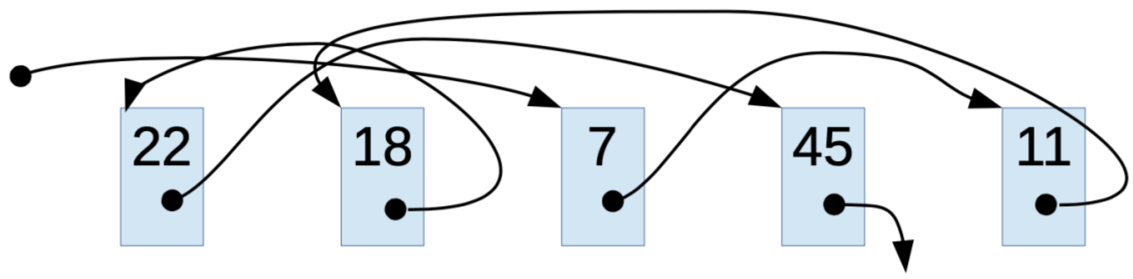
as the sort rearranges items, it moves them around in memory, so items that are nearby in our list are always stored in nearby memory locations:



This isn't how a linked list works. Even if the nodes of a list start out ordered in memory, with nearby nodes in the list order allocated in nearby memory locations:



as the list is sorted, the linking structure will change to rearrange values, but nodes aren't actually moved in memory, so accessing nearby items in the list may not exhibit much locality.



Let's see how this affects performance. I've written an array-based quicksort, **array.c**. It takes a command-line argument, n, giving the number of items to sort. Then, it generates a random list of n items and sorts them recursively with quicksort, always using the first item in a list as the pivot. After sorting, it prints out the sorted list of values (so you can verify that the sort worked).

```
$ ./array 10
424238335
596516649
719885386
846930886
1189641421
1649760492
1681692777
1714636915
1804289383
1957747793
```

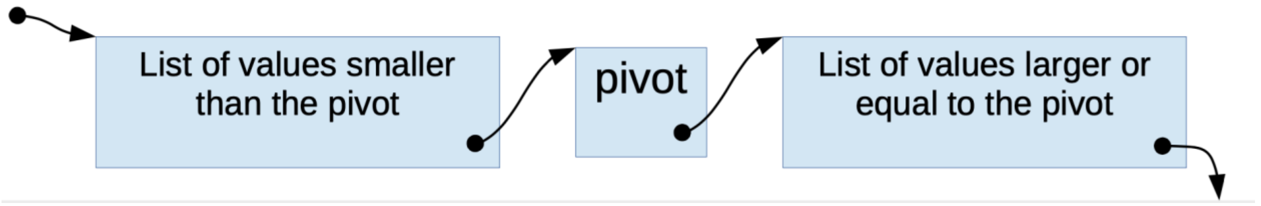
To measure performance on a larger list, we can run the program using the time command, and redirect output to /dev/null (so printing the output doesn't dominate the runtime). Here's what I get when I sort 10,000,000 values on one of the machines at remote.eos.ncsu.edu:

```
$ time ./array 10000000 >/dev/null

real   0m3.762s
user   0m3.695s
sys    0m0.059s
```

Let's compare this time to sorting a linked list. I've written a partial implementation of a linked list quicksort. It creates a list containing the same sequence of values as the array program, calls a sort() function to recursively quicksort the list, then prints out the resulting list. You get to fill in the quicksort implementation. Once you do, you should be able to run it just like the array program to see how it compares to quicksorting an array. (BTW, it took my solution about 17 seconds to quicksort a linked list of 10,000,000 random values on the same system I used above. Telling you this kind of gives away the result of this test, but it will help you to know if you've implemented the quicksort properly).

Just like the array program, your quicksort will partition the list using the first value on the list as the pivot. You'll make two temporary lists, putting all nodes with values smaller than the pivot on one list (let's call it lesser) and all nodes with values greater than or equal to the pivot on another list (let's call it greater). Then, you'll recursively sort the two smaller lists and then combine them into one big, sorted list by changing a few pointers. The head of the combined list will be the first node on the lesser list. The last node in the lesser list will need to point to the pivot as its next node, and the next pointer out of the pivot will need to point to the first node on the greater list. Keep in mind, either the lesser or the greater lists could be empty (say, if there aren't any values less than the pivot). Treat empty and single-node lists as special cases (that don't require any sorting), so there will always be a pivot node.


List of values smaller than the pivot
pivot
List of values larger or equal to the pivot

To make it easy to combine lists like this, we'll represent a list as a small struct, with a pointer to the head node in the list and a different pointer for the tail (the last node of the list). The partial implementation uses a struct that looks like the following.

```
// Representation for a list, with a head and a tail pointer.
typedef struct {
  // Head pointer for the list.
  Node *head;

  // Pointer to the last node in the list, or null if the list
  // is empty.
  Node *tail;
} List;
```

Keeping up with the last node on the list makes it possible to combine sorted lists with the pivot in constant time, without having to go find the last node every time. Doing this cut my sorting time for linked list about in half (compared to just keeping up with a head pointer and having to traverse the list to find its last node).

Optionally, you can represent your list like the following, with the tail pointer stored as the address of the null pointe at the end of the list (even if that's the head pointer). You'll need to modify the list-creation code if you represent your list this way (you should still get the same randomly-generated list, but you'll need to make some small changes to the list-building code in main), but, overall, representing the list like this saved me about a dozen lines of code since I didn't have to give as much special treatment to empty lists.

```
// Representation for a list, with a head and a tail pointer.
typedef struct {
  // Head pointer for the list.
  Node *head;

  // Pointer to the null pointer at the end of the list.
  Node **tail;
} List;
```

Once you have your quicksort working, it should print a sorted list, just like the array-based solution. In fact, asymptotically (big-O notation), your solution should be just as fast as the array-based one. However, with the sequence of values represented as a linked list, the locality-of-reference won't be as good after or during the sort. This (and a few other things) should result in a much higher runtime. Give it a try sorting 10,000,000 values and sending the output to /dev/null, like we did above with the array-based implementation.

When you're done, submit the source code for your completed **linked.c** program to the Gradescope assignment named EX13.