

Computer Science

Y. LIOW (OCTOBER 3, 2024)

Contents

103	Comparison sorting	4500
103.1	Properties of sorting algorithms <code>debug: properties-of-sorting-algorithms.tex</code>	4501
103.2	Swap and <code>std::swap</code> <code>debug: swap.tex</code>	4508
103.3	Insertion sort <code>debug: insertionsort.tex</code>	4514
103.4	Selection sort <code>debug: selectionsort.tex</code>	4538
103.5	Mergesort ... 1st Version <code>debug: mergesort1.tex</code>	4557
103.6	Mergesort ... 2nd Version <code>debug: mergesort2.tex</code>	4566
103.7	Quicksort: introduction <code>debug: quicksort-intro.tex</code>	4581
103.8	Quicksort: pivot selection <code>debug: quicksort-pivot-selection.tex</code>	4595
103.9	Quicksort: a partition strategy that is reasonable, short, and completely wrong <code>debug: quicksort-wrong.tex</code>	4600
103.10	Quicksort: pLRT partition method <code>debug: quicksort-plrt.tex</code>	4605
103.11	Quicksort: Hoare's partition strategy <code>debug: quicksort-hoare.tex</code>	4625
103.12	Quicksort: runtime analysis <code>debug: quicksort-runtime.tex</code>	4641
103.13	Comparison sort: lower bound <code>debug: comparison-sort-lower-bound.tex</code>	4648
103.14	C++: function pointers, functors, lambdas <code>debug: function-pointers-functors-lambdas.tex</code>	4649
103.14.1	Function as arguments	4649
103.14.2	Function pointers	4649
103.14.3	Functors	4652
103.14.4	Lambda expressions	4655
103.15	C++ STL <code>std::sort</code> and <code>std::stable_sort</code> <code>debug: comparison-sort-stl.tex</code>	4658
103.15.1	Basic examples	4658
103.15.2	Changing the sorting order	4659
103.15.3	Writing your own comparison	4660
103.16	Real world sorting <code>debug: comparison-sort-real-world.tex</code>	4662

Chapter 103

Comparison sorting

103.1 Properties of sorting algorithms debug:

properties-of-sorting-algorithms.tex

There are several extremely important properties of sorting algorithms besides their runtime behavior and space complexity.

A sorting algorithm is **inplace** if sorting is done directly on the array, except for a finite number of extra variables that does not depend on the array size. If an algorithm, while sorting an array of size n needs another array of size n or $n/2$ or $2n + 1$, etc., then the sorting algorithm is not inplace. inplace

If an algorithm (not necessarily a sorting one) that receives an array \mathbf{x} of size n and requires another array \mathbf{y} of size n (or $\frac{1}{2}n$ or $\frac{1}{3}n$ or $5n + 1$) to do its work, we would say that the space requirement (or space complexity) is $O(n)$ (in fact $\Theta(n)$) and that's because of \mathbf{y} . In the case of an inplace algorithm, since the amount of space needed is a constant, the space requirement is $O(1)$ (in fact $\Theta(1)$.)

A sorting algorithm is **stable** if say the algorithm sorts array \mathbf{x} in ascending order and the value at $\mathbf{x}[4]$ is \leq the value at $\mathbf{x}[9]$, then when the algorithm terminates, if the value at $\mathbf{x}[4]$ is moved to index position \mathbf{m} and the value at $\mathbf{x}[9]$ is moved to index position \mathbf{n} , then $\mathbf{m} \leq \mathbf{n}$. To be even more concrete, say you're sorting the following array in ascending order and the array has exactly two 5s: stable

		5				5	
--	--	---	--	--	--	---	--

Suppose I put tags on these two 5s:

		5 ₁				5 ₂	
--	--	----------------	--	--	--	----------------	--

Then a stable sorting algorithm must terminate with the array looking like this (the 5 that's in front stays in front):

	5 ₁	5 ₂				
--	----------------	----------------	--	--	--	--

If the algorithm terminates with this instead:

	5 ₂	5 ₁				
--	----------------	----------------	--	--	--	--

then the algorithm is not stable. Stable sorting algorithms are important because they are used by other algorithms.

A sorting algorithm is **internal** if the array being sorted is stored completely in memory. Otherwise the sorting algorithm is **external**. An external algorithm would use external storage (hard drive, tape drive, etc.) I will not talk about external sorting algorithms in CISS350. But when the data has to persist or when there is too much data to fit into memory, external algorithms come into play. This occurs for instance in CISS430.

internal
external

A sorting algorithm is a **comparison-based** algorithm if the algorithm sorts by comparing a pair of values in the array using $\leq, <, \geq, >$. Another class of sorting algorithms is the **distribution** sorting algorithm. These algorithm look at one value in the array at a time and put that value into a structure. The work done on this value does not depend on comparing it with another value in the array. Let me say that again: a distribution sorting algorithm does not compare pairs of values in the array.

comparison-based
distribution

Exercise 103.1.1. Consider the following array where each cell of the array has two values:

debug:
exercises/properties-
1/question.tex

$\{(1, "c"), (5, "b"), (7, "a"), (2, "d"), (6, "e"), (3, "a"), (7, "d"), (2, "b")\}$

- What does the array look like after sorting the array in ascending order by the first value of the cells using a stable sorting algorithm?
- What does the array look like after sorting the array in descending order by the second value of the cells using a stable sorting algorithm?

([Go to solution](#), page 4504)

□

Exercise 103.1.2. Is bubblesort inplace? stable? internal? comparison-based? What is the asymptotic runtime and space complexity? Write a bubblesort that sorts an array of `std::pairs` of (integer, string) values by the integer component of the values. Include printing after every swap. Run your code on this:

debug:
exercises/properties-
0/question.tex

$\{(1, "c"), (5, "b"), (7, "a"), (2, "d"), (6, "e"), (3, "a"), (7, "d"), (2, "b")\}$

and visually check that bubblesort is stable. ([Go to solution](#), page 4505) □

Exercise 103.1.3. A new sorting algorithm, called productsort, is used to sort pairs of positive integers by the magnitudes of the products they form.

debug: exer-
cises/stable/question.tex

Initially, the array of pairs has the following order:

$$(2, 6), (1, 2), (2, 10), (2, 2), (3, 4), (5, 4)$$

After sorting with productsort, the array of pairs has this order:

$$(1, 2), (2, 2), (3, 4), (2, 6), (5, 4), (2, 10)$$

Is productsort a stable sort? Explain.

([Go to solution](#), page [4507](#))

□

Solutions

Solution to Exercise [103.1.1](#).

debug:
exercises/properties-
1/answer.tex

(a)

$\{ (1, "c"), (2, "d"), (2, "b") (3, "a"), (5, "b"), (6, "e"), (7, "a"), (7, "d") \}$

(b)

$\{ (6, "e"), (2, "d"), (7, "d"), (1, "c"), (5, "b"), (2, "b") (7, "a"), (3, "a") \}$

□

Solution to Exercise [103.1.2](#).

Bubblesort is inplace, stable, internal, and comparison-based. The asymptotic runtime is $O(n^2)$. Space complexity is $O(1)$.

debug:
exercises/properties-
0/answer.tex

```
#include <iostream>
#include <vector>
#include <string>

typedef std::pair< int, std::string > pair;
typedef std::vector< pair > pairs;

std::ostream & operator<<(std::ostream & cout, const pairs v)
{
    cout << '{';
    std::string delim = "";
    for (auto && e: v)
    {
        cout << delim << '{' << e.first << ", " << e.second << '}';
        delim = ", ";
    }
    cout << '}';
    return cout;
}

void bubblesort(pairs & xs)
{
    int n = xs.size();
    for (int i = n - 2; i >= 0; --i)
    {
        for (int j = 0; j <= i; ++j)
        {
            if (xs[j].first > xs[j + 1].first)
            {
                pair t = xs[j];
                xs[j] = xs[j + 1];
                xs[j + 1] = t;
                std::cout << xs << '\n';
            }
        }
    }
}

int main()
{
    pairs xs {{1,"a"}, {5,"a"}, {7,"a"}, {2,"a"}, {1,"b"},
              {7,"b"}, {2,"b"}, {1,"c"}};
```



```
bubblesort(xs);  
return 0;  
}
```

The output is

```
{1, a}, {5, a}, {2, a}, {7, a}, {1, b}, {7, b}, {2, b}, {1, c}  
{1, a}, {5, a}, {2, a}, {1, b}, {7, a}, {7, b}, {2, b}, {1, c}  
{1, a}, {5, a}, {2, a}, {1, b}, {7, a}, {2, b}, {7, b}, {1, c}  
{1, a}, {5, a}, {2, a}, {1, b}, {7, a}, {2, b}, {1, c}, {7, b}  
{1, a}, {2, a}, {5, a}, {1, b}, {7, a}, {2, b}, {1, c}, {7, b}  
{1, a}, {2, a}, {1, b}, {5, a}, {7, a}, {2, b}, {1, c}, {7, b}  
{1, a}, {2, a}, {1, b}, {5, a}, {2, b}, {7, a}, {1, c}, {7, b}  
{1, a}, {2, a}, {1, b}, {5, a}, {2, b}, {1, c}, {7, a}, {7, b}  
{1, a}, {1, b}, {2, a}, {5, a}, {2, b}, {1, c}, {7, a}, {7, b}  
{1, a}, {1, b}, {2, a}, {2, b}, {5, a}, {1, c}, {7, a}, {7, b}  
{1, a}, {1, b}, {2, a}, {2, b}, {1, c}, {5, a}, {7, a}, {7, b}  
{1, a}, {1, b}, {2, a}, {1, c}, {2, b}, {5, a}, {7, a}, {7, b}  
{1, a}, {1, b}, {1, c}, {2, a}, {2, b}, {5, a}, {7, a}, {7, b}
```

Notice that initially (1, "a") is in front of (1, "b") and after the bubblesort is completed (1, "a") is still in front of (1, "b").

Solution to Exercise [103.1.3](#).

No productsort is not stable. $(2, 6) = (3, 4)$. Initially $(2, 6)$ is before $(3, 4)$, but they are switched.

debug: exer-
cises/stable/answer.tex

103.2 Swap and `std::swap`

debug: swap.tex

Here's the obvious algorithm (from CISS240) to swap the values of `x` and `y`:

```
t = x;
x = y;
y = t;
```

Note that if `x` and `y` consumes a lot of memory, of course you will run into performance problems.

C++ has a `std::swap` function template. Furthermore some C++ STL classes have `swap` methods. Run this:

```
#include <iostream>
#include <vector>

int main()
{
    int x = 0, y = 1;
    std::swap(x, y);

    std::vector< int > v {1, 2}, u {3, 4};
    std::swap(v[0], v[1]); // swap two values inside a vector

    std::swap(u, v);      // swap contents of two vectors
                          // std::swap(u, v) calls u.swap(v)

    u.swap(v);

    return 0;
}
```

Don't believe everything I say – insert some print statements to check that swaps actually occur.

Exercise 103.2.1.

debug:
exercises/swap-
0/question.tex

- Test the performance of `std::swap` against your hand-coded swap on `int` variables. Which is faster? Do this for the case of swapping values of two `int` variables and also two `int` values in an array (both static array and `std::vector`). Which swap should you use for best performance?
- Test the performance of `std::swap` against your hand-coded swap on `double` variables. Which is faster? Do this for the case of swapping values of two `double` variables and also two `double` values in an array (both static array and `std::vector`). Which swap should you use for best performance?
- Test the performance of `std::swap` against your hand-coded swap for two `std::vector` objects. Which is faster? Which swap should you use

for best performance?

([Go to solution](#), page 4510)



Exercise 103.2.2. Why is `std::swap` much faster than the obvious hand-coded swap when performing a swap on two `std::vector` objects? ([Go to solution](#), page 4512)



debug:
exercises/swap-
1/question.tex

Exercise 103.2.3. Write your own swap function template that will either use your handcoded swap computation or use `std::swap`. You want to do it in the most effective way. ([Go to solution](#), page 4513)



debug:
exercises/swap-
2/question.tex

Solutions

Solution to Exercise [103.2.1](#).

Swap int variables:

debug:
exercises/swap-
0/answer.tex

```
#include <iostream>

int main()
{
    int i = 0, j = 1;
    int t;

    for (int k = 0; k < 1000000000; ++k)
    {
        // Version 1
        t = i;
        i = j;
        j = t;

        // Version 2
        //std::swap(i, j);
    }

    return 0;
}
```

Runtime for Version 1:

real	0m1.982s
user	0m1.950s
sys	0m0.002s

Runtime for version 2:

real	0m7.403s
user	0m7.317s
sys	0m0.004s

Handed coding is faster.

Swap `std::vector< int >` objects:

```
#include <iostream>
#include <vector>

int main()
{
    std::vector< int > i(1000000);
    std::vector< int > j(1000000);
    std::vector< int > t;

    for (int k = 0; k < 1000; ++k)
    {
        // Version 1
        t = i;
        i = j;
        j = t;

        // Version 2
        //std::swap(i, j);
    }

    return 0;
}
```

Runtime for version 1:

real	0m1.664s
user	0m1.632s
sys	0m0.004s

Runtime for version 2:

real	0m0.012s
user	0m0.008s
sys	0m0.003s

`std::swap` is faster.

Solution to Exercise [103.2.2](#).

debug:
exercises/swap-
1/answer.tex

From CISS245 you know that one way to implement a class with memory intensive objects is to use pointers to point to the data in the heap, especially when you want to control the allocation/deallocation of the memory used by the objects:

```
class C
{
    ...

    int * p_; // p_ points to a huge array in the heap
};
```

In that case, if `t`, `x`, `y` are `C` objects, instead of swapping all the values of the arrays that `x.p_` and `y.p_` point to, it's faster to swap the pointer values of `x.p_` and `y.p_`:

```
t.p_ = x.p_;
x.p_ = y.p_;
y.p_ = t.p_;
```

Right? There are other details, but this is the main idea of the speedup. This illustrates (once again) the power of pointers.

This is why in the `std::vector` class there's a `swap` method, in the `std::string` class there's a `swap` method, etc.

If `x` and `y` are objects from some C++ class with `swap` method, then calling

```
std::swap(x, y);
```

will result in calling

```
x.swap(y);
```

Solution to Exercise [103.2.3](#).debug:
exercises/swap-
2/answer.tex

```
// file: myswap.h (or any suitable name)

#ifndef MYSWAP_H
#define MYSWAP_H

template < typename T >
inline
void simpleswap(T & x, T & y)
{
    T t = x;
    x = y;
    y = t;
}

namespace std
{
    template <>
    inline
    void swap(int & x, int & y)
    {
        simpleswap(x, y);
    }

    template <>
    inline
    void swap(double & x, double & y)
    {
        simpleswap(x, y);
    }

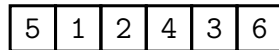
    template <>
    inline
    void swap(char & x, char & y)
    {
        simpleswap(x, y);
    }

    template <>
    inline
    void swap(bool & x, bool & y)
    {
        simpleswap(x, y);
    }
}

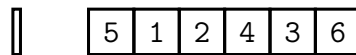
#endif
```


103.3 Insertion sort debug: insertionsort.tex

The idea behind the insertion sort is pretty simple. Suppose I have this bunch of numbers:

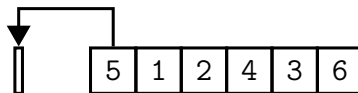


Let me create a list of *sorted* numbers on the left:

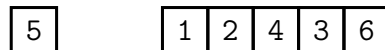


Initially of course nothing is sorted: the list on the left is empty. So the list on the left is “work done” and the list on the right is “work to be done”. I’m going to call the array of the left the DONE pile (or list) and the array on the right the TODO pile (or list).

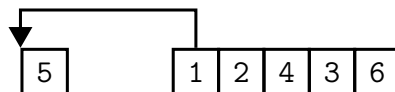
STEP 1. The first thing I’ll do is to look at the first number from my TODO pile (reading left-to-right), i.e., 5. I remove 5 from the TODO pile and put 5 immediately to my DONE pile (the left pile):



This is what I get:



STEP 2. The next piece of work to do is the leftmost number in the TODO pile, i.e., the number 1. The number 1 is put into the DONE pile so that it is sorted:

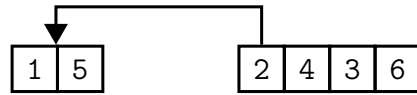


This is what I get:



Let’s ignore how 1 is placed in the left pile for the time being. I’ll come back to that issue when we have more elements in the left pile. Anyway, let’s do the next value, i.e., 2.

STEP 3. I remove 2 from the TODO pile, and put that into the DONE pile in the right place:

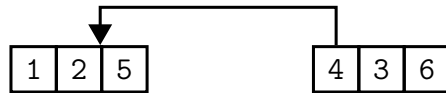


and get this:



Notice that the DONE pile is sorted.

STEP 4. The next value removed from the TODO pile is 4 and I put that in its right place in the DONE pile:

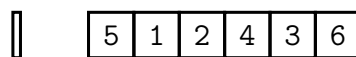


to get this:

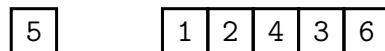


OK. I think you get the general idea. Now let me show you how to “insert” a value taken from the right pile into the left pile (that’s why this is called the insertion sort ... duh). Let’s rewind ...

If you look at step 1: From



to



And from step 2 to 3, i.e., to



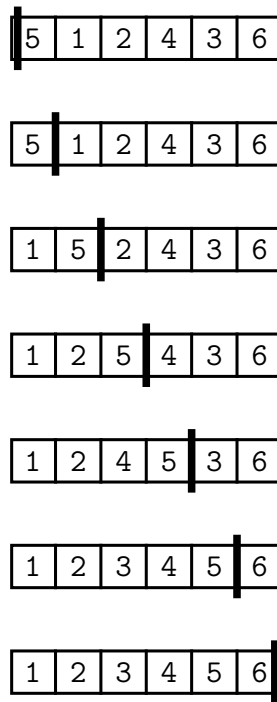
So how does 1 find it’s place?

Here’s a memory aid for the big picture: You think of two piles: the DONE and TODO piles. The DONE is a queue of people getting ready to buy tickets

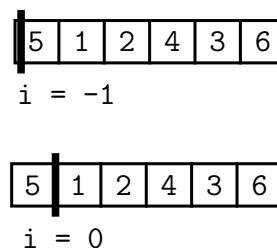
at a movie (the box office is not opened yet). People in the TODO pile leave the TODO pile and get into the ticket queue, i.e., the DONE pile. However these people are part of a mafia and they have ranks, 1 being rank 1 (the mafia boss), etc. So when a person joins the DONE pile, he can jump over people to get to his correct place based on his mafia rank. (The technical terms is that the DONE pile behaves like a priority queue.)

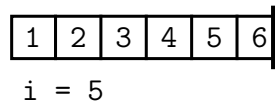
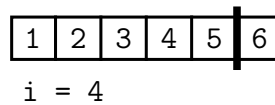
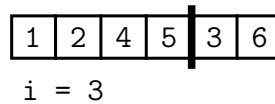
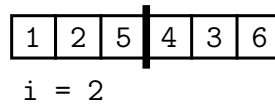
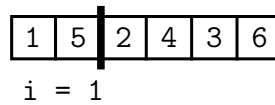
OK ... back to the array of numbers ...

First you realized in fact that all the computations can be done in a *single* array. To show you what I mean, let me put a | to divide the *same* array into two parts, the DONE pile and the TODO pile. The above computations of insertion sort would look like these:



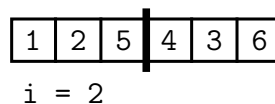
Of course there's no | in arrays for any programming language. However we can use a variable to tell us where the | is. For instance



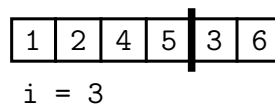


See that? Basically *i* tells me what is the largest index value of the DONE pile. Instead of running *i* from -1 to 5, you can also run *i* from 0 to 6. In that case the *i* represents the smallest index of the TODO pile. Or you can think of this *i* as the size of the DONE pile.

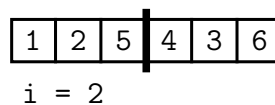
Now let's see how we take a value from the right pile (TODO pile) into the left pile (DONE pile). I'll use the case of putting 4 in it's right place. So we start with this:



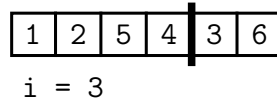
and get this:



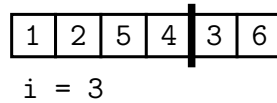
Here's how I can achieve the above. I start with this:



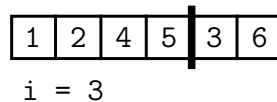
and I move the 4 into the DONE pile:



Note that i has value 3. In the DONE pile, each value is in its correct place except for 4, the newcomer. Therefore 4 has to *move past* values which are greater than 4. For instance 5 is greater than 4. So we move 4 past 5, i.e., from



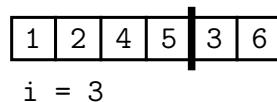
we get this



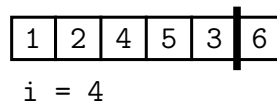
At this point, the value to the left of 4 is 2 which *is* smaller than or is equal to 4. That tells me that 4 has found its right place.

So here's the meaning of i : At the end of the i -th pass, the values in $x[0]$, ..., $x[i]$ are sorted. At the beginning of that pass, the value at $x[i]$ is the value that starts looking at its left neighbor and moves past this neighbor if necessary to make the DONE pile sorted.

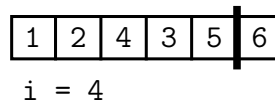
Let me do one iteration slowly: let's move 3 from the TODO pile into the DONE file. From this



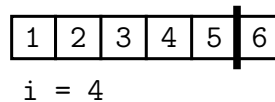
I get



and then



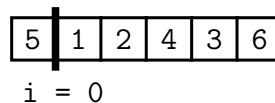
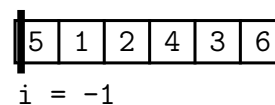
and finally this:



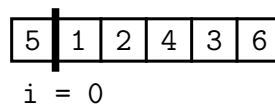
At this point, the left neighbor of 3 (i.e., 2) is ≤ 3 . So I stop.

That's the main idea behind the insertion sort.

Now, let's think about initialization of our insertion sort. Here are the first two steps of our insertion sort computation:



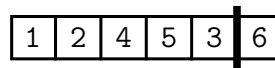
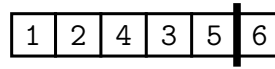
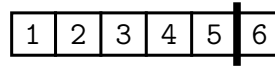
Well ... nothing's really happening to the array between the two diagrams. That's pretty obvious if you think about it. 5 is the first person to join the DONE pile. So obviously 5 is in the right spot because he's (it's?) the only person in the DONE pile and you can't have a DONE pile with one value to be not sorted. To view it in another way, 5 can't compare himself with his left neighbor because ... well ... he/it has no left neighbor!!! So we might as well start off with this as the first step:



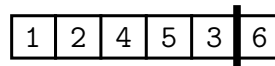
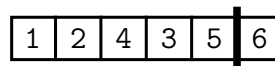
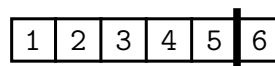
So for my initialization, I might as well set i to 0 right away.

Now let me think carefully about the computation that moves a new member of the DONE pile to the left. I need to keep track of the position of this value.

Take a look at the case when $x[4]$ is added to the DONE pile and has to move to its correct place:

 $i = 4$  $i = 4$  $i = 4$

Be careful! I can't use i to follow the index of the value 3 that is slowly moving to its correct place. Why? Because the value in i tells me what is the last index of the DONE pile, which tells me what's the next value in TODO to process (and it also tells me when to stop insertion sort). So I need *another* variable, say j , that is initialized to i and j is the index position of the value that is moving in the DONE pile to find its correct place. To see j in action, look at the case when $i = 4$:

 $i = 4$
 $j = 4$  $i = 4$
 $j = 3$  $i = 4$
 $j = 2$

Do you see that 3 going past 4 and then 5? Obviously there's a loop here for j .

Of course when doing a loop, you always need to know the initial value of the index variable and when to stop. For i , I will start with $i = 1$. (Forget about $i = 0$... remember that when $x[0]$ enters the left pile, there's nothing to do since $x[0]$ would be the only guy in the left pile.) The last value for i is of course $n - 1$ where n is the size of the array.

As for j , you see from the above that j starts with the value of i . At each iteration of the j -loop, I compare $x[j]$ against $x[j - 1]$ and swap their values if $x[j]$ is smaller than or equal to $x[j - 1]$. When do I stop? When $x[j - 1]$ is smaller than $x[j]$. Right? There's one thing you have to be careful of: When j is 0 you cannot compare $x[j]$ with $x[j - 1]$ since there's nothing to the left of $x[0]$!!! So there are actually *two* termination condition for the j -loop.

Now we are ready to write the pseudocode:

```
for i = 1, 2, 3, ..., n - 1:
    for j = i, i - 1, ..., 1:
        if x[j - 1] > x[j]:
            swap x[j - 1] and x[j]
        else:
            break
```

Remember that for the outer i -loop, we start with $i = 1$. As for the j -loop, the j index basically follows the position of the value as it moves. In this case, we also stop at $j = 1$ for the same reason as above. So i starts at 1 and j stops at 1 for pretty much the same reason.

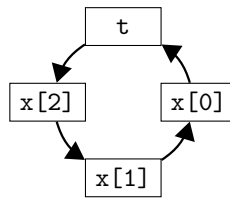
If you prefer not to have a break of course you can write this:

```
for i = 1, 2, 3, ..., n - 1:
    j = i
    while j >= 1 and x[j - 1] > x[j]:
        swap x[j - 1] and x[j]
        j = j - 1
```

In each pass of the insertion sort, you will be performing a sequence of swaps. This should remind you of an exercise in CISS240. Suppose I swap $x[0]$, $x[1]$, then swap $x[1]$, $x[2]$, then swap $x[2]$, $x[3]$. Of course each swap has 3 assignments. The above 3 swaps takes 9 assignments. If you perform k swaps, you are doing $3k$ assignments. The above can also be done this way:

```
t = x[0]
x[0] = x[1]
x[1] = x[2]
x[2] = t
```

You perform a cycle of assignments:



I'll call this a **rotation**. This takes 4 assignments, which is $4/9 = 44\%$ of the number of assignments if you do 3 swaps. Therefore this technique is $9/4 = 225\%$ faster than doing swaps. This is a very important trick and will appear again in many different algorithms. Remember it!!!

rotation

Exercise 103.3.1. If you perform k swaps among k variables, you'll be doing $3k$ assignments. When k is large, what is the performance improvement if you do rotation instead of swaps? (Go to solution, page 4527) \square

debug:
exercises/insertionsort-
0/question.tex

With this optimization, I finally get:

insertionsort

```

ALGORITHM: insertionsort
INPUT: x - array
        n - size of x

for i = 1, 2, 3, ..., n - 1:
    t = x[i]
    j = i
    while j >= 1 and x[j - 1] > t:
        x[j] = x[j - 1]
        j = j - 1
    x[j] = t
  
```

Note that there's a small variant of the above insertionsort. The last statement above

```

...
x[j] = t
  
```

can be replaced by

```

...
if i != j:
    x[j] = t
  
```

Since j tracks down the final resting place of $x[i]$, if $i == j$, then $x[j] = t$ is redundant. The above small change is useful if the assignment $x[j] = t$ is costly. It should definitely not be used if you are sorting values such as `ints`, `floats`, `doubles`. However, you do want to use this optimization if you are sorting strings, especially long ones.

[INVARIANTS? PROOF OF CORRECTNESS BY INDUCTION? ciss358]

Now for the runtime analysis. Here's our insertion sort again:

```

for i = 1, 2, 3, ..., n - 1:
    t = x[i]
    j = i
    while j >= 1 and x[j - 1] > t:
        x[j] = x[j - 1]
        j = j - 1
    x[j] = t

```

For the inner loop, in the worst case, j will run from i to 1. Therefore the inner loop has a runtime of $O(i)$ i.e., $A \cdot i + B$ for some constants A and B . Altogether the worst runtime is

$$T_w(n) = A(1 + 2 + \cdots + n) + Bn + C = O(n^2)$$

where C is some constant. For the best case scenario, the inner loop does not run at all. Therefore, for each value of i , the inner loop has a constant runtime of say A . Hence

$$T_b(n) = An + C = O(n)$$

for some constant C . I will not derive it, but the average runtime is $O(n^2)$.

Exercise 103.3.2. Perform insertion sort on the array $\{1, 3, 5, 2, 4, 6, 0\}$, listing every state of the array in your computation, including the value of variables i , j , and t . ([Go to solution](#), page 4528) \square

debug:
exercises/insertionsort-
1/question.tex

Exercise 103.3.3. Compute the best and worst runtime of the above version of insertion sort.

debug:
exercises/insertionsort-
2/question.tex

1. What scenario will give you the best runtime?
2. What scenario will give you the worst runtime?
3. What about the average?

([Go to solution](#), page 4529) \square

Exercise 103.3.4. Is insertion sort inplace? comparison-based? stable? internal? What is the space complexity? ([Go to solution](#), page 4530) \square

debug:
exercises/insertionsort-
3/question.tex

Exercise 103.3.5. Modify the insertion sort so that the right index position for t to move into is computed using binary search. Does it change the asymptotic runtime? Test your implementation and time it. Does it improve the

debug:
exercises/insertionsort-
4/question.tex

wall-clock runtime? ([Go to solution](#), page 4531)

□

Exercise 103.3.6. Modify the previous implementation in the following way: There's no need to always scan right-to-left one step at a time in the left (sorted) pile. Why not left-to-right? Keep an average of the values in the sorted part of the array. If the value to be inserted is less than the average, scan the sorted part of the array left-to-right, and if the value to be inserted is at least the average, scan the sorted part of the array right-to-left. When the right index position is found, then move the relevant values to the right by one step and insert the value. Test your implementation. Compute the runtime.

debug:
exercises/insertionsort-
5/question.tex

For the above versions, compare their wall-clock times for different n and different scenarios including best and worst case. ([Go to solution](#), page 4532)

□

Exercise 103.3.7. Notice that the best case of the insertion sort occurs when the list is already sorted. Another way to think of it is that for an array insertion to the end of the array is cheap. The worst case is when inserting at the head of the array. (Actually the big-O of the runtime of the worst and the average are the same.) Here's an idea. Create an array that allows *negative* index values. This allows you to add a value to the left of index 0, i.e., at index -1 ; and then allows you to add a value to the left of index -1 , i.e., at index -2 , etc. Of course C/C++ and most other languages do not have this feature. You have to simulate this yourself. For instance if given an array x of size 10, all you need to do is to create an array y of size 20 and start off your insertion sort by putting $x[0]$ at $y[10]$. You should be able to figure out the rest. [By the way, this tells you that you should implement a special array class that allows you to specify the starting index value and the ending index value. So if X is the name of this class, then $X(-10, 9)$ will give you an array of size 20 with index positions $-10, -9, \dots, -1, 0, 1, \dots, 9$. Once you have this class, re-implementing the above version of insertion sort is going to be a lot cleaner.] ([Go to solution](#), page 4533)

debug:
exercises/insertionsort-
6/question.tex

□

Exercise 103.3.8. Only if you know linked lists (see later chapter on linked lists): Notice that the best case of the insertion sort occurs when the list is already sorted. Another way to think of it is that for an array insertion to the end of the array is cheap. The worst case is when inserting at the head of the array. (Actually the big-O of the runtime of the worst and the average are the same.) Well, if you're putting values into a doubly linked list, inserting at head and tail and anywhere in-between is cheap. Use this idea to implement

debug:
exercises/insertionsort-
7/question.tex

the insertion sort. After sorting the values into the linked list, copy the values back to the array. Compute the runtime.

Combine the “average” and linked list method, i.e., keep an average of the sorted part of the list and determine whether to scan from head or from tail based on the average. (Go to solution, page 4534) \square

Exercise 103.3.9. What about this variant of the insertion sort that inserts a *chunk* of values? Suppose after you have found a place to insert a value, you insert a chunk of ascending values. For instance suppose you have the following scenario:

debug:
exercises/insertionsort-
8/question.tex

1	2	3	50	51	52	10	11	15	5	2	1	42	34	24	99
---	---	---	----	----	----	----	----	----	---	---	---	----	----	----	----

Wouldn't it be better to move the whole chunk of 10,11,15 between 3 and 50? The result would be this:

1	2	3	10	11	15	50	51	52	5	2	1	42	34	24	99
---	---	---	----	----	----	----	----	----	---	---	---	----	----	----	----

This seems to be best if the array has chunks of ascending values. For instance the following array has many ascending chunks of length 3:

10	11	12	1	2	3	78	79	80	7	8	9	51	52	53	99
----	----	----	---	---	---	----	----	----	---	---	---	----	----	----	----

Design an algorithm for this chunk-based insertion sort, implement it and test it against the usual insertion sort with test data with ascending chunks of length 10, 20, 30. Is it better than the usual insertion sort?

Name an example where this version of insertionsort will run faster than the usual version of insertionsort.

(There's yet another thing that can be done to the above idea. In the left pile, there's a huge gap between 3 and 50. The values in the right pile {10,11,15,5,2,1,42,34,24} can be inserted between 3 and 50. So you can either first sort {10,11,15,5,2,1,42,34,24} and then move this chunk or move this chunk and then sort. See it?) (Go to solution, page 4535) \square

Exercise 103.3.10. Write the pseudocode for insertionsort where the TODO pile is on the left and the DONE pile is on the right. Then implement it in C++. (Go to solution, page 4536) \square

debug:
exercises/insertionsort-
9/question.tex

Exercise 103.3.11. Note that insertionsort moves values to the left (the DONE pile is on the left). Think of this as one-way insertionsort. Design a two-way insertionsort, i.e., there are two DONE piles. ([Go to solution](#), page [4537](#)) \square

debug:
exercises/insertionsort-
10/question.tex

Solutions

Solution to Exercise [103.3.1](#).

Using the rotation optimization, there are $k + 1$ assignments. Since

$$\frac{3k}{k+1}$$

approaches 3 when k approaches infinity. Or, using very proper math notation, I should write

$$\lim_{k \rightarrow \infty} \frac{3k}{k+1} = 3$$

Therefore there is a 300% performance improvement.

This performance improvement also applies to insertion sort. Although this changes the wall-clock runtime, it does not change the asymptotic runtime.

□

debug:
exercises/insertionsort-
0/answer.tex

Solution to Exercise [103.3.2](#).

debug:
exercises/insertionsort-
1/answer.tex

<pre>{1,3,5,2,4,6,0}, i = 1, j = 1, t = 3 {1,3,5,2,4,6,0}, i = 1, j = 1, t = 3</pre>
--

Solution to Exercise [103.3.3](#).

debug:
exercises/insertionsort-
2/answer.tex

We first measure the inner loop. The best case is when the condition of the while-loop is never satisfied for each pass (i.e. for each i value). This means that the while-loop is always $O(1)$. Therefore the body of the i -loop is also constant as a whole. Therefore for the whole algorithm, the runtime is of the form $A(n - 1)$ which is $O(n)$. This best case scenario occurs when the body of the while-loop never executes, i.e., when the array is already sorted so that *no* value in the array moves past its left neighbor.

The worst case scenario is when the array is in reverse, i.e., the values are in descending order.

Now for the average case. The number of times the i -loop executes is the same. However the j -loop can have an early or late termination. So assuming that the average case means that for the i -th pass, the number of values to move left is half of the best and worst case, then the number of values moved in the i -th pass is about $i/2$ (ignoring floor and ceiling). Therefore the total runtime for this average case is $O(\sum_{i=1}^{n-1} i/2) = O((1/2) \sum_{i=1}^{n-1} i) = O(\sum_{i=1}^{n-1} i) = O(n^2)$. \square

Solution to Exercise [103.3.4](#).

Solution not provided.

debug:
exercises/insertionsort-
3/answer.tex

Solution to Exercise [103.3.5](#).

Solution not provided.

debug:
exercises/insertionsort-
4/answer.tex

Solution to Exercise [103.3.6](#).

Solution not provided.

debug:
exercises/insertionsort-
5/answer.tex

Solution to Exercise [103.3.7](#).

Solution not provided.

debug:
exercises/insertionsort-
6/answer.tex

Solution to Exercise [103.3.8](#).

Solution not provided.

debug:
exercises/insertionsort-
7/answer.tex

Solution to Exercise [103.3.9](#).

Solution not provided.

debug:
exercises/insertionsort-
8/answer.tex

Solution to Exercise [103.3.10](#).

Solution not provided.

debug:
exercises/insertionsort-
9/answer.tex

Solution to Exercise [103.3.11](#).

Solution not provided.

debug:
exercises/insertionsort-
10/answer.tex

103.4 Selection sort debug: selectionsort.tex

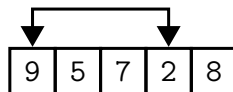
Selection sort is easy. Let's say I'm given an array x

9	5	7	2	8
---	---	---	---	---

of size $n = 5$ and I want to sort this in ascending order using the selection sort algorithm.

PASS 1. You scan $x[0], \dots, x[n - 1]$ for the smallest value and swap that with $x[0]$. In the above example, the smallest value is 2 which is at index position 3. You swap that with the value at index 0

9	5	7	2	8
---	---	---	---	---



and get this:

2	5	7	9	8
---	---	---	---	---

Of course this means that 2 has found its right place. I just repeat the above process on $x[1], \dots, x[n - 1]$ (and not touch $x[0]$ anymore). To make it easier to read the computational state of the array, I'm going to insert a $|$ to separate the 2 from the rest. So I'm here right now:

2		5	7	9	8
---	--	---	---	---	---

The values to the left of $|$ is sorted while the values to the right of $|$ are values to be processed. Similar to insertion sort, the left of $|$ is the DONE pile and the right of $|$ is the TODO pile. This is the end of pass 1.

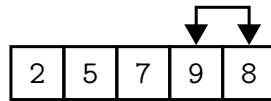
PASS 2. The smallest among $x[1], \dots, x[n - 1]$ is at index 1. In this case 5 is at the right place, so there's nothing to do. I'm now at this state:

2	5		7	9	8
---	---	--	---	---	---

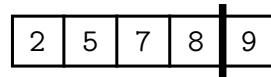
PASS 3. In the next pass, 7 is also in its right place. So here's the result at the end of this pass:

2	5	7		9	8
---	---	---	--	---	---

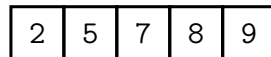
PASS 4. After that, the smallest is 8 which I will swap with 9



to get this:



And I'm done!!! There's obviously nothing to do for 9 since all the smaller values have found their place, 9 must be the largest. So I now have the sorted array:



Here's algorithm:

```
ALGORITHM: selectionsort
INPUTS: x -- array
        n -- size of x

for i = 0, 1, 2, ..., n - 2:
    minvalue = x[i]
    minindex = i
    for j = i + 1, ..., n - 1:
        if x[j] < minvalue:
            minvalue = x[j]
            minindex = j
    if minindex is not i:
        swap x[i] and x[minindex]
```

You can factor out the part of the algorithm that computes the minimum index:

```

ALGORITHM: find_minindex
INPUTS: x -- array
        i -- start index
        n -- size of x
minvalue = x[i]
minindex = i
for j = i + 1, ..., n - 1:
    if x[j] < minvalue:
        minvalue = x[j]
        minindex = j
return minindex

ALGORITHM: selectionsort
INPUTS: x -- array
        n -- size of x
for i = 0, 1, 2, ..., n - 2:
    minindex = find_minindex(x, i, n)
    if minindex is not i:
        swap x[i] and x[minindex]

```

Exercise 103.4.1.

debug:
exercises/selectionsort-
0/question.tex

- (a) Execute the selection sort algorithm on the following array of values

$$\{4, 5, 2, 7, 2\}$$

writing down the array at the end of each pass.

- (b) How many passes are there?

([Go to solution](#), page 4544)

**Exercise 103.4.2.**

debug:
exercises/selectionsort-
1/question.tex

- (a) Execute the selection sort algorithm on the following array of values

$$\{5, 3, -1, 4, 5, 2, 7, 2, 8, 5\}$$

writing down the array at the end of each pass.

- (b) How many passes are there?

([Go to solution](#), page 4545)



Exercise 103.4.3. Is selection sort inplace? Stable? What is the space complexity? ([Go to solution](#), page 4546)

debug:
exercises/selectionsort-
2/question.tex



Exercise 103.4.4.debug:
exercises/selectionsort-
3/question.tex

- (a) Describe arrays that will give the best runtime.
- (b) Test it by creating a suitable array of size 5 and executing the selection sort algorithm on this array. Count the total number of swaps. (The part of the algorithm that computes the minimum is the same for all arrays of size 5.)
- (c) What about the best runtime for an array of size n ? How many swaps are there?

[\(Go to solution, page 4547\)](#)**Exercise 103.4.5.**debug:
exercises/selectionsort-
4/question.tex

- (a) Describe arrays that will give the worst runtime.
- (b) Test it by creating a suitable array of size 5 and executing the selection sort algorithm on this array. Count the total number of swaps. (The part of the algorithm that computes the minimum is the same for all arrays of size 5.)
- (c) What about the worst runtime for an array of size n ? How many swaps are there?

[\(Go to solution, page 4548\)](#)**Exercise 103.4.6.**debug:
exercises/selectionsort-
5/question.tex

- (a) Is the selection sort a stable sorting algorithm?
- (b) Here's our selection sort algorithm again:

```
for i = 0, 1, 2, ..., n - 2:
    minvalue = x[i]
    minindex = i
    for j = i + 1, ..., n - 1:
        if x[j] < minvalue:
            minvalue = x[j]
            minindex = j
    if minindex is not i:
        swap x[i] and x[minindex]
```

What if I change it to this:

```
for i = 0, 1, 2, ..., n - 2:
    minvalue = x[i]
    minindex = i
    for j = i + 1, ..., n - 1:
        if x[j] <= minvalue:
            minvalue = x[j]
            minindex = j
    if minindex is not i:
        swap x[i] and x[minindex]
```

Is there any difference between this version and the original?

([Go to solution](#), page 4549)



Exercise 103.4.7. Compute the best and worst runtimes of the selection sort.

([Go to solution](#), page 4550)



debug:
exercises/selectionsort-
6/question.tex

Exercise 103.4.8.

debug:
exercises/selectionsort-
7/question.tex

- (a) Modify the selection sort algorithm above so as to sort an array in descending order.
- (b) Execute the descending selection sort algorithm on this array

$\{5, 3, -1, 4, 5, 2, 7, 2, 8, 5\}$

([Go to solution](#), page 4551)



Exercise 103.4.9.

debug:
exercises/selectionsort-
8/question.tex

- (a) The (ascending) selection sort above finds the smallest value in the subarray $x[i], \dots, x[n - 1]$ and puts it at $x[i]$. Modify the algorithm so that it finds the maximum in a suitable subarray and puts it in the right place.
- (b) Test your algorithm with this array:

$\{5, 3, -1, 4, 5, 2, 7, 2, 8, 5\}$

([Go to solution](#), page 4552)



Exercise 103.4.10. Modify the selection sort so that in each pass, you find the minimum *and* maximum in a suitable subarray and put them in their right places. Implement your algorithm, test it, and compute the runtimes. ([Go](#)

debug:
exercises/selectionsort-
9/question.tex

to [solution](#), page 4553)

□

Exercise 103.4.11. Be careful with this one ... Notice that using the same idea in the selection sort, instead of finding a minimum value to put at the right place, perhaps we can try to find a *chunk* of minimum *values* to insert at the right place.

debug:
exercises/selectionsort-
10/question.tex

For instance suppose you start off with this array:

4	5	9	7	8	9	0	1	2	3	8	6	7	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---

your algorithm should identify this chunk:

4	5	9	7	8	9	0	1	2	3	8	6	7	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---

so that when these two chunks were swapped:

4	5	9	7	8	9	0	1	2	3	8	6	7	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---

you get this:

0	1	2	3	8	9	4	5	9	7	8	6	7	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---

Design such an algorithm. Test your algorithm! Implement your algorithm and test your implementation thoroughly. ([Go to solution](#), page 4554) □

Exercise 103.4.12.

debug:
exercises/selectionsort-
11/question.tex

- Write a recursive version of the selection sort algorithm.
- Implement your recursive selection sort algorithm.
- Test your implementation!

([Go to solution](#), page 4555)

□

Exercise 103.4.13. When is an insertion sort better than a selection sort? When is a selection sort better than an insertion sort? ([Go to solution](#), page 4556) □

debug:
exercises/selectionsort-
12/question.tex

Solutions

Solution to Exercise [103.4.1](#).

debug:
exercises/selectionsort-
0/answer.tex

(a)

Pass 1 : $\{2, 5, 4, 7, 2\}$

Pass 2 : $\{2, 2, 4, 7, 5\}$

Pass 3 : $\{2, 2, 4, 7, 5\}$

Pass 4 : $\{2, 2, 4, 5, 7\}$

(b) There are 4 passes.

Solution to Exercise [103.4.2](#).

Solution not provided.

debug:
exercises/selectionsort-
1/answer.tex

Solution to Exercise [103.4.3](#).

Solution not provided.

debug:
exercises/selectionsort-
2/answer.tex

Solution to Exercise [103.4.4](#).

Solution not provided.

debug:
exercises/selectionsort-
3/answer.tex

Solution to Exercise [103.4.5](#).

Solution not provided.

debug:
exercises/selectionsort-
4/answer.tex

Solution to Exercise [103.4.6](#).

Solution not provided.

debug:
exercises/selectionsort-
5/answer.tex

Solution to Exercise [103.4.7](#).

Solution not provided.

debug:
exercises/selectionsort-
6/answer.tex

Solution to Exercise [103.4.8](#).

Solution not provided.

debug:
exercises/selectionsort-
7/answer.tex

Solution to Exercise [103.4.9](#).

Solution not provided.

debug:
exercises/selectionsort-
8/answer.tex

Solution to Exercise [103.4.10](#).

Solution not provided.

debug:
exercises/selectionsort-
9/answer.tex

Solution to Exercise [103.4.11](#).

Solution not provided.

debug:
exercises/selectionsort-
10/answer.tex

Solution to Exercise [103.4.12](#).debug:
exercises/selectionsort-
11/answer.tex

```
ALGORITHM: find_minindex_helper
INPUTS: x -- array
        i -- start index
        n -- size of x
        minvalue -- running min value
        minindex -- running min index
if i >= n:
    return minindex
else:
    if x[i] < minvalue:
        return find_minindex_helper(x, i + 1, n, x[i], i)
    else:
        return find_minindex_helper(x, i + 1, n, minvalue, minindex)

ALGORITHM: find_minindex
INPUTS: x -- array
        i -- start index
        n -- size of x
return find_minindex_helper(x, i + 1, n, x[i], i)

ALGORITHM: selectionsort_helper
INPUTS: x -- array
        i -- start index
        n -- size of x

if i >= n - 2:
    return
else:
    minindex = find_minindex(x, i, n)
    if minindex is not i:
        swap x[i] and x[minindex]
    selectionsort_helper(x, i + 1, n)

ALGORITHM: selectionsort
INPUTS: x -- array
        n -- size of x
selectionsort_helper(x, 0, n)
```

Solution to Exercise [103.4.13](#).

Insertion sort can be better than selection sort when the list is partially sorted, while selection sort can be better than insertion sort when exchanges/swaps are expensive (e.g., in terms of time or memory space).

debug:
exercises/selectionsort-
12/answer.tex

103.5 Mergesort ... 1st Version debug: mergesort1.tex

Before I talk about the actual mergesort, I want to talk about the version that is easier to understand but uses a lot more memory.

The first version of mergesort will use extra arrays. Once I have described the idea behind mergesort, in the next section, I'll talk about how you reduce the use of auxiliary arrays.

The version of mergesort for this section will accept an array and return the array sorted. Mergesort use two ideas: split and merge. Also, mergesort (the CISS350 version) uses recursion. There's another faster version (the CISS358 version) that does not use recursion.

The idea behind mergesort is very simple.

When I send an array to mergesort, mergesort splits the array into two equal halves. If the array has an odd size, then the first array is 1 smaller than the second. I'll call these two arrays **left** and **right**. Mergesort will now recurse: it will call mergesort with **left** and mergesort with **right**. In later mergesort function calls, when a mergesort receives a tiny array, an array of size 0 or size 1, it will just return the array since an array of size 0 or size 1 is already sorted.

The above is what happens when a mergesort function calls forward. But what happens backward on return? A mergesort will call mergesort with **left** and mergesort with **right**. Let me call the array **x**. At this point our pseudocode looks like this:

```
mergesort(x):
  if x has size 0 or 1:
    return x
  else:
    split x into left and right
    left = mergesort(left)
    right = mergesort(right)
    // left and right are now sorted
    ...
```

In the recursive part of the function, mergesort will now merge the sorted **left** and the sorted **right**. Merge basically combines left and right into a sorted array:

```
mergesort(x):  
    if x has size 0 or 1:  
        return x  
    else:  
        split x into left and right  
        left = mergesort(left)  
        right = mergesort(right)  
        ret = merge(left, right)  
        return ret
```

Merge is easy: You initialize an empty array, say **ret** and then you scan **left** and **right** left-to-right, pick the smaller value and put that into **ret**. For instance if you have

$$\begin{aligned}\text{left} &= \{\underline{2}, 5, 6, 9\}, & \text{right} &= \{1, \underline{3}, 4, 8\} \\ \text{ret} &= \{\}\end{aligned}$$

you put 1 of **right** into **ret**. Now we look at

$$\begin{aligned}\text{left} &= \{\underline{2}, 5, 6, 9\}, & \text{right} &= \{1, \underline{3}, 4, 8\} \\ \text{ret} &= \{1\}\end{aligned}$$

The smaller value is now 2 from **left**. So I put 2 into **ret**

$$\begin{aligned}\text{left} &= \{2, \underline{5}, 6, 9\}, & \text{right} &= \{1, \underline{3}, 4, 8\} \\ \text{ret} &= \{1, 2\}\end{aligned}$$

And then

$$\begin{aligned}\text{left} &= \{2, \underline{5}, 6, 9\}, & \text{right} &= \{1, 3, \underline{4}, 8\} \\ \text{ret} &= \{1, 2, 3\}\end{aligned}$$

Etc. At some point, one of the arrays will be used up. Break this loop, and copy the remaining values of the other unfinished array into **ret**. That's it. The level of difficulty of array merge computation is more or less a CISS240 exercise. Ultimately all values of **left** and **right** are dumped into **ret** so that **ret** is sorted. **ret** is then returned.

```
ALGORITHM: merge
INPUTS: left, right -- two sorted arrays
OUTPUT: ret -- an array

let i = 0 and j = 0 and k = 0.
i is an index for left, j for right, k for ret.
let ret be an array of sufficient size.

while i < size of left and j < size of right:
    if left[i] <= right[j]:
        copy left[i] into ret[k], increment i and k
    else:
        copy right[j] into ret[k], increment j and k

copy unused part of left to ret
copy unused part of right to ret
return ret
```

Merging is a common operation in CS. The above is a merge of two sorted arrays to produce a sorted array. In the future, you'll see merge of other structures: merge of sets, merge of heaps, etc.

Mergesort is a very important sorting algorithm because historically it was the first to break the quadratic asymptotic runtime and furthermore it's stable.

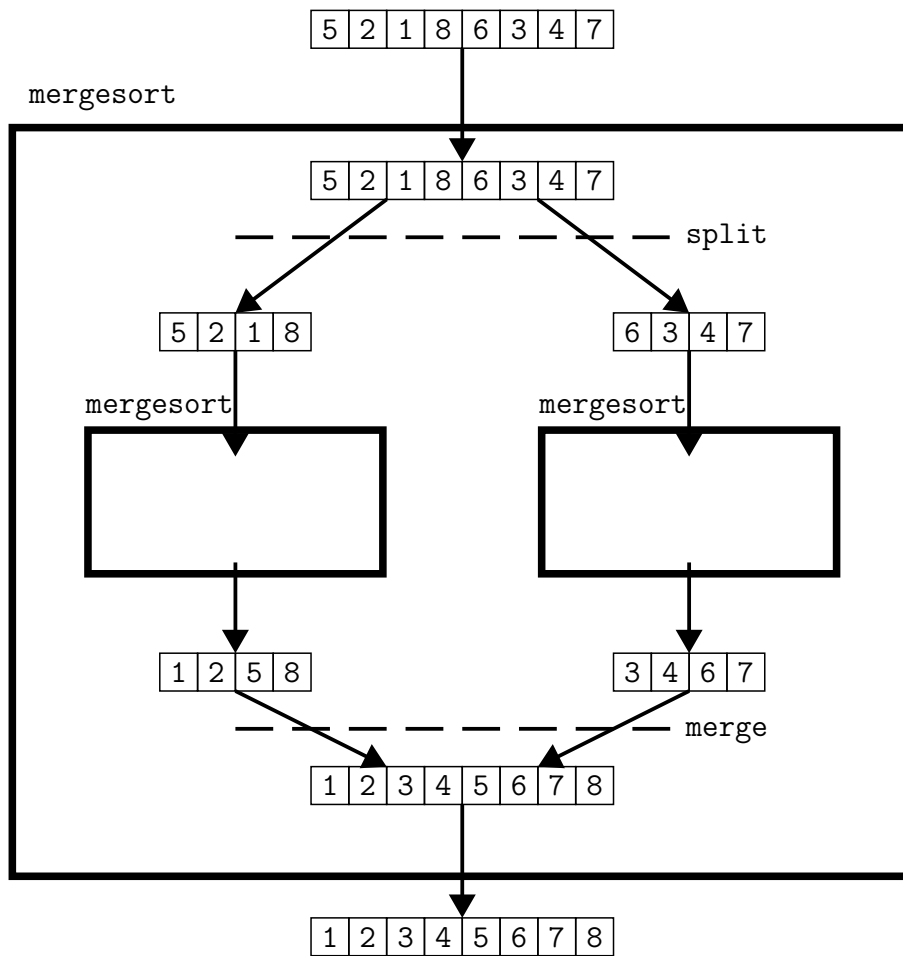
Also, mergesort is more important than just sorting since there is a general concept, an algorithm design technique, behind mergesort. This technique is called **divide-and-conquer** (DAC) and appears very frequently in the design of algorithms. In the case of mergesort,

divide-and-conquer

- Divide means dividing the problem into smaller subproblems, i.e., “instead of sorting the whole array, I'll sort the left half and then I'll sort the right half”.
- Conquer means combine the subsolutions into a big solution, i.e., “I'll merge the two sorted subarrays to sort the original larger array”.

(Some people explain the above in three steps: divide, conquer, combine.) This will be covered in great detail in CISS358. So mergesort, besides being a very important algorithm, embodies two very important techniques in designing algorithms: divide-and-conquer and merge.

Here's a diagram:



I'm not going to analyze the asymptotic runtime of this version of mergesort since the one in the next section is going to be better and is the right version.

Exercise 103.5.1. Write a `split` function that takes array `x` and puts the first half of the values of `x` into array `left` and the second half the values in array `right`. If the length of `x` is odd, then the length of `left` is one smaller than the length of `right`. Here's the C++ prototype:

```
void split(int x[], int size_x,
           int left[], int & size_left,
           int right[], int & size_right);
```

When you are done, convert that to a template. Here's a version that uses vectors:

```
template < typename T >
void split(const std::vector< T > & x,
           std::vector< T > & left,
           std::vector< T > & right);
```

debug:
exercises/mergesort-
0/question.tex

([Go to solution](#), page 4563)



Exercise 103.5.2. Implement a merge function that takes two sorted arrays and put the values, sorted, into another array. Here's a version that uses arrays:

debug:
exercises/mergesort-
1/question.tex

```
void merge(int ret[], int & size_ret,
           int left[], int & size_left,
           int right[], int & size_right);
```

(After you are done, you should change that to a template version.) Here's a version that uses vectors:

```
template < typename T >
std::vector< T > merge(const std::vector< T > & left,
                     const std::vector< T > & right);
```

([Go to solution](#), page 4564)



Exercise 103.5.3.

debug:
exercises/mergesort-
2/question.tex

(a) Complete the mergesort function. Start with this prototype:

```
void mergesort(int x[], int size_x);
```

You only need to sort arrays of size 1000. After you are done, change that to a template:

```
template < typename T >
void mergesort(T x[], int size_x);
```

Then use vectors:

```
template < typename T >
void mergesort(std::vector< T > & x);
```

(b) Assume the prototypes for mergesort functions are

```
template < typename T >
void split(const std::vector< T > & x,
           std::vector< T > & left,
           std::vector< T > & right);

template < typename T >
std::vector< T > merge(const std::vector< T > & left,
                    const std::vector< T > & right);

template < typename T >
std::vector< T > mergesort(std::vector< T > & x);
```

and assume the (asymptotic) runtime for memory allocation of an array of size

n is $O(n)$, what is the (asymptotic) runtime of this mergesort? (For C++, when you ask for an array from the heap, the amount of time for memory request depends on the algorithm used for memory management. But once the array is allocated, the standard practice for C++ is to fill the array with zeroes. So the runtime is at least n (asymptotically), i.e., $\Omega(n)$.) ([Go to solution](#), page [4565](#)) \square

Solutions

Solution to Exercise [103.5.1](#).

Solution not provided.

debug:
exercises/mergesort-
0/answer.tex

Solution to Exercise [103.5.2](#).

Solution not provided.

debug:
exercises/mergesort-
1/answer.tex

Solution to Exercise [103.5.3](#).

Solution not provided.

debug:
exercises/mergesort-
2/answer.tex

103.6 Mergesort ... 2nd Version debug: mergesort2.tex

Now that you understand the basic idea behind mergesort, let's move on to improve the memory usage of the previous mergesort.

The idea is similar to what I did for insertion sort. I'll try as much as possible to work inplace.

When I call mergesort, besides sending in the array, I'll send in two index values to indicate which subarray this mergesort should sort. Calling `mergesort(x, 0, 10)` means I want mergesort to sort `x[0..9]`. Each recursive mergesort function call will be working on some subarray of the *same* array `x`.

In the previous section, when given an array `x`, I create two arrays, `left` and `right`, and then distribute the value of `x` into `left` and `right`. Now split becomes very easy – since I'm working inplace, there are no extra `left` and `right` new arrays. For `mergesort(x, 0, 10)`, `left` is the subarray `x[0..4]` and `right` is the subarray `x[5..9]`.

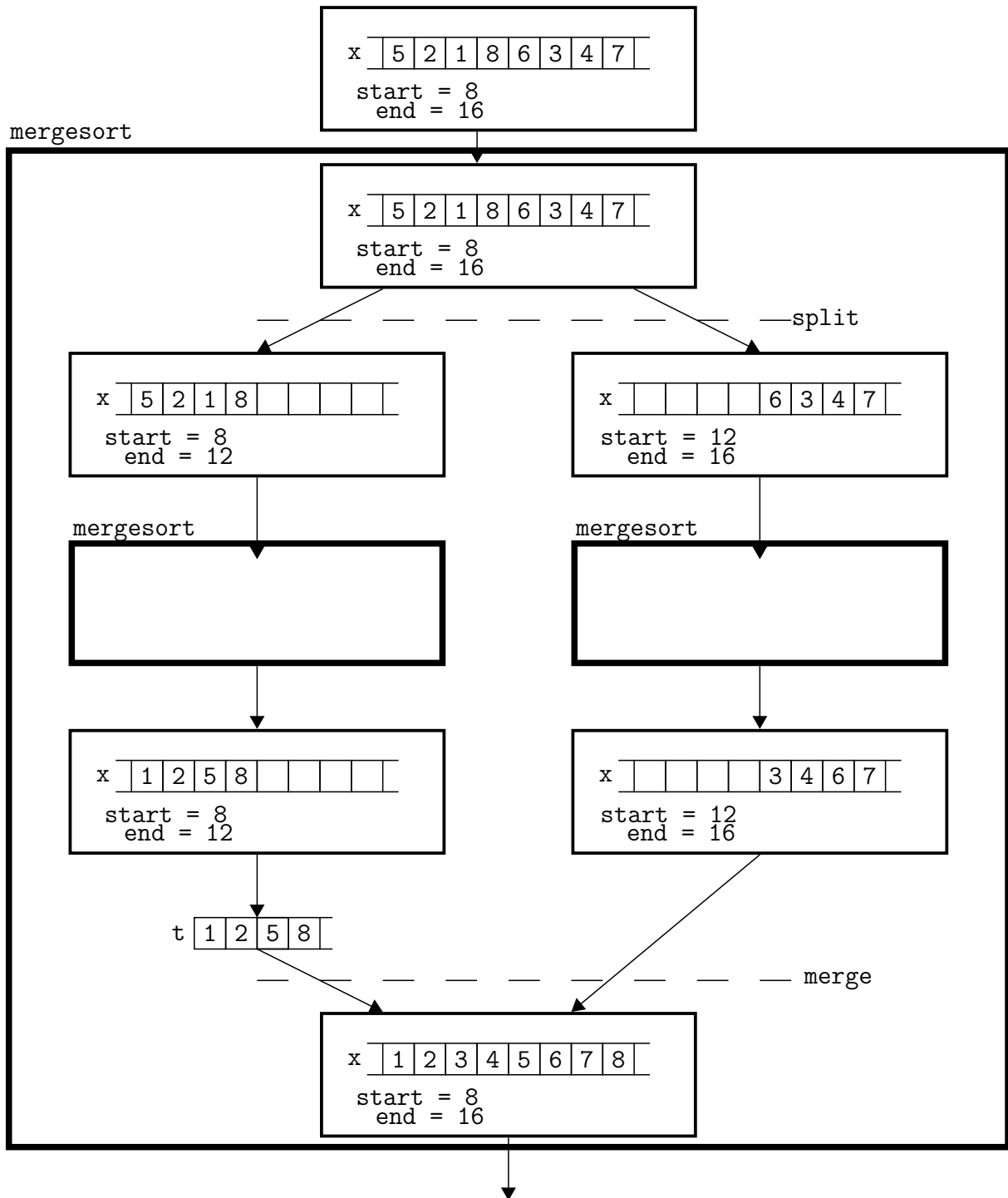
As for merge, when I merge say `x[0..4]` and `x[5..9]` into `x[0..9]`, there's going to be some problem. If all the values of `x[5..9]` are smaller than `x[0]`, then the values of `x[5..9]` should be copied to `x[0..4]`. Therefore I need to move the values of `x[0..4]` to a temporary array `t` of size 4, half of the size of the original subarray `x[0..9]`. So mergesort cannot be completely inplace.

Let's think about this temporary array. If the first function call is `mergesort(x, 0, 8)`, i.e., you want to sort an array of size 8, note that tallest stack of function calls has a size of 4. These 4 function calls are each working on an array of size 8, 4, 2, 1. The last function call is the base case and does not involve merging. Therefore the amount of memory usage for the temporary arrays for all these functions are temporary arrays of size 4, 2, 1. If the size of the array is 2^{10} , then the amount of memory is equivalent to an array of size $2^9 + 2^8 + 2^7 + \dots + 1 = 2^{10} - 1$. If you are using vectors, that's the amount. If you are using static arrays, you cannot vary the size of your array – it's fixed. If you believe the largest array to sort has size 2^{10} , you'll need to hardcode the declaration of your temporary array with size 2^9 in your mergesort function. Then the maximum amount of memory is at most $2^9 + 2^9 + 2^9 + \dots + 2^9 = 10 \cdot 2^9$ times the amount of memory of an element of your array.

But if you think about it very carefully, the temporary array `t` can be *shared* by all the mergesort function calls. Right? So if you are sorting an array `x` of size `n`, you create a temporary array of size `n/2` and pass `t` to the mergesort

function. This mergesort function call will also pass \mathfrak{t} during its two recursive function calls. When the two recursive function calls return, the original mergesort function can use the \mathfrak{t} that was already used by the two recursive function calls. So for sorting an array of size n , we only need *one* temporary array of size $n/2$!

Here's a diagram:



The algorithm for merge is easy and is similar to the merge in the previous section.

Mergesort is then

```

ALGORITHM: mergesort
INPUT: x          -- array
      start, end  -- x[start..end - 1] is to be sorted
      t          -- temporary array for merge
n = end - start
if n == 0 or n == 1:
    return
else:
    let mid = start + (end - start) / 2
    mergesort(x, start, mid, t)
    mergesort(x, mid, end, t)
    merge(x, start, mid, end, t)

```

Before I talk about MERGESORT, I'll talk about MERGE. As I mentioned earlier, the MERGESORT uses MERGE.

Exercise 103.6.1. (a) Implement a merge function. Here's the prototype:

```

template < typename T >
void merge(std::vector< T > & x, int start0, int end0, int start1, end1);

```

debug:
exercises/mergesort-
3/question.tex

where you are trying to merge $x[start0..end0 - 1]$ and $x[start1..end1 - 1]$. But these two subarrays are next to each other: $end0$ is actually $start1$. So

```

template < typename T >
void merge(std::vector< T > & x, int start0, int start1, int end1);

```

is enough. Also, don't forget you need to pass in the temporary t . So the right prototype is

```

template < typename T >
void merge(std::vector< T > & x, int start0, int start1, int end1,
          std::vector< T > & t);

```

Test it thoroughly.

(b) Once you are done with the above, you should also implement one that uses pointers. Here's the prototype:

```

template < typename T >
void merge(T * start0, T * start1, T * end1, T * t);

```

([Go to solution](#), page 4574)

□

Exercise 103.6.2. (a) Implement mergesort where the prototype is

```

template < typename T >
void mergesort(std::vector< T > & x, int start, int end
              std::vector< T > & t);

```

debug:
exercises/mergesort-
4/question.tex

But the user has to pass in a `t`. To make your mergesort user-friendly, do this instead: Call the above

```
template < typename T >
void mergesort_(std::vector< T > & x, int start, int end
               std::vector< T > & t);
```

And your mergesort is

```
template < typename T >
void mergesort(std::vector< T > & x)
{
    int n = x.size();
    std::vector< T > t(n / 2);
    mergesort_(x, 0, n, t);
}
```

(b) Of course you should also implement a version that uses pointers:

```
template < typename T >
void mergesort(T * start, T * end);
```

([Go to solution](#), page 4575)

□

Now let's analyze the time complexity and space complexity of MERGESORT. Space complexity is easy:

$$\text{SPACE}(n) = O(n)$$

(The temporary variables use up $O(n)$ space and the maximum number of active recursive function calls is $\lg n$ which means that the local variables used by all recursive function calls not including `t` is constant $\cdot \lg n$. So the space complexity is $O(n + c \lg n) = O(n)$. For the mergesort of the previous section, *each* function call has it's own temporary array. That means that the space complexity of mergesort from the previous section is $O(n \lg n)$.)

The runtime of the split function is

$$T_{\text{SPLIT}}(n) = O(1)$$

(In the case of the mergesort from the previous section, I actually have to create separate arrays for `left` and `right`. In this case the time complexity is $O(n)$.)

Now for the time complexity. The runtime of merge is

$$T_{\text{MERGE}}(n) = O(n)$$

(The first while-loop in merge will run $n/2$ times. Outside the first while-loop, the loop on `left` will run at most n times and the loop on `right` will run at most n times. Before doing the merge, I have to copy the `left` to `t`. That's a loop that takes $n/2$ iterations.)

Ignoring the time spent allocating memory for `t`, the time complexity is

$$\begin{aligned} T_{\text{MERGESORT}}(n) &= T_{\text{SPLIT}}(n) \\ &\quad + T_{\text{MERGESORT}}(\lfloor n/2 \rfloor) + T_{\text{MERGESORT}}(n - \lfloor n/2 \rfloor) \\ &\quad + T_{\text{MERGE}}(n) + A \end{aligned}$$

Since $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$, the runtime is

$$\begin{aligned} T_{\text{MERGESORT}}(n) &= T_{\text{SPLIT}}(n) \\ &\quad + T_{\text{MERGESORT}}(\lfloor n/2 \rfloor) + T_{\text{MERGESORT}}(\lceil n/2 \rceil) \\ &\quad + T_{\text{MERGE}}(n) + A \end{aligned}$$

Hence the runtime $T_{\text{MERGESORT}}(n)$ is of the form

$$T(n) = \begin{cases} A & \text{if } n = 0, 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + Bn + C & \text{otherwise} \end{cases}$$

Ignoring floors and ceilings, the runtime function looks like this:

$$T(n) = 2T(n/2) + Bn + C$$

and this is the case if n is even. Let me try to bring this to a closed form.

If I make the substitution $n = 2^m$, I get

$$\begin{aligned} T(2^m) &= 2T(2^{m-1}) + 2^m B + C \\ &= 2(2T(2^{m-2}) + 2^{m-1} B + C) + 2^m B + C \\ &= 2^2 T(2^{m-2}) + 2 \cdot 2^m B + (1 + 2)C \\ &= 2^2 (2T(2^{m-3}) + 2^{m-2} B + C) + 2 \cdot 2^m B + (1 + 2)C \\ &= 2^3 T(2^{m-3}) + 3 \cdot 2^m B + (1 + 2 + 2^2)C \end{aligned}$$

At this point you see that you will get

$$\begin{aligned} T(2^m) &= 2^m T(2^0) + m \cdot 2^m B + (1 + 2 + \cdots + 2^{m-1})C \\ &= 2^m T(2^0) + m \cdot 2^m B + (2^m - 1)C \end{aligned}$$

Replacing m by n using $n = 2^m$, hence $m = \lg n$, you get

$$\begin{aligned}T(n) &= nT(1) + (\lg n) \cdot nB + (n - 1)C \\&= B(n \lg n) + (C + T(1))n - C \\&= O(n \lg n)\end{aligned}$$

assuming $B \neq 0$ of course. Remember that the time spent allocating memory for \mathfrak{t} is not included in the above. At this point if I want to include that, assuming the time spent allocating an array of size n is $O(n)$, then the above $T(n)$ is still the same.

AHA! This is better than bubblesort which is $O(n^2)$.

Note that the above is not a proof!!! All it says is that *if* n is a power of 2, then

$$T(n) = nT(1) + n \lg n \cdot B + (n - 1)C$$

There's no reason to believe that above is still true if n is not a power of 2. Later I'll show you that the computations can be easily repaired to take into account the case where n is not a power of 2 and the end result is that the big-O is actually as stated above. (See CISS358.)

Note that for mergesort, each recursion part of mergesort does not have a branching that allows mergesort to terminate early. So in fact the time complexity is not only $O(n \lg n)$, it's actually

$$T(n) = \Theta(n \lg n)$$

So the best time, average time, worst time is $\Theta(n \lg n)$.

Exercise 103.6.3. Using the method above, try to compute big-O of $T(n)$ for the following cases.

debug:
exercises/mergesort-
5/question.tex

- (a) $T(n) = 3T(n/3) + Bn + C$
- (b) $T(n) = 5T(n/3) + Bn + C$
- (c) $T(n) = 2T(n/2) + C$
- (d) $T(n) = 2T(n/2) + Bn^2 + Cn + D$
- (e) $T(n) = 2T(n/2) + Bn \lg n + C$

(The $T(n)$ in (a) is the runtime of a mergesort that splits an array into three parts instead of two. The $T(n)$ in (c) is the runtime of the many easy divide-and-conquer recursions in the recursion chapter of CISS245.) ([Go to solution](#), page [4576](#)) \square

Exercise 103.6.4. Is mergesort inplace? stable? ([Go to solution](#), page 4577) ☐

debug:
exercises/mergesort-
6/question.tex

Exercise 103.6.5. For the case of a sorted array, what is one possible optimization to mergesort if you are only allowed to change the merge function? And if the array is sorted, what is the runtime? ([Go to solution](#), page 4578) ☐

debug:
exercises/mergesort-
optimization-sorted-
array/question.tex

Exercise 103.6.6. Design a mergesort that is optimized for space complexity and that uses the least amount of memory for the case when the array is “almost sorted”. (Hint: Avoid the work done in merge if possible. Perform memory allocation for the temporary array only when necessary and only request the amount that is needed.) ([Go to solution](#), page 4579) ☐

debug:
exercises/mergesort-
7/question.tex

Exercise 103.6.7. The merge requires copying the `left` to an temporary array. Think of a way to avoid this copying operation. (This one is tricky.) ([Go to solution](#), page 4580) ☐

debug:
exercises/mergesort-
8/question.tex

The version of mergesort described in this section (and previous) is internal. However mergesort can be external. In fact historically, mergesort was first used primarily in sorting data stored on tape drives. You noticed that the subarrays which are merges are just next to each other. This is great for the tape drives (and disk-based harddrives). Otherwise a tape drive would have to spin quite a bit to get two subarrays which are far apart. (I’ve worked with tape drives in the late 80s. They are slow.) Mergesort on data stored externally is sometimes called external mergesort.

Solutions

Solution to Exercise [103.6.1](#).

Solution not provided.

debug:
exercises/mergesort-
3/answer.tex

Solution to Exercise [103.6.2](#).

Solution not provided.

debug:
exercises/mergesort-
4/answer.tex

Solution to Exercise [103.6.3](#).

Solution not provided.

debug:
exercises/mergesort-
5/answer.tex

Solution to Exercise [103.6.4](#).

Solution not provided.

debug:
exercises/mergesort-
6/answer.tex

Solution to Exercise [103.6.5](#).

In the merge function, if the last element of `Left` is \leq the first element of `Right`, return immediately.

debug:
exercises/mergesort-
optimization-sorted-
array/answer.tex

The best runtime for the sorted array case will then satisfy

$$T_b(n) = 2T_b(n/2) + A$$

where A is a constant. It can be shown that the runtime is then

$$T_b(n) = O(n)$$

Solution to Exercise [103.6.6](#).

Solution not provided.

debug:
exercises/mergesort-
7/answer.tex

Solution to Exercise [103.6.7](#).

Solution not provided.

debug:
exercises/mergesort-
8/answer.tex

103.7 Quicksort: introduction debug: quicksort-intro.tex

Now we come to one of the most powerful comparison-based sorting algorithms for arrays: the quicksort.

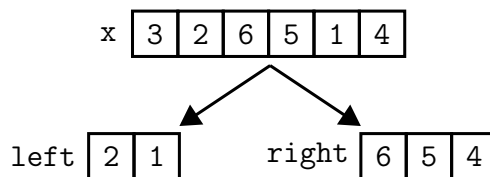
In this section, I will show you the main idea behind quicksort. The method here is not the real quicksort but it's easier to understand quickly so that with this overview of quicksort you can easily understand the real one (see next section). In particular for this section, I'll be creating lots of arrays. In other words, the quicksort in this section is *not* inplace.

Suppose

x

3	2	6	5	1	4
---	---	---	---	---	---

Using the first value $x[0] = 3$, we split x into two parts, **left** and **right** where **left** has values from x which are ≤ 3 **right** has values from x which are > 3 . One way to compute **left** is to scan x left-to-right and put values ≤ 3 into **left**. **right** is formed in the same way: scan x left-to-right and put value > 3 into **right**. 3 is called a **pivot**. This picture helps:



Joining up **left**, the pivot 3 as a array of one value $\{3\}$, and the **right** together

$\text{left} + \{3\} + \text{right}$

(I'll write + for array concatenation) and I get

2	1	3	6	5	4
---	---	---	---	---	---

where I've added two $|$'s in the array diagram above to show you the part of the array that is from **left**, the pivot, and **right**.

Note that partition also depends on which value in x is chosen to be the pivot.

Forming this new array from the original x is called **partitioning** of x . A **partition algorithm** is an algorithm that partitions x . The above partition algorithm is not efficient and should not be used in quicksort. In the next few

partitioning
partition algorithm

sections, I'll talk about more efficient partition algorithms.

A **pivot selection algorithm** is just a method to selection a value in the array to be the pivot (duh). There are also many pivot selection algorithms. For this section, I'll always select the first value of the array to be the pivot.

pivot selection
algorithm

Exercise 103.7.1. Write a function

```
int bad_pivot_index_selection(const std::vector< int > & x);
```

that always returns the index of the first value of `x`. This should take you 10 seconds! ([Go to solution](#), page 4587) ☐

debug:
exercises/quicksort-
12/question.tex

Exercise 103.7.2. Write a function

```
std::vector< int > bad_partition(const std::vector< int > & x,  
                                int pivot_index);
```

that implements the partition idea above. In your `bad_partition`, you'll have to form `left` and `right`. You'll also need to concatenate `left`, the pivot, and the `right`. If the size of `x` is 0 or 1, you simply return `x`. You have 5 minutes. ([Go to solution](#), page 4588) ☐

debug:
exercises/quicksort-
13/question.tex

There are many partition algorithms, each giving a different partition of the above array `x` for a pivot value of 3. Here are some valid partitions by pivot 3 using different partition algorithms:

1	2	3	6	5	4
---	---	---	---	---	---

2	1	3	5	6	4
---	---	---	---	---	---

1	2	3	4	6	5
---	---	---	---	---	---

Now let's look at our example above:

2	1	3	6	5	4
---	---	---	---	---	---

The array is clearly not sorted. But note the following *really important* facts about all partitions:

- The pivot, i.e. 3, is in the correct place. Future reorganization of the

array need not move the **pivot**.

- Although each individual value in **left** or **right** might not be in their right place yet, it is at least correct *with respect to the pivot*. For instance 2 is obviously not in the correct place yet, but at least it is to the left of the pivot 3.

See that?

So what should we do? Well ... we'll just perform quicksort on the **left** and **right** as well. They will have their own **left** and **right** and we do the same on them. Recurse! In other words, if I write

$$\text{quicksort}(x) = \text{left} + \{3\} + \text{right}$$

i.e., if quicksort is “pick a pivot and partition”, the resulting array is not sorted, ... but ... I'm going to recursively apply quicksort on **left** and **right**:

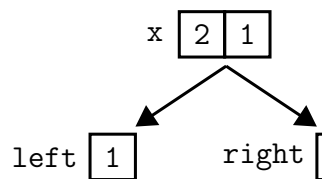
$$\text{quicksort}(x) = \text{quicksort}(\text{left}) + \{3\} + \text{quicksort}(\text{right})$$

In other words, for **left**, I pick a pivot value in **left** and perform partitioning on **left**. And I do the same for **right**. And I keep doing this, stopping the recursion when an array has size 0 or size 1. For this case, I simply return the array. This is the base case. In other words the base case of quicksort is

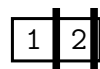
$$\text{quicksort}(x) = x \text{ if size of } x \text{ is } \leq 1$$

This is similar to mergesort. This makes sense. After all an array of size 0 or 1 is already sorted. Why bother doing anything to it?

For instance applying the above partition method to the subarray {2,1} with 2 chosen as the pivot, I get:

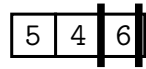


where the **right** in this case is an empty array (I'm using a skinny rectangle to denote an empty array). In *this* case, if we join up the **left**, the pivot as a single-value-array, and **right**, I get



When I call quicksort with $\{1\}$ I simply get $\{1\}$ since this is the base case. Same for $\{\}$.

And for $\{6, 5, 4\}$, suppose I choose 6 to be the pivot, I get



For $\{5, 4\}$, using 5 as the pivot, I get



Here's a complete trace (make sure you read this very slowly and very carefully):

```
quicksort({3,2,6,5,1,4})
= quicksort({2,1}) + {3} + quicksort({6,5,4}), pivot = 3
= (quicksort({1}) + {2} + quicksort({})) + {3} + quicksort({6,5,4}), pivot = 2
= ({1} + {2} + quicksort({})) + {3} + quicksort({6,5,4}), base case
= ({1} + {2} + {}) + {3} + quicksort({6,5,4}), base case
= {1,2} + {3} + quicksort({6,5,4}), concat
= {1,2} + {3} + (quicksort({5,4}) + {6} + quicksort({})), pivot = 6
= {1,2} + {3} + ((quicksort({4}) + {5} + quicksort({})) + {6} + quicksort({})), pivot = 5
= {1,2} + {3} + (({4} + {5} + quicksort({})) + {6} + quicksort({})), base case
= {1,2} + {3} + (({4} + {5} + {})) + {6} + quicksort({}), base case
= {1,2} + {3} + ({4,5} + {6} + quicksort({})), concat
= {1,2} + {3} + ({4,5} + {6} + {}), base case
= {1,2} + {3} + {4,5,6}, concat
= {1,2,3,4,5,6}, concat
```

You now have the general idea of quicksort. The next few things to do are as follows:

- (a) I'll make quicksort inplace.
- (b) I'll talk about better pivot selection algorithms.
- (c) I'll talk about better partition algorithms.

The first item is easy. Just like mergesort, I'll work with one array x throughout

and for each quicksort function call, I'll call quicksort with

`quicksort(x, start, end)`

which will sort `x[start..end - 1]`. The `left` and `right` won't be separate arrays, but will be part of `x[start..end - 1]`.

I'll talk about better pivot selection and partition algorithms in the next few sections.

Exercise 103.7.3. The following array

debug:
exercises/quicksort-
11/question.tex

`{10, 4, 21, 43, 30, 64, 98, 70}`

is obtained after one partition of quicksort. Which value was chosen to be the pivot? (There might be more than one possible pivot.)

([Go to solution](#), page 4589)

□

Exercise 103.7.4. Using the same functional notation as above, trace and compute `quicksort({1,5,4})` using the pivot selection method and partition method in this section. ([Go to solution](#), page 4590)

debug:
exercises/quicksort-
0/question.tex

□

Exercise 103.7.5.

debug:
exercises/quicksort-
1/question.tex

- (a) Using the same functional notation as above, trace and compute `quicksort({3,1,5,2,4})` using the pivot selection method and partition method in this section. How many function calls to quicksort were made? Draw the function call graph.
- (b) Next, do the same for `quicksort({1,2,3,4,5})`. How many function calls to quicksort were made? Draw the function call graph.
- (c) How many function calls to quicksort will be made when you call `quicksort({5,4,3,2,1})`?
- (d) Which of the above three cases seems to require fewer steps of computation?

([Go to solution](#), page 4591)

□

Exercise 103.7.6.

debug:
exercises/quicksort-
2/question.tex

- (a) What is the (asymptotic) runtime of the version of quicksort in this section when the array is `{1, 2, 3, ..., n - 1}`? What is the space

complexity?

- (b) Redo the above but now change the pivot selection so that you also pick the value in the “middle” value of the array. (“Middle” means if the array has an odd number of values pick the value in the middle of the array and if the array has an even number of values you can pick either the first or second value of the two values at the middle of the array.)

([Go to solution](#), page 4593)



Exercise 103.7.7. Implement the above version of quicksort, `bad_quicksort`, of this section. You can use `std::vector< int >` for your array.

debug:
exercises/quicksort-
7/question.tex

```
std::vector< int > quicksort(const std::vector< int > & x)
{
    if (x.size() <= 1)
    {
        // Base case
        return x;
    }
    else
    {
        // Recursive case
        int pivot_index = bad_pivot_index_selection(x);
        std::vector< int > left, right, ret;
        //compute left and right using the pivot_index
        left = quicksort(left);
        right = quicksort(right);
        //concatenate left, {pivot}, right to get ret
        return ret;
    }
}
```

([Go to solution](#), page 4594)



Solutions

Solution to Exercise [103.7.1](#).

Solution not provided.

debug:
exercises/quicksort-
12/answer.tex

Solution to Exercise [103.7.2](#).

Solution not provided.

debug:
exercises/quicksort-
13/answer.tex

Solution to Exercise [103.7.3](#).

If \mathbf{x} is the array and the pivot is $p = \mathbf{x}[i]$, then $\mathbf{x}[0..i-1]$ must have values $\leq p$ and $\mathbf{x}[i+1..7]$ must have values $> p$.

There are two possible pivots: 21 and 64.

debug:
exercises/quicksort-
11/answer.tex

Solution to Exercise [103.7.4](#).

debug:
exercises/quicksort-
0/answer.tex

```
quicksort({1,5,4})  
= quicksort({}) + {1} + quicksort({5,4}), pivot = 1  
= {} + {1} + quicksort({5,4}), base case  
= {} + {1} + (quicksort({4}) + {5} + quicksort({})), pivot = 5  
= {} + {1} + ({4} + {5} + quicksort({})), base case  
= {} + {1} + ({4} + {5} + {}), base case  
= {} + {1} + {4,5}, concat  
= {1,4,5}, concat
```

Solution to Exercise [103.7.5](#).

(a)

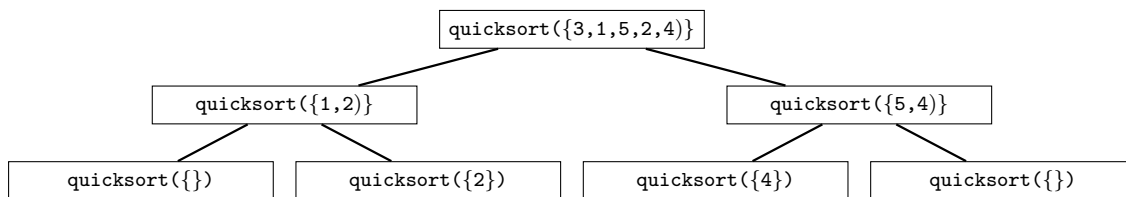
debug:
exercises/quick-
1/answer.tex

```

quicksort({3,1,5,2,4})
= quicksort({1,2}) + {3} + quicksort({5,4}), pivot = 3
= (quicksort({}) + {1} + quicksort({2})) + {3} + quicksort({5,4}), pivot = 1
= ({ } + {1} + quicksort({2})) + {3} + quicksort({5,4}), base case
= ({ } + {1} + {2}) + {3} + quicksort({5,4}), base case
= {1,2} + {3} + quicksort({5,4}), concat
= {1,2} + {3} + (quicksort({4}) + {5} + quicksort({})), pivot = 5
= {1,2} + {3} + ({4} + {5} + quicksort({})), base case
= {1,2} + {3} + ({4} + {5} + { } ), base case
= {1,2} + {3} + {4,5}, concat
= {1,2,3,4,5}, concat

```

7 function calls.



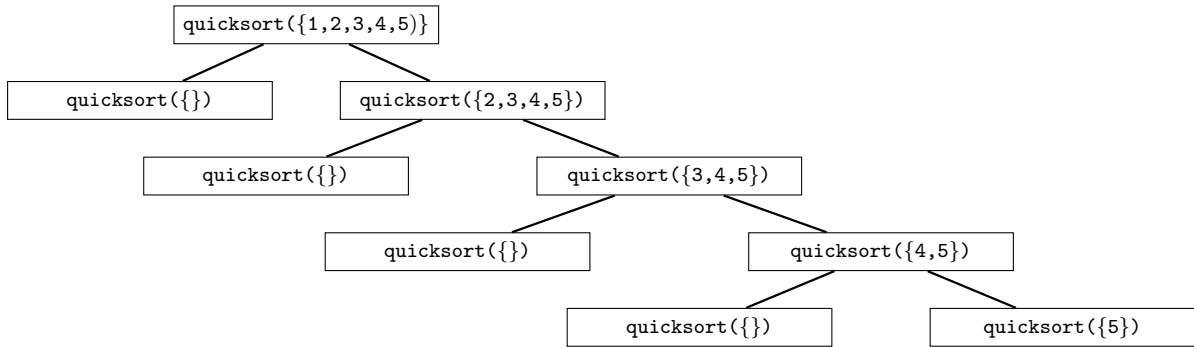
(b)

```

quicksort({1,2,3,4,5})
= quicksort({}) + {1} + quicksort({2,3,4,5}), pivot = 1
= {} + {1} + quicksort({2,3,4,5}), base case
= {} + {1} + (quicksort({}) + {2} + quicksort({3,4,5})), pivot = 2
= {} + {1} + ({} + {2} + quicksort({3,4,5})), base case
= {} + {1} + ({} + {2} + (quicksort({}) + {3} + quicksort({4,5}))), pivot = 3
= {} + {1} + ({} + {2} + ({} + {3} + quicksort({4,5}))), base case
= {} + {1} + ({} + {2} + ({} + {3} + (quicksort({}) + {4} + quicksort({5})))), pivot = 4
= {} + {1} + ({} + {2} + ({} + {3} + ({} + {4} + quicksort({5})))), base case
= {} + {1} + ({} + {2} + ({} + {3} + ({} + {4} + {5}))), base case
= {} + {1} + ({} + {2} + ({} + {3} + {4,5})), concat
= {} + {1} + ({} + {2} + {3,4,5}), concat
= {} + {1} + {2,3,4,5}, concat
= {1,2,3,4,5}, concat

```

9 function calls.



(c) 9 function calls.

(d) (a)

Solution to Exercise [103.7.6](#).

debug:
exercises/quicksort-
2/answer.tex

From the previous exercise, there are $n-1$ calls to the recursive branch of quicksort and n calls to the base case branch of quicksort. Each recursive branch of quicksort, not including the two recursive calls it makes, is determined by the (bad) partition algorithm and also concatenation, which altogether has a time complexity of $O(n)$. Therefore altogether the asymptotic runtime is

$$O(n^2)$$

Another way to derive the quadratic runtime is as follows. Let $T(n)$ be the time taken by quicksort for array $\{1, 2, 3, \dots, n-1\}$ including the recursive calls. Then

$$T(n) = An + B + T(n-1)$$

By using substitutions, I get

$$\begin{aligned} T(n) &= An + B + T(n-1) \\ &= An + B + A(n-1) + B + T(n-2) \\ &= An + B + A(n-1) + B + A(n-2) + B + T(n-3) \end{aligned}$$

etc. Therefore

$$T(n) = A(n + (n-1) + (n-2) + \dots + 1) + Bn + T(0)$$

Hence $T(n) = O(n^2)$.

The space used by quicksort when processing **left** and **right** of $\{1, 2, 3, \dots, n-1\}$ is $O(n)$. Counting all the memory usage in the longest function call chain, the memory usage is $A(n + (n-1) + \dots + 1) = O(n^2)$. \square

Solution to Exercise [103.7.7](#).

Solution not provided.

debug:
exercises/quicksort-
7/answer.tex

103.8 Quicksort: pivot selection debug: quicksort-pivot-selection.tex

In the previous section, I select the first value of the subarray to be sorted as the pivot selection algorithm. This is a standard “textbook pivot selection algorithm” that is not used in the real world. This algorithm is not the best and is in fact usually among the worst.

A better way is to select a random value in the subarray to be sorted. If the pivot is randomly chosen, then this version of quicksort is called **randomized quicksort**.

randomized quicksort

```
ALGORITHM: random_pivot_selection
INPUTS: x - array
        start, end - indices

return rand() % (end - start) + start
```

(For this algorithm, the array x is not used.)

The third way (generally considered the best) is the **median of three** method. You take the first, last and middle value. For instance if the following section of the array is to be sorted:

median of three

...,3,5,1,2,7,5,3,4,6,4,1,...

The first value is 3, the last value is 1, and the middle value is 5:

...,3,5,1,2,7,5,3,4,6,4,1,...

In this case, the median of 3,1,5 is 3 because after sorting 3,1,5, I have 1,3,5 and the value in the middle is 3. The three (sorted) values are then put back into the array:

...,1,5,1,2,7,3,3,4,6,4,5,...

And then the quicksort algorithm is carried out for this section of the array using the underlined 3 in the middle as pivot.

In the case when there is an odd number of values, there is one value in the middle. But if there's an even number of values, then there are two middle values. You pick the one on the left. For instance for

...,6,7,3,2,0,8,...

3,2 are the middle values and you pick 3 for the middle value.

The median of three is known to produce fewer comparisons: on the average

about $1.188n \lg n$. On the other hand using a randomly chosen value in the array on the average requires $1.386n \lg n$ comparisons. The benefit is that this method does not require generating random index value.

A variation of the median of three is to select three values at three *randomly* chosen distinct index values and then perform median of three on these three values. Another variation called **median of median of 3** is to do the same computation of median of three 3 times using 9 more or less equally spaced values in the array: you compute the median of three for the first 3 values, do the same for the next three, do the same for the last three. You then do median of three on the 3 medians. For instance suppose I have this array:

median of median of 3

2,5,1,2,7,5,3,4,6,4,1,7,2,5,8,9

I pick 9 values, as equally spaced out as possible:

2,5,1,2,7, 5,3,4,6,4,1,7,2,5,8,9,7

Then I do median of three on every 3 values:

median of 2, 1, 7 = 2

median of 3, 6, 1 = 3

median of 2, 8, 7 = 7

and I do median of three of the above three medians:

median of 2, 3, 7 = 3

The array becomes

1,5,2,2,7, 5,1,4,**3**,4,6,7,2,5,7,9,8

and the median is the value that is in bold. This method is also called **ninther**.

ninther

Exercise 103.8.1. What is the array

$\{5, 1, 2, 6, 8, 9, 6, 4, 2, 0, 7, 3\}$

after you performed median of three on it? If an array has size n , what is the index of the second value that will be considered in the median of three calculation? (Go to solution, page 4598) \square

debug:
exercises/median-of-
three-0/question.tex

Exercise 103.8.2. Write down the pseudocode for `median_of_three_pivot_selection`

debug:
exercises/median-of-
three-1/question.tex

Next, implement

```
int median_of_three_pivot_selection(int x[], int start, int end);
```

Test it thoroughly. Then implement this version:

```
template < typename T >
int median_of_three_pivot_selection(std::vector< T > & x,
                                   int start, int end);
```

and this version:

```
template < typename T >
T * median_of_three_pivot_selection(T * start, T * end);
```

([Go to solution](#), page [4599](#))

□

Solutions

Solution to Exercise [103.8.1](#).

After performing median of three, the array becomes

$$\{\underline{3}, 1, 2, 6, 8, \underline{5}, 6, 4, 2, 0, 7, \underline{9}\}$$

The index is $(n - 1)/2$.

□

debug:
exercises/median-of-
three-0/answer.tex

Solution to Exercise [103.8.2](#).

Solution not provided.

debug:
exercises/median-of-
three-1/answer.tex

103.9 Quicksort: a partition strategy that is reasonable, short, and completely wrong debug:

quicksort-wrong.tex


The next is to decide where exactly to put values in the `left` and the `right` chunks. In the above long example, I form the `left` chunk by simply scanning the given array left-to-right and put a value into `left` to the end of `left`. Same for `right`. Because I'm going to use the same array (to make quicksort inplace), the above does not work.

Let's try something just to get into the swing of things. Here's a naive way of getting an array partitioned. Suppose we have this array:

3	1	6	2	4
---	---	---	---	---

and suppose the pivot I choose is 2 at index 4. I now scan the array and put the values relative to the pivot ... and don't forget that I want to do this inplace, i.e., work on the array directly and not use a separate temporary array. Scanning left-to-right the only reasonable first value to process is 3. Clearly 3 must be on the right of the pivot 2. A reasonable thing to do is to swap 3 and 2:


3	1	6	2	4
---	---	---	---	---



2	1	6	3	4
---	---	---	---	---

Now the index of my pivot is 0. Next I swap 1 to get to the left of the pivot:

2	1	6	3	4
---	---	---	---	---

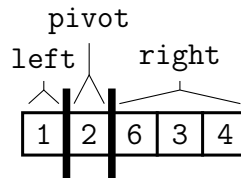


1	2	6	3	4
---	---	---	---	---

At this point, 1 and 3 are on the correct side of the pivot 2. The next index position to process has value 6 which is already on the correct side of the pivot so we don't do anything. The next value to process has value 3 which was actually already processed, but if we loop over the values left-to-right without knowing which one is processed, we would have to process this value anyway.

Since 3 is on the correct side we don't do anything. Same thing for 4. We're done.

Hmmm ... it looks like the strategy works. The end result is:



Exercise 103.9.1. Execute the above algorithm on this array with 2 as the pivot:

$\{3, 1, 0, 2, 6, 4\}$

debug:
exercises/quicksort-
8/question.tex

Does the partition procedure partition the above correctly? ([Go to solution](#), page 4602) ☐

Exercise 103.9.2. Execute the above algorithm on this array with 2 as the pivot:

$\{2, 0, 1, 3, 6, 4\}$

debug:
exercises/quicksort-
9/question.tex

([Go to solution](#), page 4603)

☐

Exercise 103.9.3. Execute the above algorithm and this array with 2 as the pivot:

$\{3, 1, 6, 0, 2, 4\}$

debug:
exercises/quicksort-
10/question.tex

Does the partition procedure really partition the above? ([Go to solution](#), page 4604) ☐

Now you see that our greedy and short-sighted approach fails. In the next few sections, I'll show you different strategies that actually work. The key idea is to *temporarily move the pivot out of the way*.

Solutions

Solution to Exercise [103.9.1](#).

Solution not provided.

debug:
exercises/quicksort-
8/answer.tex

Solution to Exercise [103.9.2](#).

Solution not provided.

debug:
exercises/quicksort-
9/answer.tex

Solution to Exercise [103.9.3](#).

Solution not provided.

debug:
exercises/quicksort-
10/answer.tex

103.10 Quicksort: pLRT partition method debug:

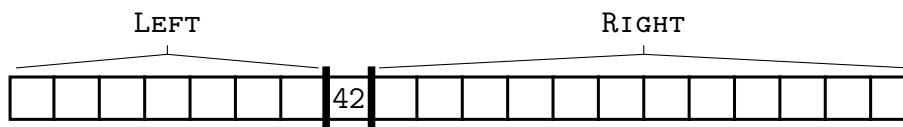
quicksort-plrt.tex

Let's talk about an efficient inplace partition algorithm. (There are many.)

Suppose this is my array x during quicksort:

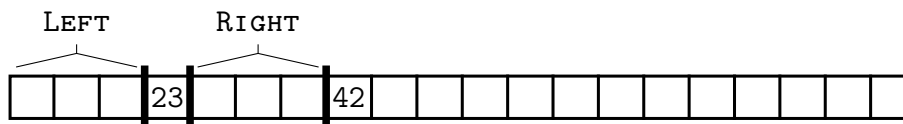


After applying my partition strategy, say with a pivot of 42, I might get this:



In other words, I want a partition algorithm where the LEFT and RIGHT is part of the original array x . This partition algorithm is inplace.

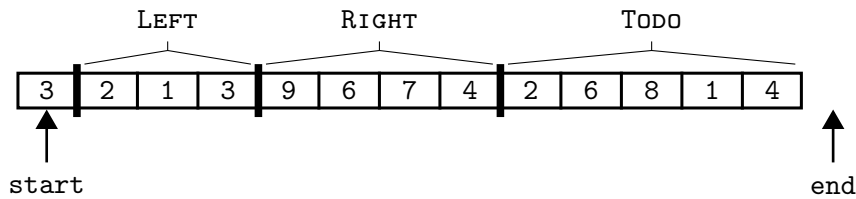
And like the bad partition algorithm, I then call quicksort a second time on LEFT (later I'll also call quicksort on RIGHT). In the second function call to quicksort on the LEFT, the partition algorithm might give me this where a pivot of 23 is used:



And of course recursion keeps going until base case is reached. This is what I want. Here's the idea of the new partition algorithm.

The above is the shape of my array after a partition is completed. *During* the algorithm, the subarray that I'm sorting is subdivided into 4 sections: pivot-LEFT-RIGHT-TODO. I'll call this partition algorithm pLRT. So that we are all the same page, unless otherwise stated, pLRT will be the default partition algorithm (for quizzes, assignments, tests).

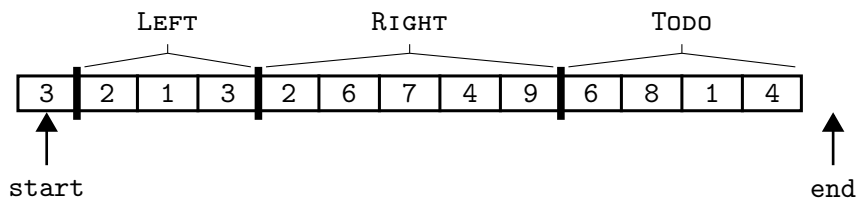
Here's a picture to help understand this partition strategy. First of all, after the pivot has been chosen, I swap the pivot with the first value of the subarray (i.e., I swap the pivot with $x[\text{start}]$). Here's an example of a snapshot during the execution of this partition where the pivot is 3 (the value at index `start`):



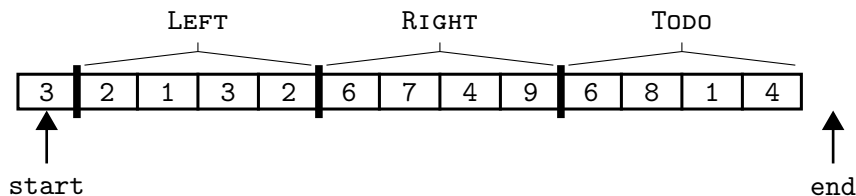
At this point, the pivot 3 is moved to the beginning of the subarray that the current quicksort function call is working on. The **LEFT** is {2, 1, 3}. All values in **LEFT** are \leq the pivot 3. The **RIGHT** is {9, 6, 7, 4}. All values in **RIGHT** are $>$ 3. The values not yet processed are in **TODO** which is {2, 6, 8, 1, 4}.

This is how the pLRT partition works.

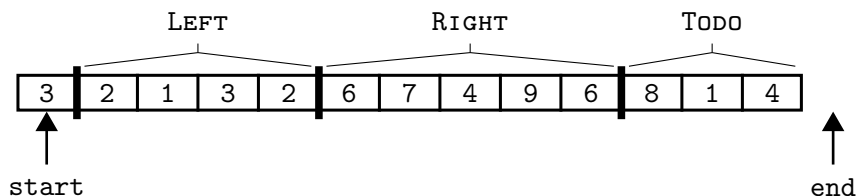
Look at the leftmost value of **TODO**: the next value to be processed is 2. 2 should go into **LEFT**. To do that, I swap 2 with the leftmost value of **RIGHT**:



and then move the dividing line | between **LEFT** and **RIGHT** to the right by one step:



The next value in **TODO** is 6, which should go into **RIGHT**. I'll just move the divider between the **RIGHT** and **TODO**:

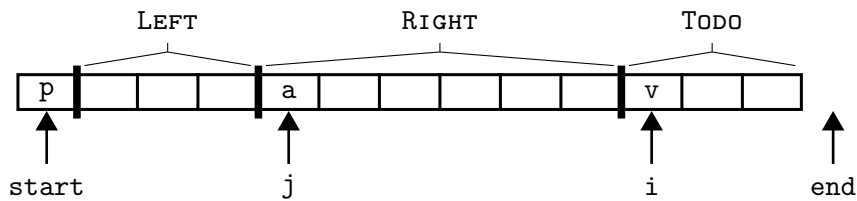


Once **TODO** is empty, I'll move the pivot (at index **start**) between **LEFT** and **RIGHT** with one swap, by swapping it with the rightmost value of **LEFT** if

LEFT is not empty. If LEFT is empty, I don't have to do anything. (Right?)

Let's analyze this algorithm in detail.

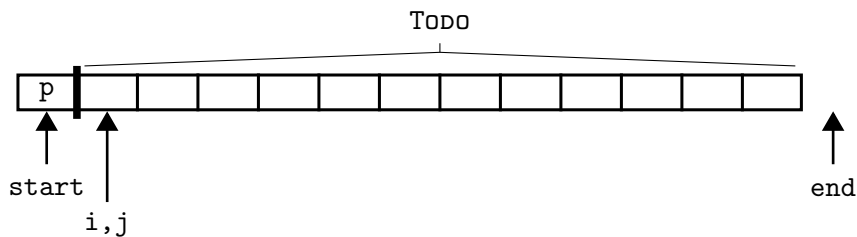
In terms of pseudocode, let i be the leftmost index of TODO and j be the leftmost index of RIGHT. I don't need the index of the first value of the LEFT since that's just $\text{start} + 1$. Let p be the value of the pivot.



INITIALIZATION. Of course at the beginning, i has to be $\text{start} + 1$. But where is j !?! Well the number of values in RIGHT is $i - j$. Initially, RIGHT is empty. Therefore $i = j$.

Also, the number of values in LEFT is $i - (\text{start} + 1)$. Initially, LEFT is empty and therefore $i = \text{start} + 1$. Hence initially

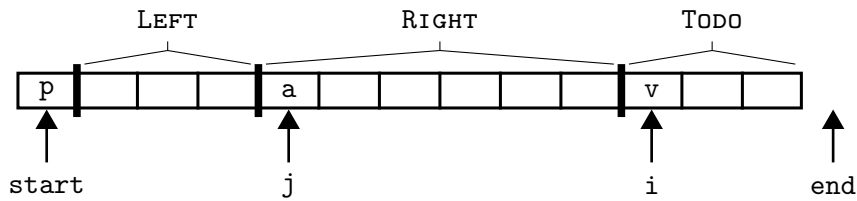
$$i = \text{start} + 1, j = i = \text{start} + 1$$



Let me also record the following very useful facts:

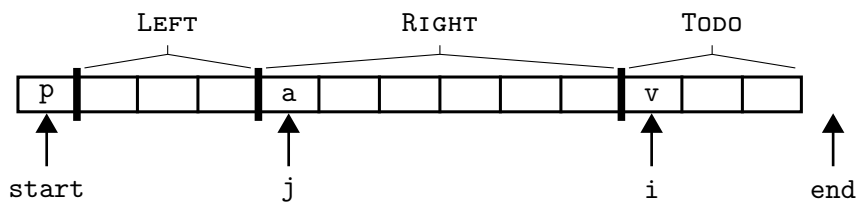
$$\begin{aligned} \text{size of LEFT} &= j - (\text{start} + 1) \\ \text{size of RIGHT} &= i - j \end{aligned}$$

Now let's see what happens in each iteration of the pLRT partition. Look at this picture:

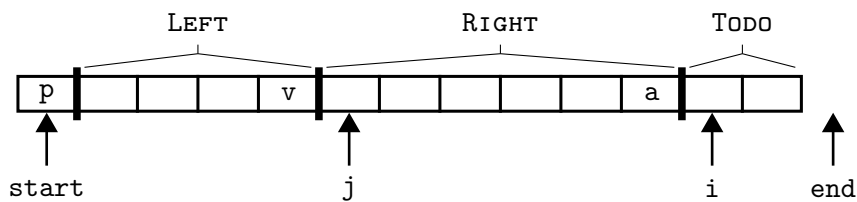


The next value to be processed (in TODO) is $x[i] = v$. There are two cases: either $v \leq p$ or $v > p$.

CASE $v \leq p$. In this case, v has to go into LEFT. I can do it this way:

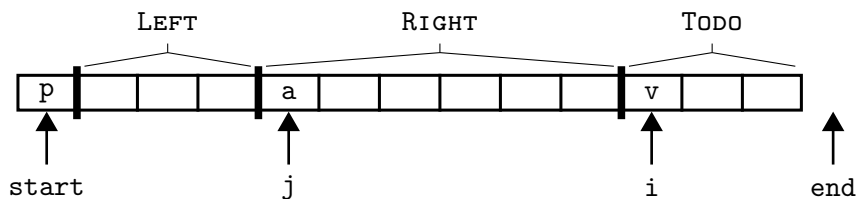


becomes

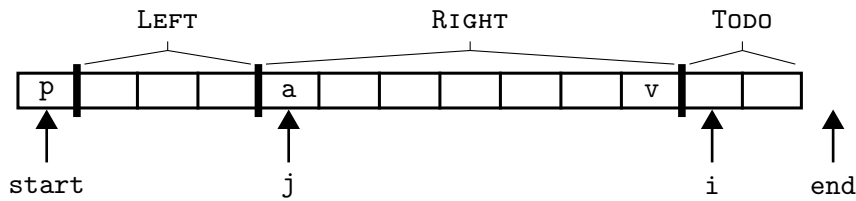


which means swapping $x[i]$ and $x[j]$ (the leftmost value of RIGHT) and then incrementing j . Of course I also have to increment i . But hang on, this assumes that RIGHT is nonempty. If RIGHT is empty, then I only increment j and i . (Correct?)

CASE $v > p$. Then v has to go into RIGHT:



becomes



which means simply incrementing `i`. (Is the logic still correct if `LEFT` is empty? if `RIGHT` is empty?)

Be carefully: algorithms must work for all scenarios. If you are using pictures to help derive the logic of your algorithm, make sure your pictures cover all cases. If you are in CISS350, you should know by now that you should be careful with corner cases. Did you consider cases such as when `LEFT` is empty? when `RIGHT` is empty? when both are empty? Etc. Make sure you completely analyze all cases.

TERMINATION. Clearly the loop ends when `TODO` is empty, i.e., when `i = end`. Furthermore it's clear that `i` increments by 1 in each iteration. So definitely I won't run into an infinite loop.

After `TODO` is completed, the last step of the partition algorithm is to place the pivot (at index `start`) between the `LEFT` and the `RIGHT`. The pivot can be swapped with the rightmost value of `LEFT` (if `LEFT` is not empty). If `LEFT` is empty, then this swap is not necessary. (Correct?)

Another thing to consider is this: Earlier on, our base case is when the subarray has size ≤ 1 . But wait a minute ... if you are using median of three for your pivot selection algorithm, you are assuming the subarray you are sorting has at least 3 values. Right? What should you do when the subarray has size 2? And furthermore if the size is 3, do you really want to do median of three and then call quicksort?

Now complete the algorithm for this partition strategy when given the index of the pivot. The algorithm should return the final index of the pivot.

Here's our partition algorithm:


```

ALGORITHM: pLRT_partition
INPUTS: x - array
        start, end - indices
        pivot_index - index position of pivot

swap x[start], x[pivot_index]
let p = x[start]

let j = start + 1
for i = start + 1, start + 2, start + 3, ..., end - 1:
    if x[i] <= p:
        if RIGHT is nonempty, i.e. i - j > 0:
            swap x[i], x[j]
            j = j + 1

if LEFT is nonempty, i.e. j - (start + 1) > 0:
    swap x[start], x[j - 1]
    return j - 1
else:
    return start

```

and assuming we are using median of three for pivot selection, quicksort is then

```

ALGORITHM: quicksort
INPUTS: x - array
        start, end - indices

let n = end - start (the size of the subarray to sort)

if n < 2:
    return
else if n <= 3:
    sort x[start..end - 1] by an unrolled insertionsort
else:
    pivot_index = median_of_three_pivot_selection(x, start, end)
    pivot_index = pLRT(x, start, end, pivot_index)
    quicksort(x, start, pivot_index)
    quicksort(x, pivot_index + 1, end)

```

Exercise 103.10.1. Execute one partition of

$$\{3, 9, 6, 4, 1, 0, 8, 5, 7, 2\}$$

using pLRT-partition for the following cases:

- (a) the pivot selected is 3
- (b) the pivot selected is 0

debug:
exercises/quicksort-
3/question.tex

- (c) the pivot selected is 9
- (d) median of three is used

([Go to solution](#), page 4617)

□

Exercise 103.10.2. For the array,

$\{3, 9, 6, 4, 1, 0, 8, 5, 7, 2\}$

perform quicksort using pLRT-partition and

- (a) Using first value for pivot selection
- (b) Using the median of three for pivot selection

([Go to solution](#), page 4618)

□

Exercise 103.10.3.

- (a) Trace one partition on the array $\{0, 1, 2, 3, 4, 5, 6, 7\}$ assuming you are using median of three and the pLRT-partition. This is the case of quicksort on an array that is already sorted.
- (b) Do the same for $\{7, 1, 2, 3, 4, 5, 6, 0\}$. This is an example of a case where the array is almost sorted.
- (c) Do the same for $\{4, 5, 6, 7, 0, 1, 2, 3\}$. This is an example of a case where the array contains sorted subarrays.
- (d) Do the same for $\{0, 2, 4, 6, 1, 3, 5, 7\}$.
- (e) Do the same for $\{0, 1, 0, 1, 0, 1, 0, 1\}$.
- (f) Do the same for $\{0, 0, 0, 0, 0, 0, 0, 0\}$.

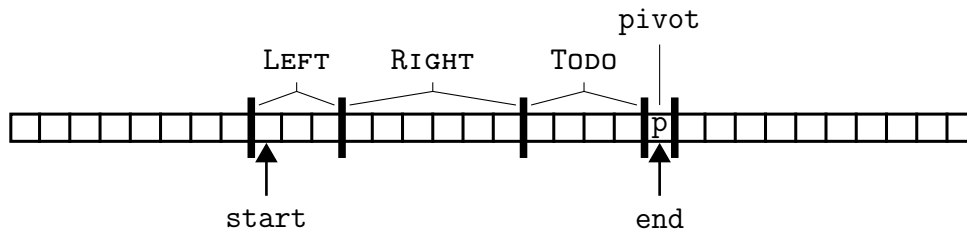
These are examples of test cases researchers in sorting algorithms worry about.

([Go to solution](#), page 4619)

□

Exercise 103.10.4. Write a LRTp-partition where the subarray to be sorted is organized as LEFT-RIGHT-TODO-pivot, placing the pivot at the right end. Write a quicksort that uses this partition.

debug:
exercises/quicksort-
LRTp/question.tex



([Go to solution](#), page 4620)

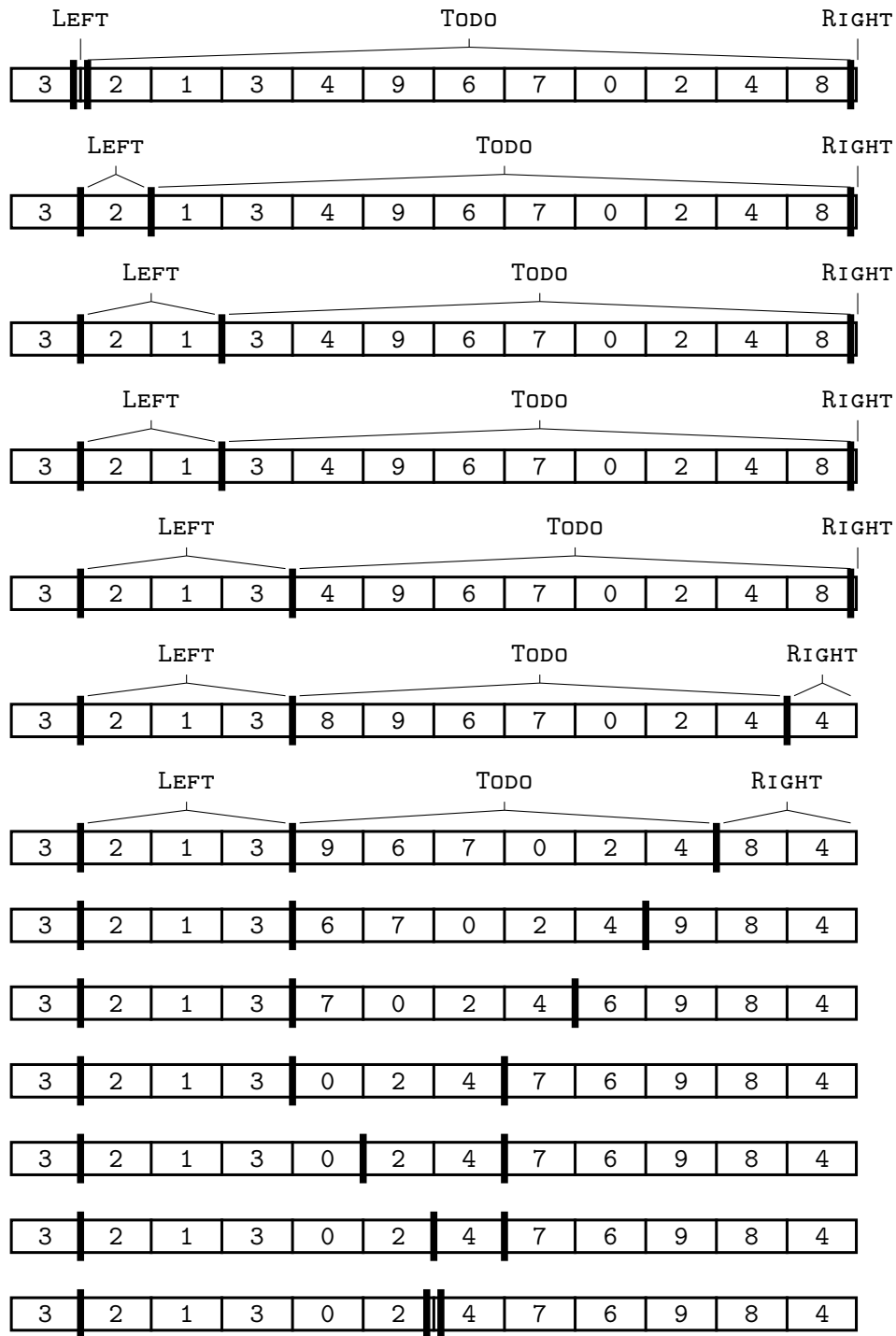
□

Exercise 103.10.5. Analyze and implement the pLTR partition and a quicksort that uses pLTR. Note that in this case `TODO` is in the middle. To make sure you understand what to do, the following is a trace of quicksort using pLTR where the pivot selection is “pick the first value”. Here’s an incomplete trace on this array `xs`:

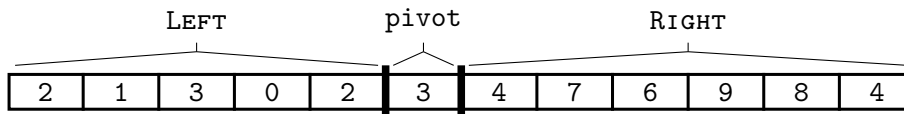
debug:
exercises/quicksort-
pLTR/question.tex

3	2	1	3	4	9	6	7	0	2	4	8
---	---	---	---	---	---	---	---	---	---	---	---

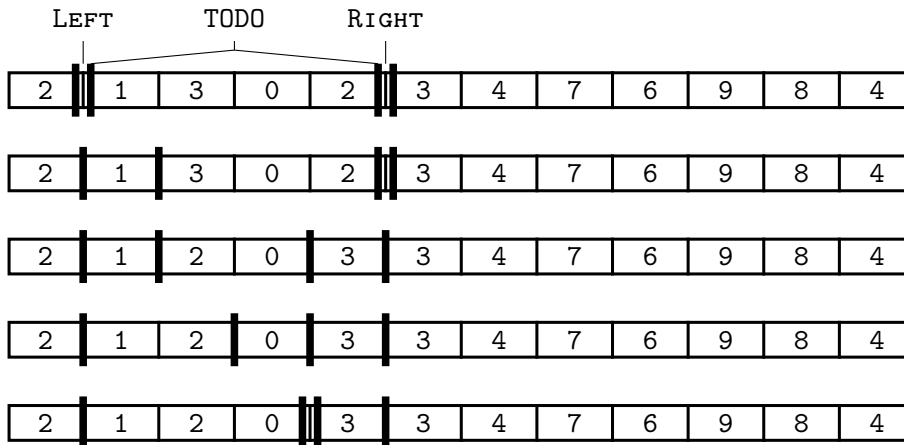
`quicksort(xs, 0, 12):`



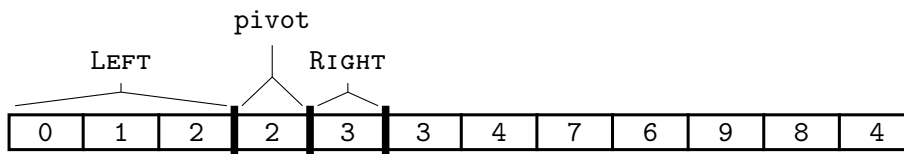
and swapping the pivot 3 with the rightmost value in LEFT (if it's not empty):



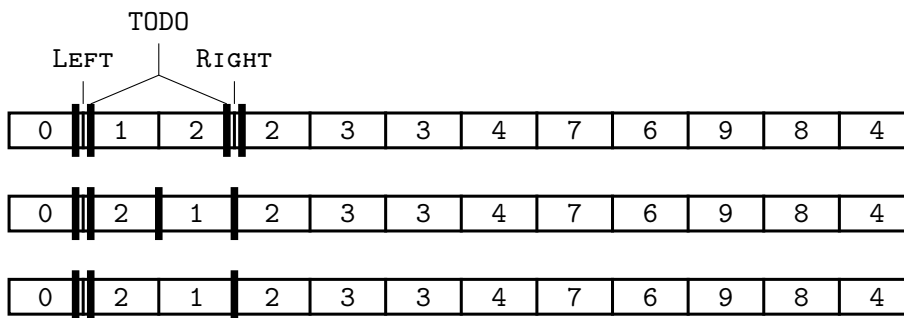
`quicksort(xs, 0, 5):`



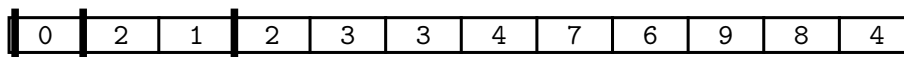
and swapping the pivot 2 with the rightmost value of LEFT (if it's not empty):



`quicksort(xs, 0, 3):`

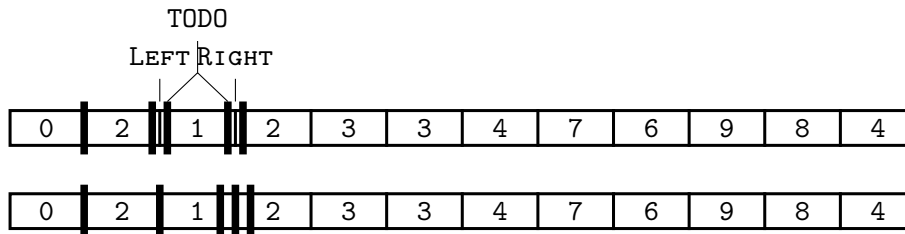


and swapping the pivot 0 is not necessary since LEFT is empty:

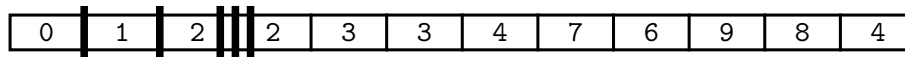


`quicksort(xs, 0, 0)`: Base case. Return right away.

`quicksort(xs, 1, 3)`:



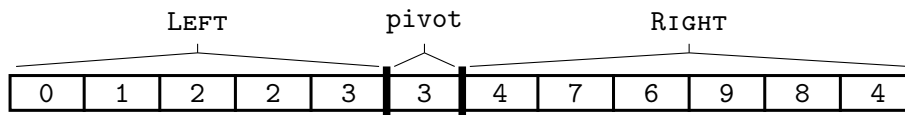
Swapping the pivot 2 with the rightmost value in the LEFT pile, we get



`quicksort(xs, 1, 2)`: Base case. Return right away.

`quicksort(xs, 3, 3)`: Base case. Return right away.

I'm now done partitioning `xs[0..4]` (the LEFT from the very first `quicksort` function call) down to base cases. So `xs[0..4]` is sorted:



I'll leave it to you to finish the other function calls. ([Go to solution](#), page 4621) ☐

Exercise 103.10.6. Write a psLRT-partition where the subarray to be sorted is organized as pivots-LEFT-RIGHT-TODO where repeats of the pivot are stored next to each other. ([Go to solution](#), page 4622) ☐

debug:
exercises/quicksort-
psLRT/question.tex

Exercise 103.10.7. Show that quicksort is unstable when

- Using median of three pivot selection on an array of size 3 and then performing the pLRT partition. (If an array has size 3, after median of three it would be sorted so you actually don't have to do partitioning on it.)
- Using first value pivot selection and pLRT partition

debug:
exercises/show-
quicksort-is-
unstable/question.tex

([Go to solution](#), page [4623](#))

□

Exercise 103.10.8. Which sorting algorithm (bubblesort, insertionsort, selectionsort, mergesort, quicksort) has the fastest asymptotic and wallclock time when sorting an array that is already sorted? Why? ([Go to solution](#), page [4624](#))

□

debug:
exercises/fastest-
comparison-based-
sorting-for-sorted-
array/question.tex

Solutions

Solution to Exercise [103.10.1](#).

debug:
exercises/quicksort-
3/answer.tex

(a)

$\{3|||9, 6, 4, 1, 0, 8, 5, 7, 2\}$
 $\{3|9|6, 4, 1, 0, 8, 5, 7, 2\}$
 $\{3|9, 6|4, 1, 0, 8, 5, 7, 2\}$
 $\{3|9, 6, 4|1, 0, 8, 5, 7, 2\}$
 $\{3|1|6, 4, 9|0, 8, 5, 7, 2\}$
 $\{3|1, 0|4, 9, 6|8, 5, 7, 2\}$
 $\{3|1, 0|4, 9, 6, 8|5, 7, 2\}$
 $\{3|1, 0|4, 9, 6, 8, 5|7, 2\}$
 $\{3|1, 0|4, 9, 6, 8, 5, 7|2\}$
 $\{3|1, 0, 2|9, 6, 8, 5, 7, 4\}$
 $\{2, 1, 0|3|9, 6, 8, 5, 7, 4\}$

i.e.,

$\{3, 9, 6, 4, 1, 0, 8, 5, 7, 2\}$
 $\{3, 9, 6, 4, 1, 0, 8, 5, 7, 2\}$
 $\{3, 9, 6, 4, 1, 0, 8, 5, 7, 2\}$
 $\{3, 9, 6, 4, 1, 0, 8, 5, 7, 2\}$
 $\{3, 1, 6, 4, 9, 0, 8, 5, 7, 2\}$
 $\{3, 1, 0, 4, 9, 6, 8, 5, 7, 2\}$
 $\{3, 1, 0, 4, 9, 6, 8, 5, 7, 2\}$
 $\{3, 1, 0, 4, 9, 6, 8, 5, 7, 2\}$
 $\{3, 1, 0, 4, 9, 6, 8, 5, 7, 2\}$
 $\{3, 1, 0, 2, 9, 6, 8, 5, 7, 4\}$
 $\{2, 1, 0, 3, 9, 6, 8, 5, 7, 4\}$

(b),(c),(d): Solutions not provided.

Solution to Exercise [103.10.2](#).

Solution not provided.

debug:
exercises/quicksort-
4/answer.tex

Solution to Exercise [103.10.3](#).

Solution not provided.

debug:
exercises/quicksort-
5/answer.tex

Solution to Exercise [103.10.4](#).

Solution not provided.

debug:
exercises/quicksort-
LRTp/answer.tex

Solution to Exercise [103.10.5](#).

Solution not provided.

debug:
exercises/quicksort-
pLTR/answer.tex

Solution to Exercise [103.10.6](#).

Solution not provided.

debug:
exercises/quicksort-
psLRT/answer.tex

Solution to Exercise [103.10.7](#).

debug:
exercises/show-
quicksort-is-
unstable/answer.tex

(a) Let the array be $\{0,0,0\}$. To differentiate the three 0s, we will write $\{0a,0b,0c\}$. The median is $0b$.

- Moving the pivot (i.e., the median) to the beginning of the array gives us $\{0b,0a,0c\}$
- After processing the values in the TODO list, we get $\{0b,0a,0c\}$. Note that LEFT is $\{0a,0c\}$ and RIGHT is empty.
- Moving the pivot to the rightmost position of LEFT, we get $\{0c,0a,0b\}$.

Hence quicksort using median-of-three pivot selection and the pLRT partition is unstable.

(Note: Again if an array has size 3, after median of three, there's no need to partition it. The above is just to show that if you do partition it, the whole process is unstable. This means that an example such as $0,0,0,0$ will be unstable after median of 3 and partitioning since the three 0s that took part in the median of 3 will undergo an unstable permutation.)

(b) Let the array be $\{0,0,0\}$. To differentiate the three 0s, we will write $\{0a,0b,0c\}$. The pivot is $0a$.

- The pivot $0a$ is already in the right place.
- After processing the values in the TODO list, we get $\{0a,0b,0c\}$. Note that LEFT is $\{0b,0c\}$ and RIGHT is empty.
- Moving the pivot to the rightmost position of LEFT, we get $\{0c,0b,0a\}$.

Hence quicksort using median-of-three pivot selection and the pLRT partition is unstable.

Solution to Exercise [103.10.8](#).

Insertion sort is the fastest. If the array is sorted, the inner loop executes its body 0 times since the element to process is in its right place. This means that the runtime is $O(n)$. Bubblesort and selection sort have runtime of $O(n^2)$ regardless, i.e., the best, average, worst runtime is the same. The asymptotic runtime of mergesort for best, average, worst are the same, i.e., $O(n \lg n)$. Quicksort runs with best and average runtime of $O(n \lg n)$.

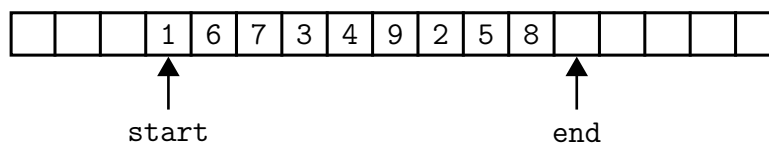
debug:
exercises/fastest-
comparison-based-
sorting-for-sorted-
array/answer.tex

103.11 Quicksort: Hoare's partition strategy debug:

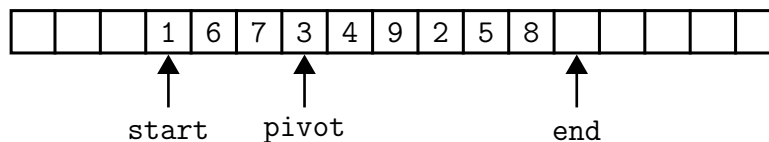
quicksort-hoare.tex

Here's the original quicksort partition strategy. This was invented by Hoare around 1960. In fact this was the first partition strategy since Hoare was the one who discovered quicksort.

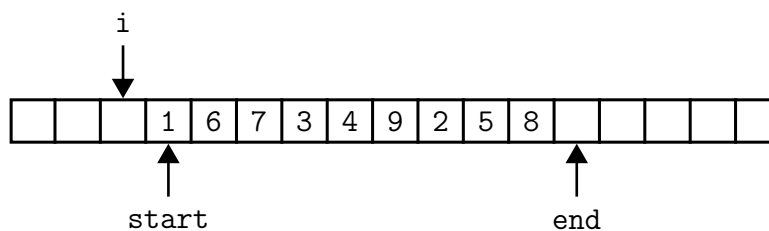
There's a slight difference between Hoare's version and our earlier version: For Hoare's version, the section of array to be sorted is split into two parts (by the pivot), but the pivot need not be exactly in the middle.



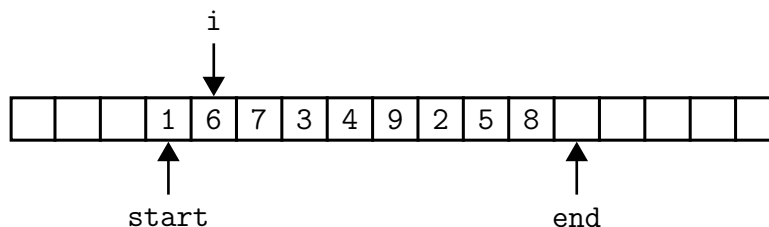
Suppose we choose 3 as the pivot value:



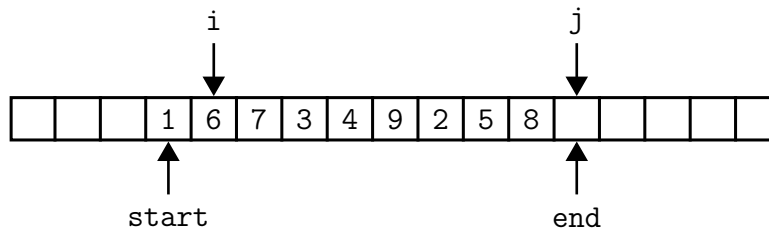
Starting with an index value that is one smaller than the smallest index value in the subarray, i.e. `start - 1`,



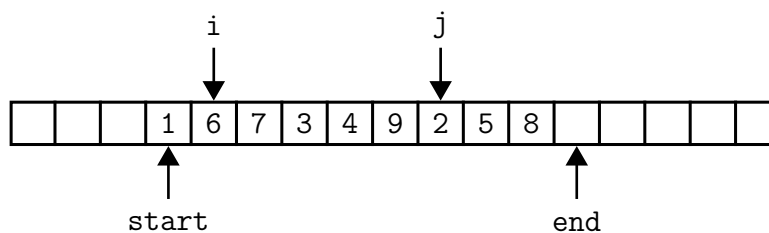
we increment an index variable, say `i`, until `x[i] >= pivot`:



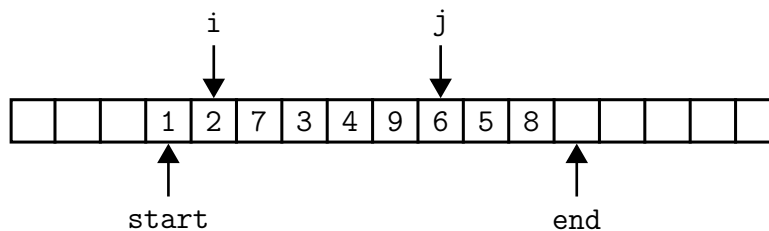
Next, starting with the one larger than highest index value, i.e. **end**,



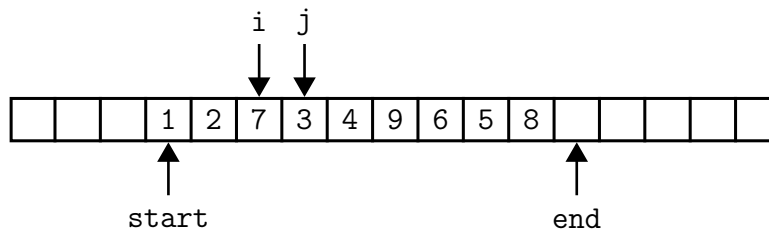
we decrement an index variable, say j , until $x[j] \leq \text{pivot}$:



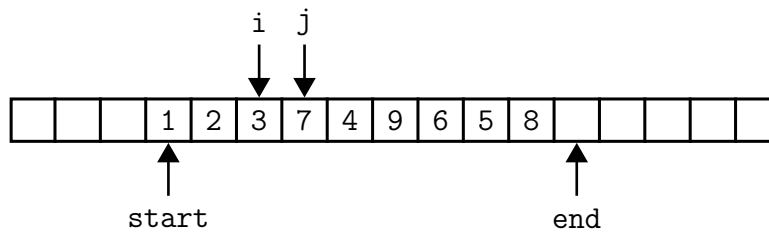
At this point, we swap $x[i]$ and $x[j]$:



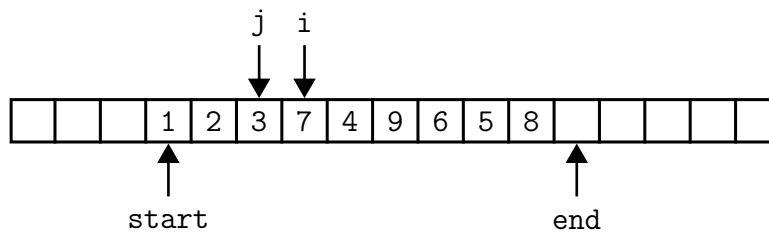
We repeat: we continually increment i and then perform the same check on i as before and we continually decrement j and perform the same check on j until we get to this point:



We swap $x[i]$ and $x[j]$ again to get



We repeat the same procedure to get this:



It's important (in fact *very* important) to note that you increment i before you check the loop termination condition on i . Likewise, it's important to note that you decrement j before you check the loop termination condition on j .

Note that at this point,

- $x[\text{start} \dots j]$ contains values $\leq \text{pivot}$
- $x[j + 1 \dots \text{end}]$ contains values $\geq \text{pivot}$

Here is Hoare's partitioning strategy:

```

ALGORITHM: hoare_partition
INPUTS: x - array
        start, end - index values, x[start..end-1] is to be sorted
        pivot_index - index of pivot

pivot = x[pivot_index]
i = start - 1
j = end

while 1:
    while 1:
        i = i + 1
        if x[i] >= pivot:
            break

    while 1:

```

```
        j = j - 1
        if x[j] <= pivot:
            break

    if i < j:
        swap x[i], x[j]
    else:
        break

return j
```

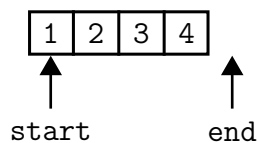
```
ALGORITHM: quicksort
INPUTS: x - array
        start, end - indices

n = end - start
if n > 1:
    j = hoare_partition(x, start, end)
    quicksort(x, start, j + 1)
    quicksort(x, j + 1, end)
```

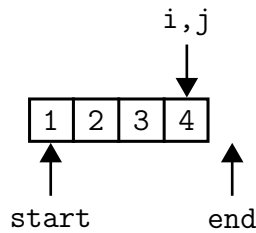
If you are sorting the whole array, of course for the first call to quicksort, `start` is 0 and `end` is $n - 1$ where n is the size of the array `x`.

The general idea is that on applying Hoare partition on `x[start .. end-1]`, you will get an index value returned, `j`, such that `x[start .. j]` has values \leq the pivot and `x[j + 1 .. end-1]` has values \geq the pivot. The pivot need not be at index `j`.

There's an important warning: If the pivot you pick is the *largest and unique* value and the pivot is at the *rightmost index*, you will get into an *infinite loop*. For instance consider this:



where we choose 4 to be the pivot. Then for this execution of Hoare partition we have this:



which will result in a return with value of $j = 3$. Unfortunately, back in quicksort, this will result in calling quicksort $\text{start} = 0$ and $\text{end} = 4$ again. The case where $i = j = \text{end} - 1$ is called the **trivial split case of Hoare partition**.

trivial split case of
Hoare partition

Therefore in your pivot selection do not pick a pivot value that is the largest and the unique value of the array that is at the rightmost index.

Exercise 103.11.1.

debug:
exercises/quicksort-
30/question.tex

- (a) And what if the pivot is the largest and unique value in the array and is at the beginning (index 0)?
- (b) What happens if the pivot is the largest and unique value in the array that is not at beginning and not at the end of the array?

(Go to solution, page 4632)

□

Exercise 103.11.2. Perform Hoare partition on $\{5, 3, 2, 6, 4, 1, 3, 7\}$ with pivot value of 5. What is the index value returned (i.e., the last value of j)? Is the index value returned always the index value of the pivot? On return is the value of i always the same as j ? Is the pivot at index j ? (Go to solution, page 4633)

debug:
exercises/quicksort-
21/question.tex

□

Exercise 103.11.3. Is Hoare partition (and hence quicksort using Hoare partition) stable? (Go to solution, page 4634)

debug:
exercises/quicksort-
23/question.tex

□

Exercise 103.11.4. Perform quicksort (using Hoare partition) on $\{1, 1, 1, 1\}$. What happens when you do the same for an array of size n of the same values? What is the runtime? (Go to solution, page 4635)

debug:
exercises/quicksort-
24/question.tex

□

Exercise 103.11.5. Perform quicksort (using Hoare partition) on $\{1, 2, 3, 4\}$ (which is already sorted). Say you pick the first value as pivot. What about

debug:
exercises/quicksort-
25/question.tex

the general case of $\{1, 2, 3, \dots, n\}$? What is the runtime? What if you use median of 3 for pivot selection? ([Go to solution](#), page 4636) ☐

Exercise 103.11.6. Give yourself a random array and perform a complete quicksort (using Hoare partition) on the array. Next, for each partitioning, compare the values of i and j . Repeat the above a couple of times. Conjecture a relationship between i and j . ([Go to solution](#), page 4637) ☐

debug:
exercises/quicksort-
26/question.tex

Exercise 103.11.7. Recall that if we choose 4 for the pivot when performing quicksort (using Hoare partition) on $\{1, 2, 3, 4\}$, we run into a problem.

debug:
exercises/quicksort-
27/question.tex

- (a) What about the following: Perform Hoare partition on $\{4, 1, 2, 3\}$ where you choose 4 for the pivot. What is the index value returned? Will you run into an infinite loop when you perform the full quicksort on the above array using Hoare partition?
- (b) Repeat the above for $\{4, 1, 2, 4\}$.

([Go to solution](#), page 4638) ☐

Exercise 103.11.8. What will go wrong if you use this partitioning strategy instead (make sure you see that it's different from Hoare's partition):

debug:
exercises/quicksort-
29/question.tex

```
ALGORITHM: HOARE_PARTITION
INPUTS: x - array
        start, end - indices

pivot = select a pivot value in x[start..end]
i = start
j = end

while 1:
    while 1:
        if x[i] >= pivot:
            break
        i = i + 1

    while 1:
        if x[j] <= pivot:
            break
        j = j - 1

    if i < j:
```

```
        swap x[i], x[j]
    else:
        break

return j
```

([Go to solution](#), page 4639)



Exercise 103.11.9. This question is relevant to quicksort that uses any partition algorithm. (This is actually a *very* important question ...) Assume \mathbf{x} is an array of distinct values.

debug:
exercises/quicksort-
31/question.tex

- (a) Prove that in quicksort every pair of values in \mathbf{x} is never compared twice.
- (b) Let's be more precise. Assume n is the size of \mathbf{x} . Pick two random values \mathbf{a} and \mathbf{b} in \mathbf{x} . During the first partition of \mathbf{x} (while performing quicksort), what is the chance that \mathbf{a} and \mathbf{b} are compared?

([Go to solution](#), page 4640)



Solutions

Solution to Exercise [103.11.1](#).

Solution not provided.

debug:
exercises/quicksort-
30/answer.tex

Solution to Exercise [103.11.2](#).

$\{3, 3, 2, 1, 4, 6, 5, 7\}$. Return index value is $j = 4$. The value of i is 5. Note that the pivot 5 ends up at index 6.

debug:
exercises/quicksort-
21/answer.tex

Solution to Exercise [103.11.3](#).

No. Consider $\{5, 5\}$. The two 5's switch their relative positions.

debug:
exercises/quicksort-
23/answer.tex

Solution to Exercise [103.11.4](#).

Solution not provided.

debug:
exercises/quicksort-
24/answer.tex

Solution to Exercise [103.11.5](#).

Solution not provided.

debug:
exercises/quicksort-
25/answer.tex

Solution to Exercise [103.11.6](#).

Solution not provided.

debug:
exercises/quicksort-
26/answer.tex

Solution to Exercise [103.11.7](#).

Solution not provided.

debug:
exercises/quicksort-
27/answer.tex

Solution to Exercise [103.11.8](#).

The point: in Hoare's partition, the `i` increments before the check and the `j` decrements before the check. For the algorithm in this question, the check on `i` is before the increment and the check on `j` is before the decrement. If the `i` and `j` both point to a pivot and $i < j$, `i` and `j` will stop moving – you get an infinite loop. This is the case for instance if the array is `{1,2,2,3}` and 2 is the pivot.

debug:
exercises/quicksort-
29/answer.tex

Solution to Exercise [103.11.9](#).

Solution not provided.

debug:
exercises/quicksort-
31/answer.tex

103.12 Quicksort: runtime analysis debug: quicksort-runtime.tex

Here's the runtime analysis of quicksort. Suppose for an array of size n , I pick k for the pivot index (of course $0 \leq k \leq n - 1$.) Therefore the size of `left` is k and the size of `right` is $n - k - 1$. Therefore

$$T_{\text{QUICKSORT}}(n) = T_{\text{PARTITION}}(n) + T_{\text{QUICKSORT}}(k) + T_{\text{QUICKSORT}}(n - k - 1)$$

(ignoring the base case.) The problem of course is that we do not know the value of k . Suppose we assume that k is approximately $n/2$ (i.e., either $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$). Then ignoring floor and ceiling the runtime is of the form

$$T_{\text{QUICKSORT}}(n) = 2T_{\text{QUICKSORT}}(n/2) + T_{\text{PARTITION}}(n)$$

(again ignore the base case.) Clearly $T_{\text{PARTITION}}(n) = O(n)$. Hence the runtime of quicksort is of the form

$$T(n) = \begin{cases} A & \text{if } n = 0, 1 \\ 2T(n/2) + Bn + C & \text{otherwise} \end{cases}$$

Exercise 103.12.1. Suppose you're given an array of ascending values (example: $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$) and the median (example: 5) is chosen as the pivot, what will each partition method do to the array? ([Go to solution](#), page 4643) □

debug:
exercises/quicksort-
14/question.tex

Exercise 103.12.2. Suppose you know that an array is almost sorted. How would you pick the pivot? ([Go to solution](#), page 4644) □

debug:
exercises/quicksort-
15/question.tex

Exercise 103.12.3. Suppose an array contains long chains of ascending values. How would you modify quicksort so that you take advantage of this fact? ([Go to solution](#), page 4645) □

debug:
exercises/quicksort-
16/question.tex

Exercise 103.12.4. Design a version of the quicksort so that during partition phase, instead of having just a LEFT chunk and a RIGHT chunk, you also have a chunk for all pivot values, i.e., if the pivot appears three times, all these three values go into this pivot chunk. ([Go to solution](#), page 4646) □

debug:
exercises/quicksort-
17/question.tex

Exercise 103.12.5. Test your quicksort with an array with a huge number of duplicates. Why is the performance for this case so slow? What is one

debug:
exercises/quicksort-
18/question.tex

possible modification to quicksort to overcome this issue?

([Go to solution](#), page [4647](#))



Solutions

Solution to Exercise [103.12.1](#).

Solution not provided.

debug:
exercises/quicksort-
14/answer.tex

Solution to Exercise [103.12.2](#).

Solution not provided.

debug:
exercises/quicksort-
15/answer.tex

Solution to Exercise [103.12.3](#).

Solution not provided.

debug:
exercises/quicksort-
16/answer.tex

Solution to Exercise [103.12.4](#).

Solution not provided.

debug:
exercises/quicksort-
17/answer.tex

Solution to Exercise [103.12.5](#).

Solution not provided.

debug:
exercises/quicksort-
18/answer.tex

103.13 Comparison sort: lower bound debug:

comparison-sort-lower-bound.tex

A comparison sort on an array of size n must use more than $n \lg n - 1.44n$ comparisons in the worst case. So the runtime complexity for mergesort and quicksort, being $O(n \log_2 n)$ is the best possible for comparison based sort.

While $T(n) = O(g(n))$ says that $T(n)$ is asymptotically bounded *above* by $g(n)$, one can also define the concept of asymptotic lower bound and say that $T(n)$ is asymptotically bounded *below* by $g(n)$, and we write $T(n) = \Omega(g(n))$, if there are constants N and C such that

$$|T(n)| \geq C|g(n)|$$

for $n \geq N$. With this concept, we would say that if $T(n)$ is the runtime of any comparison-based sorting algorithm, then $T(n)$ is asymptotically bounded below by $n \lg n$, i.e.,

$$T(n) = \Omega(n \lg n)$$

(See CISS358 for details.)

Basically, the above result says that *asymptotically*, mergesort has the best possible runtime. This is the same for quicksort as long as the data does not drive quicksort toward the worst case. Also, later I'll talk about heapsort which also has a runtime of $O(n \lg n)$ like mergesort.

103.14 C++: function pointers, functors, lambdas

debug: function-pointers-functors-lambdas.tex

103.14.1 Function as arguments

Frequently you want a function to be an argument. For instance suppose you want a bubblesort function to be able to sort all type of data, not just integers. In that case you might need a special comparison function on your values.

Also, what if you want to sort in descending order? If your data can only be arranged in two ways (ascending or descending), then a boolean value can be used to select how the sorting can be done. But even when sorting integers, there are times when you want to compare them in different ways. For instance what if you are sorting 8-digit integers and you want those that begin with 9999 to appear before any other numbers?

In this case it would be better if the function prototype looks like this:

```
void bubblesort(std::vector< int > & v, [some comparison function]);
```

where the comparison function, say `cmp`, takes in two integers `x` and `y`, and `cmp(x, y)` returns `true` if `x` should appear before `y`.

This requires a function to be an argument for function calls. In C++ there are three ways of achieving this.

- Function pointers
- Functors
- Lambda functions

103.14.2 Function pointers

Suppose you have a function with the following prototype:

```
int f(double, char, bool);
```

then you can think of `f` as a variable. Except that you have to think of `f` as a pointer – it's a **function pointer**. Try this:

function pointer

```
#include <iostream>

int f(double x, char y, bool z)
{
    std::cout << "f\n";
}
```



```
int main()
{
    int (*v) (double, char, bool) = f;
    v(3.14, 'A', true);
    return 0;
}
```

With this, a function can now accept a function as input:

```
#include <iostream>
#include <cmath>

double derivative(double (*f) (double), double x, double h=0.0001)
{
    return (f(x + h) - f(x)) / h;
}

double square(double x)
{
    return x * x;
}

int main()
{
    std::cout << derivative(sin, 0) << '\n';
    std::cout << derivative(cos, 0) << '\n';
    std::cout << derivative(tan, 0) << '\n';
    std::cout << derivative(square, 0) << '\n';

    return 0;
}
```

In the above the `derivative` function works with the value of `f` which holds function addresses of `sin`, `cos`, etc. In other words, `f` is a function pointer and it can hold the address of a function.

Exercise 103.14.1. Write a bubblesort function that accepts the equivalence of a “<”.

```
bool less(int x, int y)
{
    return (x < y);
}

bool greater(int x, int y)
{
    return (x > y);
}

void bubblesort(int x[], int n,
                bool (*less) (int, int))
{
    // TODO
}
```

(More programmers will call the comparison function `cmp` or `less` or `comparator` etc. For myself I also use `is_before`.)

In that case

```
bubblesort(x, n, less);
```

will sort `x[0..n-1]` in ascending order while

```
bubblesort(x, n, greater);
```

sorts in descending order.

Exercise 103.14.2. Now make “<” the default case so that

```
bubblesort(x, n);
```

will sort `x[0..n-1]` in ascending order while

```
bubblesort(x, n, cmp);
```

will sort according to the function pointer passed in to `cmp`.

Exercise 103.14.3. It’s even better to use pointers. Complete this:

```
// Sorts values *start, ..., *(end - 1)
void bubblesort(int * start, int * end, bool (*cmp) (int, int))
{
    // TODO
}
```

Exercise 103.14.4. Now make everything into templates. Complete this:

```
template < typename T >
void bubblesort(T * start, T * end, bool (*cmp) (const T &, const T &))
{
    // TODO
}
```

and make sure the default case is ascending order.

103.14.3 Functors

It's not too surprising that since `<` and `>` comparisons are so common as functions that in fact C++ has already defined `std::less` and `std::greater` that does the obvious except that these are not functions – they are **structs** or classes with function call operator `operator()`. Variables/objects from such structs/classes are called **functors**. You have already seen such objects in CISS245.

functors

In other words `std::less` is something like

```
namespace std
{
    template < typename T >
    struct less
    {
        bool operator()(const T & x, const T & y)
        {
            return (x < y);
        }
    };
}
```

or

```
namespace std
{
    template < typename T >
    class less
    {
    public:
        bool operator()(const T & x, const T & y)
        {
            return (x < y);
        }
    };
}
```

Make sure you run this:

```
#include <iostream>
#include <functional>

int main()
{
    std::cout << std::less< int >()(1, 2) << '\n';
    std::cout << std::less< int >()(1, 1) << '\n';
    std::cout << std::less_equal< int >()(1, 1) << '\n';
    return 0;
}
```

Functors are useful because they are just struct values or objects. And you know from CISS245 that you can pass struct values or objects into functions and you can return struct values or objects from function calls. Of course this can also be done with function pointers. But it's clear that the C++ code to create functors is simpler. Other reasons include the fact that struct values and objects can store values in their member variables. For instance a functor can keep the last 10 computations as a history and use them if necessary to speed up computation by avoiding re-computations and simply use a lookup table. You can do that with a function too (with static variables), but a class is more flexible since you can also use inheritance in case you have functor classes which are related by inheritance. By the way, for most usage, functors are not that complex that you would need inheritance. So most functor struct are used instead of functor classes.

In the case of using C++ STL, I'll use functors to do the following:

- To specify the sorting order for sorting functions. In particular the sorting functions provided by C++ STL uses functors to specify your own sorting order.
- To specify the ordering of values in a container

Exercise 103.14.5. Complete the following and test it:

```
template < typename T >
struct less
{
    bool operator()(const T & x, const T & y) const
    {
        return (x < y);
    }
};

template < typename T >
```

```
void bubblesort(std::vector< T > & v)
{
}

template < typename T, typename Comparator >
void bubblesort(std::vector< T > & v, const Comparator & cmp)
{
}
```

After you are done, write a version for pointers:

```
template < typename T >
struct less
{
    bool operator()(const T & x, const T & y) const
    {
        return (x < y);
    }
};

template < typename T, typename Comparator >
void bubblesort(T * start, T * end, const Comparator & cmp)
{
}
```

□

Note that comparator objects usually do not have member variables. So frequently programmers will simply not bother making the method constant and write

```
bool operator()(const T & x, const T & y)
```

instead of

```
bool operator()(const T & x, const T & y) const
```

and also will write

```
template < typename T, typename Comparator >
void bubblesort(std::vector< T > & v, Comparator & cmp)
{
}
```

instead of

```
template < typename T, typename Comparator >
void bubblesort(std::vector< T > & v, const Comparator & cmp)
{
}
```

103.14.4 Lambda expressions

`[] (int x){ return x * x; }` is an example of a lambda expression. This expression computes the square of an integer. Try this:

```
#include <iostream>

int main()
{
    std::cout << ([] (int x){ return x * x; })(3) << '\n';
    return 0;
}
```

You'll get an output of 9. A lambda expression is an anonymous function because it is an expression of function computation – there's no function name involved. Of course you can give a lambda function to a name if you like:

```
#include <iostream>

int main()
{
    auto f = [] (int x){ return x * x; };
    std::cout << f(3) << '\n';
    return 0;
}
```

You can also have a lambda expression template:

```
#include <iostream>

int main()
{
    auto f = [] (auto x){ return x * x; };
    std::cout << f(3) << '\n';
    std::cout << f(3.1) << '\n';
    return 0;
}
```

By the way, you can pass by reference for lambda expressions. Instead of “auto x” like the above, you write “auto & x”. No surprises there.

Here's an example of a lambda expression with two inputs:

```
#include <iostream>

int main()
{
    auto f = [](auto x, auto y){ return x <= y; };
    std::cout << f(3, 4) << '\n';
    std::cout << f(3.1, 3.0) << '\n';
    return 0;
}
```

Exercise 103.14.6. Write a header file that contains the `max`, `min`, `swap` of two values defined using lambda expressions. Make sure they are templates. Test your header file. Lambdas are easy, right? ([Go to solution](#), page 4657)

debug:
exercises/functor-
0/question.tex

□

Although you can write functions like the above exercise, it's more common to write lambda expressions and not assign them to function names. For instance look at this:

```
#include <iostream>

template < typename T, typename Comparator >
void bubblesort(T * start, T * end, Comparator less)
{
    // TODO
}

int main()
{
    int x[] = {5, 3, 1, 4, 2};
    bubblesort(x, x + 5, [](int x, int y){ return x < y; }); // ascending
    bubblesort(x, x + 5, [](int x, int y){ return x > y; }); // descending
    return 0;
}
```

Here, you just want to write a lambda expression for a comparator when calling `bubblesort`. You don't really care to keep the lambda expression around after it's used.

The benefit of the lambda expression is that you don't have to create a class/struct for your comparison functor and you don't have to create a function for your comparison function pointer.

C++ lambda expressions come from lambda functions in the study of lambda calculus. More information in CISS445. (See my notes on C++ lambda functions for more information.)

Solutions

Solution to Exercise [103.14.6](#).

debug:
exercises/functor-
0/answer.tex

```
// file: main.cpp
#include <iostream>
#include "test.h"

int main()
{
    std::cout << min(3, 2) << '\n';
    std::cout << max(3, 2) << '\n';
    int x = 0, y = 42;
    swap(x, y);
    std::cout << x << ' ' << y << '\n';
    return 0;
}
```

```
// file: test.h
#ifndef TEST_H
#define TEST_H

auto min = [](auto x, auto y){ return (x <= y ? x : y); };
auto max = [](auto x, auto y){ return (x >= y ? x : y); };
auto swap = [](auto & x, auto & y){ auto t = x; x = y; y = t; };

#endif
```


103.15 C++ STL `std::sort` and

`std::stable_sort` debug: comparison-sort-stl.tex

C++ STL provides two comparison-based sorting functions.

- `std::sort` – for unstable sorting. This is a combination of quicksort and heapsort/insertionsort. The heapsort/insertionsort is used when the subarray to be sorted has a smaller size.
- `std::stable_sort` – for stable sorting. This is a mergesort implemented using bottom-up recursion and possibly with some slight variations.

103.15.1 Basic examples

Here are some examples:

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm> // for sorting functions

std::ostream & operator<<(std::ostream & cout,
                        const std::vector< int > & v)
{
    std::string delim = "";
    cout << '{';
    for (auto x: v)
    {
        cout << delim << x;
        delim = ", ";
    }
    cout << '}';
    return cout;
}

int main()
{
    std::vector< int > v0 {5, 3, 1, 4, 2};
    std::cout << v0 << '\n';
    std::sort(v0.begin(), v0.end());
    std::cout << v0 << '\n';

    v0 = {5, 3, 1, 4, 2};
    std::cout << v0 << '\n';
    std::stable_sort(v0.begin(), v0.end());
    std::cout << v0 << '\n';

    return 0;
}
```

```
}

```

Here's the output:

```
[student@localhost comparison-based-sort] g++ main.cpp; ./a.out
{5, 3, 1, 4, 2}
{1, 2, 3, 4, 5}
{5, 3, 1, 4, 2}
{1, 2, 3, 4, 5}
```

Of course if you want to change the sorting range, you just change the iterators in `std::sort(v0.begin(), v0.end())`. (For more information on iterators, see chapter on containers.)

Exercise 103.15.1. Modify the above to sort the first half of the `std::vector` object.

103.15.2 Changing the sorting order

You can specify the sorting order:

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm> // for sorting functions
#include <functional> // for std::less, std::greater
std::ostream & operator<<(std::ostream & cout,
                        const std::vector< int > & v)
{
    std::string delim = "";
    cout << '{';
    for (auto x: v)
    {
        cout << delim << x;
        delim = ", ";
    }
    cout << '}';
    return cout;
}

int main()
{
    std::vector< int > v0 {5, 3, 1, 4, 2};
    std::cout << v0 << '\n';
    std::sort(v0.begin(), v0.end(), std::less< int >());
}
```

```

    std::cout << v0 << '\n';

    v0 = {5, 3, 1, 4, 2};
    std::cout << v0 << '\n';
    std::stable_sort(v0.begin(), v0.end(), std::greater< int >());
    std::cout << v0 << '\n';

    return 0;
}

```

Here's the output:

```

[student@localhost comparison-based-sort] g++ main.cpp; ./a.out
{5, 3, 1, 4, 2}
{1, 2, 3, 4, 5}
{5, 3, 1, 4, 2}
{5, 4, 3, 2, 1}

```

103.15.3 Writing your own comparison

Here's an example where I provide my own comparison functor:

```

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

std::ostream & operator<<(std::ostream & cout,
                        const std::vector< int > & v)
{
    std::string delim = "";
    cout << '{';
    for (auto x: v)
    {
        cout << delim << x;
        delim = ", ";
    }
    cout << '}';
    return cout;
}

struct less // or use class
{
    bool operator()(int x, int y)
    {
        return (x < y);
    }
}

```

```
};

struct greater // or use class
{
    bool operator()(int x, int y)
    {
        return (x > y);
    }
};

int main()
{
    std::vector< int > v0 {5, 3, 1, 4, 2};
    std::cout << v0 << '\n';
    std::sort(v0.begin(), v0.end(), less());
    std::cout << v0 << '\n';

    std::vector< int > v1 {5, 3, 1, 4, 2};
    std::cout << v1 << '\n';
    std::stable_sort(v1.begin(), v1.end(), greater());
    std::cout << v1 << '\n';

    return 0;
}
```

(The `greater` here is similar to the `std::greater` except that `std::greater` is a template.) Here's the output

```
[student@localhost comparison-based-sort] g++ main.cpp; ./a.out
{5, 3, 1, 4, 2}
{1, 2, 3, 4, 5}
{5, 3, 1, 4, 2}
{5, 4, 3, 2, 1}
```

Exercise 103.15.2. Create a comparator that sort integers in the usual way except that integers that begin with the digit 9 comes earlier. For the integers that begin with digit of 9, sort them based on the remaining digits. Here's a test case:

```
[student@localhost comparison-based-sort] g++ main.cpp; ./a.out
{59, 93, 91, 45, 55, 3}
{91, 93, 3, 45, 55, 59}
```

The first line of output is the array before the sorting and the second line of output is the array after the sorting. □

103.16 Real world sorting debug: comparison-sort-real-world.tex

In general, for most programming tasks, you would use whatever sorting algorithm is provided on your development tools. However it's still important to know your sorting algorithms because sorting algorithms help train you to think algorithmically. They do appear in job interviews. (Among my students, the first to receive an interview from google couldn't answer a question on quicksort implementation – no I was not his 350 instructor.) And furthermore, there are times when you do need to write your own sorting algorithms.

Another thing to note is that real world sorting is almost never based on just one algorithm. Frequently you have to use several. For instance many sorting algorithms from libraries uses quicksort or mergesort, but for base cases, when the size of the subarray to sort is “small”, it switches to something else such as insertion sort. Furthermore, the sorting algorithms in this chapter are general sorting algorithms. For specific scenarios, there might be something special about the data that allows one to tweak and optimize a sorting algorithm in a non-standard way.

Here's a post on quorum:

What is the fastest sorting algorithm?

“Oh, such an easy question to answer.

- The fastest sorting algorithm is the one that exploits the peculiarities of your data on your hardware, subject to your external constraints.
- The second-fastest sorting algorithm is the one in the good enough sort library (perhaps the one in your programming language's standard library) that you didn't have to write.

“The reason why you go through all those sort algorithms as an undergraduate isn't because you can just drop one into your program and it's optimised for everything. It's to get you to think algorithmically.

“I've written quite a bit of sorting code in my time, including the sort subsystem for a database server (which took six months, and involved at least six “algorithms”). Real-world industrial-strength sort systems have some interesting features that you tend not to see as an undergraduate:

- The basic sort algorithms that you learned as an undergraduate are pieces from which a “real” sort is written. You may have already seen this when you were told that quick sort should use insertion sort as its “base case”.
- Industrial-strength sort systems keep a close eye on what is hap-

pening and adjust or retune.

- The algorithm itself is often not the thing that has the biggest effect on performance, it's the constraints under which the sort algorithm has to run. Accessing a sort key can be an expensive operation if it involves an additional disk seek or network packet or even cache miss. (Real example: Consider sorting XML documents by title. Accessing that sort key requires parsing the XML.)
- Everything involves a tradeoff. I mean *everything*.

“The sort algorithm built into your programming language’s standard library (or algorithms) are often comparison-based, not radix-based. Did you ever stop to wonder why? It’s not for raw performance reasons, it’s because the algorithm needs to work on user-defined types, and it’s much more convenient for the programmer to provide a “less than” comparison operator than some other query. The programmer has a job to do and just wants this sorted with the minimum of boilerplate.

“As another real-world example, I once had to write some firmware which involved reading 1000 or so integers from a hardware device into a buffer and then performing quantile extraction. The microcontroller that this ran on had essentially no extra read-write memory available, so algorithms such as radix sort or quicksort were not an option, and the size of the data set wasn’t large enough to make heap sort worth it. I used Shell sort which was easily the best tradeoff: subquadratic with no extra read/write memory needed.

“And as always, make sure that the sort algorithm is actually a bottleneck before getting fancy.”

Index

comparison-based, [4502](#)

distribution, [4502](#)

divide-and-conquer, [4559](#)

external, [4502](#)

function pointer, [4649](#)

functors, [4652](#)

inplace, [4501](#)

insertionsort, [4522](#)

internal, [4502](#)

median of median of 3, [4596](#)

median of three, [4595](#)

ninther, [4596](#)

partition algorithm, [4581](#)

partitioning, [4581](#)

pivot, [4581](#)

pivot selection algorithm, [4582](#)

randomized quicksort, [4595](#)

rotation, [4522](#)

stable, [4501](#)

trivial split case of Hoare partition,
[4629](#)