when turns are sharp, i don't cut corners questing for my pivot
i'm a coding shark. i have honor. i'll code it and i'll give it

i sort em sorry small fry to one side - my search for bigass tuna simplified

Pivoting

Hooray! In this quest you get to implement quicksort all by yourself. All 10 lines of it, I think. This may just be your quest with the maximum reward per byte ratio. Actually, no. I forgot that time we danced in Hanoi.

Anyway, before you get to do this quest, you have to pass the free entry challenge.

Entry Challenge

Implement the following global scope in-place sort method in a file called `Entry_Pass.cpp`

```
void my_questing_sort_in_place(vector<int> &elems);
```

On return from this method, `elems` must be sorted in place in non-descending order.[1]

There are two ways to get through this entrance test.

One is to make your method secretly invoke the standard library's `sort()` on the vector. I'm not gonna check. But the coding shark don't do it that way - even to just peek and see what's ahead.

In fact, if you have never implemented a sorting algorithm before, I would say that you are among an EXCEEDINGLY LUCKY FEW. Don't squander away this incredible opportunity to appreciate some of the nuances of sorting. Do it yourself first without looking up any references. This is a fantastic way to get to appreciate the magnitude of work that has gone into crafting about 10 lines of code that are *as simple as possible, but no simpler.*

BTW, in-place means you don't get to allocate more space on the heap. You are allowed to recurse if you pass the elements by reference, but you don't have to.

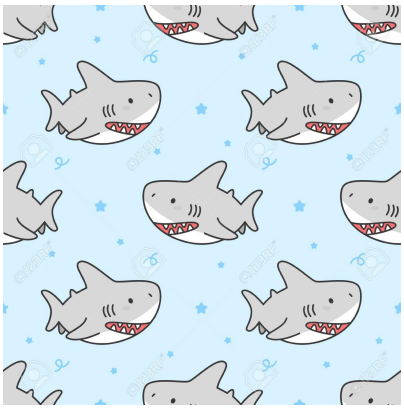
Minis

Ok. Now comes the exciting part. But before you start, here is a helpful general note about this quest: If any of your methods needs a vector to be passed in as a parameter by reference, I'll

[1] That means there should be no pair of elements in the vector that are out of order. If you visually imagine your array spanning cells from left to right, then you'll call out all pairs in which the right is smaller than the left as an out of order pair. These also go by the simpler name, *inversion*.

make sure that the vector is not empty. This should save you from having to build in redundant checks in your code for things that can be easily controlled from outside.

Figure 1 shows you a picture of a template class[2] called `Pivoting`. It is a wrapper for a collection of algorithms you could conceivably implement, all of which share the common theme of *pivoting*. This quest concerns the implementation of three such public methods:



- `find_kth_least()`
- `find_median()`
- `do_qsort()`

and some private helpers.

```
template <typename T>
class Pivoting {
private:
    static size_t _partition(vector<T> &elems, size_t lo, size_t hi);
    static void _do_qsort(vector<T> &elems, size_t lo, size_t hi);
    static T _find_kth_least_elem(vector<T> &elems, size_t lo, size_t hi, size_t k);

public:
    static T find_median(vector<T> &elems);
    static T find_kth_least_elem(vector<T> &elems, size_t k);
    static void do_qsort(vector<T> &elems);

    friend class Tests;
};
```

Figure 1. The Pivoting Class

Fun facts

- No news may be good news. But it most certainly is no good news. If you don't implement any of the methods and simply submit an empty template class called `Pivoting`, I'm not gonna complain. But trophies will automatically appear for things done right, and the password will automagically appear after a certain number of trophies have been won.
- In what order should you implement the methods? Should you implement one before another? In some cases the answer is obvious. In others, not so much! It's like a metaquest. Go figure.

[2] How might you generalize this implementation even further? I mean, it's already a template class, allowing you to operate on vectors of any type that supports the less than comparator.

Rewarding facts

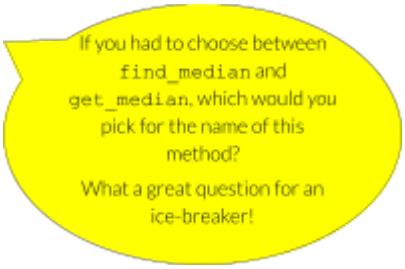
_do_qsort(vector<T> &elems, size_t lo, size_t hi)

You're being given a vector of things that guarantee the less than operator can be used to compare them.

Not only should this method quicksort the segment of the vector spanning `elems[lo] ... elems[hi]`, but it should do it using the exact same strategy (including partitioning) that the reference code does.[3] Or you'll ouch out with not much more else to say.

After you partition the array to find the partition index k, you may find it helpful to sort the sub-arrays `[lo ... k]` and `[k+1 ... hi]` rather than some other division. You can usually afford one quick check at the start of the method and immediately return if possible. This kind of early-checking has saved me many convoluted downstream checks.

find_median(vector<T> &elems)



Return the median element from `elems`. That'd be the middle element if `elems` was sorted.

Except you don't have to sort `elems`. How much faster can this be than if you had to sort it?

What if there's no one middle element (even-sized vectors)? In that case, rather than mess around with arithmetic requirements on the template type, simply return the second of the two middle items.[4]

_find_kth_least_elem(vector<T> &elems, size_t lo, size_t hi, size_t k)

This should return the element that would be at index `k` in `elems`, *if `elems` was sorted*. But you don't have to sort it. Sorting is how the non-programming types do it if they don't know better.

Leverage these two important insights:

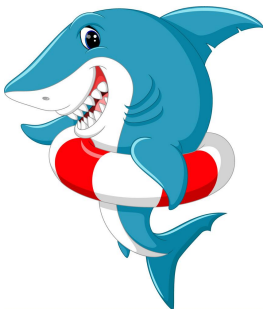
1. Your partition[5] algorithm will divide up the vector around a *pivot* value into 2 pieces.
2. Your kth least element is *certainly*[6] located in at most one of them (you can afford to completely ignore the other).

Remember that your kth least element will keep jumping around in your array as you search for it.[7] That's perfectly fine. You only need to return it after it has settled down. When it has settled it will be in index `k` (but not *only* then).

---

[3] There's only a few different ways to do it right. So try 'em all and one might click.
[4] What would it take to make the method return the correct arithmetic median of the vector?
[5] Partitioning is like tiling. The partitions must not overlap, and together they must cover the whole area.
[6] *certainly* or *only*?
[7] After each jump, it will usually only have about enough energy left for a jump half as long as the previous one. Why?

You can recursively invoke yourself after adjusting for just the one partition in which your target can be found. By ignoring half the search space (on average) at every recursion, you can find the target in at worst linear time.



There are MANY corner cases here, and although this method is only a few lines long, it could take you many hours to get right if you go down a wrong path. I wish you a struggle that will be worth it for years to come.

find_kth_least_elem(vector<T> &elems, size_t k)

This is the public facing version.

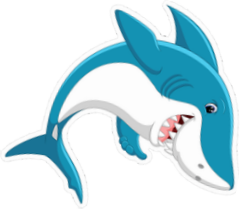
Just so we're all on the same page, let's define k to be the array index, rather than conventional notions of 1st, 2nd, etc.

You should check if `k` is valid and invoke your private helper with the appropriate parameters. If `k` is not valid, you must simply return a default `T` object. Here is where you should do this check. Not in the private helper. Why? Also, note that it returns a default T for invalid k, rather than throw an exception. You should be able to change this in a snap if an application demands it.

size_t _partition(vector<T> &elems, size_t lo, size_t hi)

This method should choose an appropriate pivot (described below) and partition the elements in `elems` into two disjoint chunks such that the first chunk consists of elements *no bigger than* the pivot and the right chunk consists of elements *no smaller than* the pivot. It should then return the index of the first element of the second partition.[8]

This is the most important method to get right for this quest. Obviously there are many different ways of partitioning an array. And you should try at least a few different variants *on your own* before you read on (and before you hit a reference). But to pass this miniquest, you have to partition it using the strategy described below. Both your pivot as well as your final permutation need to match up with the reference's.



[8] Note that it would have crossed-over by now.

Partitioning

We want to divide up `_elems` into two disjoint parts. The first consists of all elements no bigger than a value we call the pivot, and the second consists of all elements at least as big as the pivot. If there's only one element and that's the pivot, it should obviously only go to one part.

You then return the array index of the first element of the second part. See Figure 2 for an example.

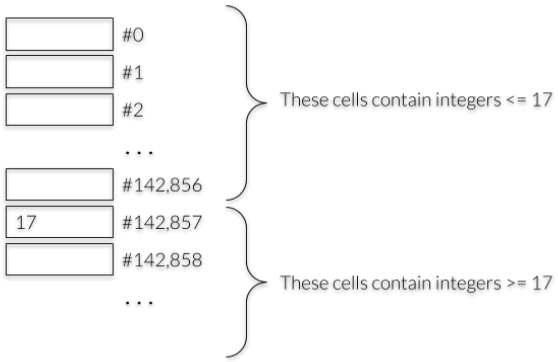


Figure 2. Suppose your `partition()` returned 142,857 and that the value at that location is 17. You can then make the above inferences.

Overview of the winning strategy[9]

- Select your pivot index = lo + (hi-lo)/2. Your pivot is the element at this index[10]
- Now set up two runners (indices) at the two ends of the array. They're going to race towards each other as fast as they can. Except:
    - The left runner can't cross cells > pivot
    - The right runner can't cross cells < pivot
    - When they both get stuck, check to see if they have met each other.
    - If they have, then you can consider the location of the right runner your partition point (How can you say that for sure?)
    - If they haven't met each other, then we know for sure that either (1) the elements that they are stopped at are both equal to the pivot or (2) the left element is greater than the right element. In either case, it is safe to swap them. So swap, and restart runners at the next locations.[11]

[9] You must have convinced yourself by now that there are several correct strategies, but only one winning strategy. If not, this is another chance to try and find some other correct strategies.
[10] Hmm… Why the added complexity? When might this be different from the average of `lo` and `hi`?
[11] I wish you many hours of wonderful exploration as you look for cool reasons, like for choosing to allow redundant swaps (pivot with pivot) instead of skipping them conditionally (why?).

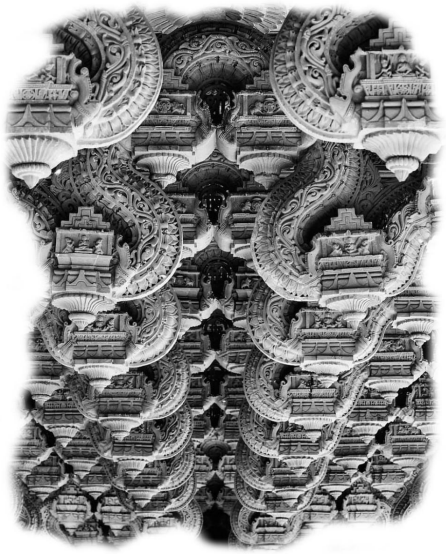
Here is an important design decision about `partition()` and its rationale. You don't have to do anything, but I'm offering this nugget here as an example of decisions you may have to make when you get to the final mile with your own algorithms.

Note that `partition()` first calculates the value of the pivot index and gets the pivot from `elems`. If you blindly follow rules laid by others you may argue that the method should check `elems[]` for emptiness before accessing it.

However, when you eventually get to squeezing cycles out of tight frequently touched code, you also get to ask how many of these checks you can eliminate if you can guarantee certain input conditions. Sometimes you say "*Well, it's easier and better to enforce that `elems` NOT be empty through the documentation.*" That is why the recursive `qsort()` or `partition()` methods are able to eschew MANY tests.[12]

The win from this decision looks like a couple of lines of code, but when you profile your code with and without these checks, you'll notice the difference.

Where is this win?



Where is the real win in the Quicksort algorithm that you see, for example, on Wikipedia?

The real win is not in what you see. But rather, in what you don't see.

As you implement your own quicksort, you'll see that it's relatively easy to get a correct implementation with many checks here and there.

In sculpting metaphors, you can say that you have now identified the right kind of rock to work on.

Years of meticulous chipping away by many master artisans result in many of the masterpieces we see around us today.

One angle with which to polish your own quicksort is to look at minimizing the number of operations within the tight loops. This is likely to give you a solid intuition about when each element is accessed. This is also when you can start worrying about the actual machine cycles a

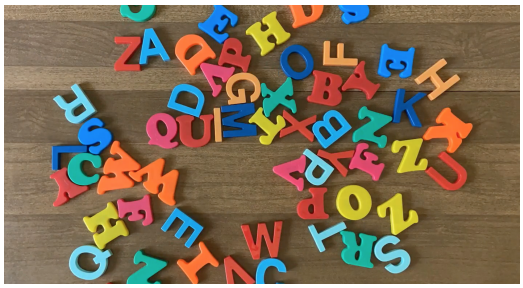
[12] If a conditional test can easily be moved up the code-tree hierarchy for no cost, it makes sense to check if you can factor the test out of the function it is in, into the function's invoker or even higher if possible.

method might consume. Some people who needlessly worry about machine cycles without a correct intuition about the algorithm don't know what they're talking about.

If all goes well you'll see benchmark results of your runs compared with reference runs and, where relevant, c++ library calls.

Icing (Edit Jun 1, 2021)

Thanks to https://reddit.com/u/lane_johnson, I'm able to share the following really cool video he put together. It's at the end of the spec for a reason. Understand the concept first, try and fail many times, and then watch this video for maximum enjoyment. You can meet Lane in the STEM center.



Lane's Quicksort Vid

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
2. Enter the secret password for this quest in the box.
3. Drag and drop your source files[13] into the button and press it.

Wait for me to complete my tests and report back (usually a minute or less).

Happy Questin While You're Restin,

&

[13] `Pivoting.h and Entry_Pass.cpp`