

Ryan Christopher

MET CS566

Assignment 3

1. Design a stack that supports push, pop, and retrieving the minimum element in constant time.

The folder “stack” contains mod.rs with the Stack struct. Tests are called in main.rs

```
// Stack struct has two fields, elements (stored as a vector) and
// min_index (an optional usize value)
#[derive(Default)]
2 implementations
pub struct Stack {
    pub elements: Vec<i32>,
    pub min_index: Option<usize>,
}

// implementations on the Stack struct
impl Stack {
    // push an element to the end of the stack
    fn push(&mut self, new_val: i32) {
        self.elements.push(new_val);
        // if the vector is empty, the first value will be the minimum value
        if self.elements.len() == 1 {
            self.min_index = Some(0);
        }
        // otherwise compare the value added to the current minimum
        else {
            if self.elements[self.min_index.unwrap() as usize] > new_val {
                self.min_index = Some(self.elements.len() - 1);
            }
        }
    }

    // pop the element from the end of the stack
    fn pop(&mut self) -> Option<i32> {
        let pop_element: Option<i32> = self.elements.pop();
        // call find_min when the minimum value is popped
        if self.min_index == Some(self.elements.len()) {
            self.find_min();
        }
        pop_element
    }

    // when a new minimum value needs to be found, find_min
    // will iterate through the elements and find the index
    // of the lowest value
    fn find_min(&mut self) {
        let mut new_min_index: usize = 0;
        for n: usize in 1..self.elements.len() {
            if self.elements[n] < self.elements[new_min_index] {
                new_min_index = n;
            }
        }
        self.min_index = Some(new_min_index);
    }

    // retrieve the minimum value from the stack
    fn min(&self) -> String {
        if self.elements.len() > 0 {
            self.elements[self.min_index.unwrap()].to_string()
        }
        // if the vector is empty, there is no min
        else {
            "None".to_string()
        }
    }
}
} impl Stack
```

```
=== Stack 1 ===
[]
Minumum Element: None
[5, 2, 10]
Minumum Element: 2
[5, 2, 10, 1]
Minumum Element: 1
Popped val: 1
[5, 2, 10]
Minumum Element: 2
=== Stack 2 ===
[]
[1, 7, 2]
Minumum Element: 1
Popped val: 2
[1, 7]
```

2. You wish to store a set of n numbers in either a max-heap or a sorted array. For each application below, state which data structure is better, or if it does not matter. Explain your answers.

a) Want to find the maximum element quickly.

To find the maximum element quickly, it does not matter if you use a max-heap or a sorted array. The max heap would have the maximum element at the root of the tree, and the sorted array would have the maximum element at the end of the array. In this case, retrieving the element from either data structure would be $O(1)$.

b) Want to be able to delete an element quickly.

To delete an element quickly, it would be better to store a set of n numbers as a sorted array. This is because a sorted array can quickly shift the position of all elements by one to accommodate the deleted element, whereas a max-heap could potentially require reheaping the entire set.

c) Want to find the median element quickly.

To find the median element quickly, it would be better to store a set of n numbers as a sorted array. Since the index value of an element is directly tied to its size, the median element can quickly be found by dividing the length of the array by 2 and finding the value at that index.

3. Implement an external sort, which uses intermediate files to sort files bigger than main memory. What is the fastest external sort algorithm? What about mergesort, heapsort, quicksort, and B-tree based sort algorithms for external sort? Test your algorithm within the limits of time / power, RAM of your computer.

The folder "external_sort" contains mod.rs with the sort function. Tests are called in main.rs.

```
24 println!("==== Mergesort ====");
25 external_sort::sort(method: "mergesort", file: "./rand_nums.txt");
26 println!("==== Heapsort ====");
27 external_sort::sort(method: "heapsort", file: "./rand_nums.txt");
28 println!("==== Quicksort ====");
29 external_sort::sort(method: "quicksort", file: "./rand_nums.txt");
30 println!("==== B-tree Sort ====");
31 external_sort::sort(method: "btree", file: "./rand_nums.txt");
```











For this problem, I have a loop that is in main.rs that generates a sequence of 12.5 million integers between 1 and 25 million.

```
rand_nums.txt
1 22584039 7645579 7990997 785676 23457317 14309845 16084507 17075776 16818593 13352578
2 12812869 7831201 17467202 24345634 3231559 11727832 6690905 24814955 21435143 4692042
3 2773076 3677283 23641674 1148839 9143507 20838813 17442751 20642465 14460998 16667278
4 10511444 16471130 8902846 8496224 1335901 12065308 24129014 3145976 4130102 15973623
5 13687971 19134707 13197854 14671577 1169910 23339239 982618 2833545 4376744 19023873
6 14144111 12596415 19697806 341540 3679171 8442388 7438987 11114674 6301617 1261043 2
7 17665282 10242523 3844010 5474082 23855170 7868848 6132875 14345718 19974047 1285867
8 8352720 10182299 11118538 14782501 13579539 20351657 5660758 20602682 21470949 218700
9 2496719 16373179 19750549 7310377 21579896 20702876 21077170 11835506 5078072 1687410
10 9387840 8381949 15804317 3796304 1519700 13876287 6009104 17112354 4793931 5553371 1
11 13259569 23904124 3110385 9941691 12126722 12581925 16644882 22150343 17129117 54242
12 7037530 14278457 6001264 10800490 23636559 6864344 16559908 3350337 14621685 2337681
13 24763850 23924828 11062477 19545196 10460685 3918666 14982602 10481201 13028924 86278
14 2449848 6233553 17609490 979695 2811053 14252022 78292 22984010 6941840 1253775 22230
15 3607134 22848818 4339737 2758235 7544967 9519714 5470481 23350417 3015838 16273372 4
16 4827864 7577713 11385293 7823732 6084705 3097757 7152524 16595150 3978586 6705058 14
17 17425814 22982218 6607751 21127759 3136131 11176179 19019127 19821684 21507462 24944
18 10222697 7823545 21750750 23218218 10003065 11031273 5745832 12979012 1997147 1232410
19 9625663 6035756 6051028 10261647 3828930 16351117 20522197 22695800 54772 20578373 78
20 18580329 20618949 13083986 232699 13710052 11192017 10024971 20498869 15530308 11440
21 15599018 6635074 4245628 5490935 18078132 19559931 24311443 22296478 5656763 22877538
22 10557904 7321506 23983205 5629283 15557639 8303453 23228283 770083 14738261 11588043
23 3998874 14690712 15466827 10165129 1005077 4415284 22477320 224578 7728949 24408775
24 5855276 6058123 8635951 14994631 7187550 23887310 6855510 20813475 11019369 4545926
25 18734099 17107883 21635944 9146301 2316982 12250941 15032902 17345115 10040789 109458
26 18072990 19941244 16556115 9592193 8989661 19806305 7558582 13631027 22481930 406202
27 17212593 14554933 21095617 7310590 12599057 6707655 18661406 24260282 7125708 3852588
28 2724213 11017447 17979252 15005636 5601191 13418943 24231888 17432989 10549999 19903
29 7363857 17843023 19814629 21749699 18910311 14069062 17630207 9287870 10979332 37072
30 1667702 18674920 8865882 11620168 23210308 22669409 18877578 12998151 16499305 10351
31 2399469 15826254 20332271 22553432 22492146 2275853 2901319 22646845 10398008 189510
32 10689008 21046512 14883576 22255497 20003866 20605206 24955210 17921370 736735 230058
33 14439905 10544243 19963102 7598127 18701918 4088354 18750075 3438536 6689459 18944508
34 10747549 19219297 18774181 17577359 12269467 20701532 21092207 15738248 2899797 1710
35 21824264 18409379 11311037 23503370 5948190 19879717 2380493 14509915 15389923 28170
36 5548437 6373096 8401717 9260683 18675308 8396381 24623316 22104152 13208236 2063412
37 20130330 2335121 3070794 2830784 964762 13449774 15950162 10471635 10260006 19152536
38 13291029 10715160 21964791 12459768 3204263 13833491 13891549 4097520 16230857 862688
39 19559068 19606846 8957289 22879763 9364020 22455096 100247 14400351 10967642 8887807
40 1241158 15772370 4693588 24079603 10185592 18402159 16083319 7255332 1646915 4824400
41 3659191 8519619 11007441 14266566 5690607 11925940 18599701 1178090 7633122 24103449
```











The file is approximately 100mb:

rand_nums	2/25/2025 11:21 PM	Text Source File	104,489 KB
-----------	--------------------	------------------	------------











My algorithm then splits the large file into chunks and writes each chunk to a txt file, where each intermediate file is roughly 10mb. This is done with two groupings of \ intermediate files. One directory is created for the unsorted chunks, and another is created for the chunks once they become sorted.

 temp1	2/26/2025 12:27 AM	Text Source File	10,188 KB
 temp2	2/26/2025 12:27 AM	Text Source File	10,189 KB
 temp3	2/26/2025 12:27 AM	Text Source File	10,189 KB
 temp4	2/26/2025 12:27 AM	Text Source File	10,188 KB
 temp5	2/26/2025 12:27 AM	Text Source File	10,188 KB
 temp6	2/26/2025 12:27 AM	Text Source File	10,188 KB
 temp7	2/26/2025 12:27 AM	Text Source File	10,186 KB
 temp8	2/26/2025 12:27 AM	Text Source File	10,190 KB
 temp9	2/26/2025 12:27 AM	Text Source File	10,187 KB
 temp10	2/26/2025 12:27 AM	Text Source File	10,171 KB

▼ temp_sorted_files

-  temp1.txt
-  temp2.txt
-  temp3.txt
-  temp4.txt
-  temp5.txt
-  temp6.txt
-  temp7.txt
-  temp8.txt
-  temp9.txt
-  temp10.txt

▼ temp_unsorted_files

-  temp1.txt
-  temp2.txt
-  temp3.txt
-  temp4.txt
-  temp5.txt
-  temp6.txt
-  temp7.txt
-  temp8.txt
-  temp9.txt
-  temp10.txt

Here is what the unsorted temp files look like:

```
a3 > temp_unsorted_files > temp1.txt
1 22584039 7645579 7990997 785676 23457317
5820173 4529745 8397639 8931012 16245893
22376062 3721569 20062843 19006877 242409
12295493 17168282 12704643 12702846 55929
20630690 8296608 13612748 3460183 1091228
4907189 5823368 8979006 21512217 2174281
12935137 15348792 17414718 23564249 64753
22820635 23225660 16530307 18078433 15720
6400055 0000710 1005060 16410777 0070110
```

And here is what the sorted temp files look like:

```
1 1 46 70 71 73 90 101 124 125 134 136 144 150 202
808 820 821 828 928 960 965 978 1040 1073 1075 10
1372 1389 1432 1444 1497 1545 1558 1563 1591 1598
2126 2126 2131 2145 2162 2167 2169 2178 2190 2190
2449 2452 2546 2567 2600 2601 2609 2620 2644 2675
3197 3198 3212 3230 3231 3234 3263 3277 3289 3309
3689 3698 3716 3717 3718 3745 3837 3838 3845 3871
4254 4261 4279 4292 4333 4337 4458 4464 4486 4523
5203 5206 5206 5251 5257 5285 5293 5296 5298 5302
5794 5819 5837 5838 5874 5904 5995 5997 6055 6056
6477 6478 6502 6504 6510 6529 6574 6581 6644 6661
7139 7143 7162 7168 7180 7183 7199 7226 7256 7287
```


Once the chunks are combined, an output file is generated with the sorted values.

```
a3 > sorted_vals.txt
1 1
2 2
3 4
4 6
5 7
6 7
7 8
8 11
9 12
10 12
11 15
12 19
13 24
14 31
15 31
16 40
17 41
18 42
19 43
20 46
21 46
22 51
23 53
```

```
12499987 24999965
12499988 24999973
12499989 24999975
12499990 24999975
12499991 24999975
12499992 24999975
12499993 24999976
12499994 24999977
12499995 24999980
12499996 24999981
12499997 24999982
12499998 24999987
12499999 24999994
12500000 24999999
```

For each of the sorting algorithms, here was the time required to complete sorting:

```
Finished `dev` prof
Running `target\de
===== Mergesort =====
33.4596907s
```

```
Finished `dev` pro
Running `target\d
===== Quicksort =====
26.8642615s
```

```
Finished `dev` pr
Running `target\
===== Heapsort =====
28.1047856s
```

```
Finished `dev` prof
Running `target\de
===== Btree Sort =====
28.9904361s
```

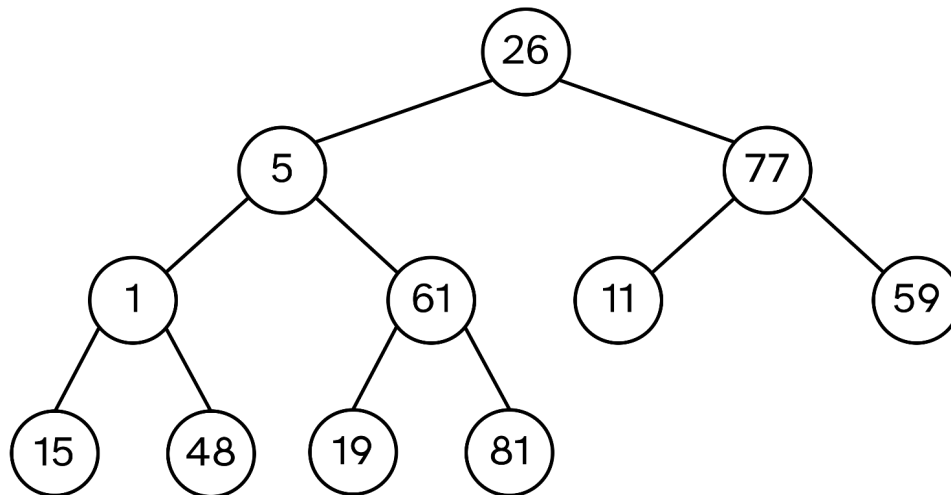
While the four of them were fairly close in running time, quicksort finished the fastest at 26.86 seconds, roughly 1.3 seconds ahead of heapsort.

4. Given the following array:

26, 5, 77, 1, 61, 11, 59, 15, 48, 19, 81

Sort this array by hand using **heapsort** and answer the following questions:

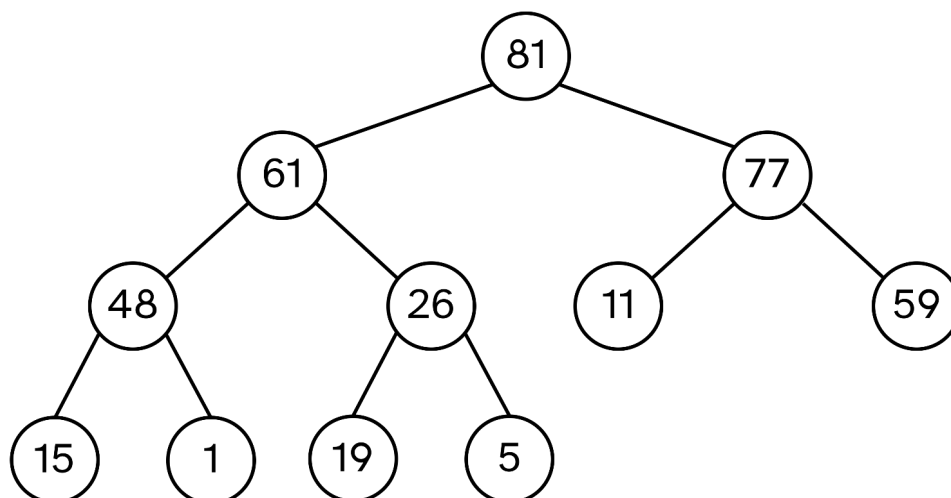
a) Draw the initial **complete** tree representation of the file **before** the **heap** is created.



array:

0	1	2	3	4	5	6	7	8	9	10
26	5	77	1	61	11	59	15	48	19	81

b) Draw the initial heap.

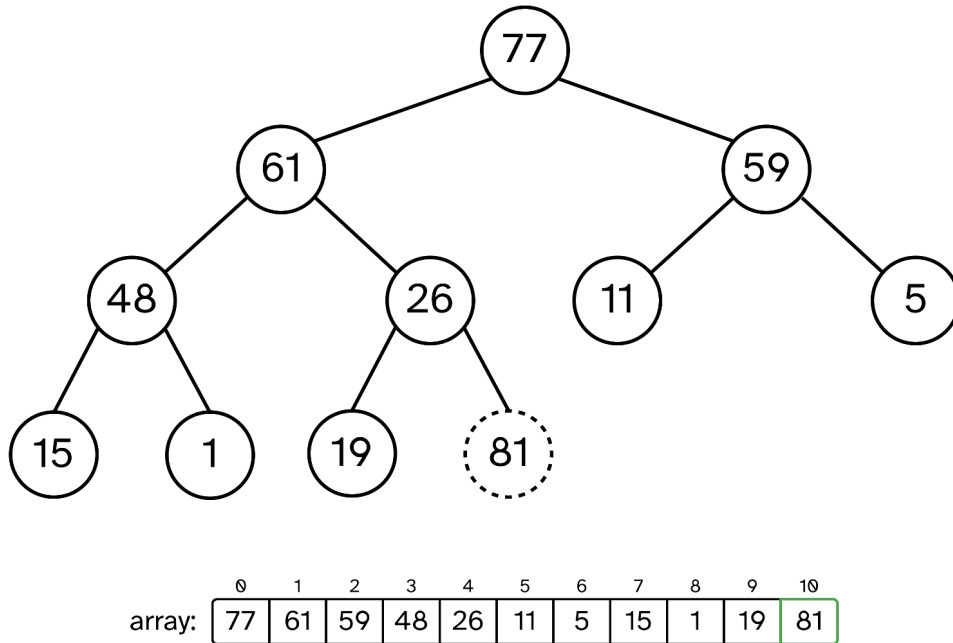


array:

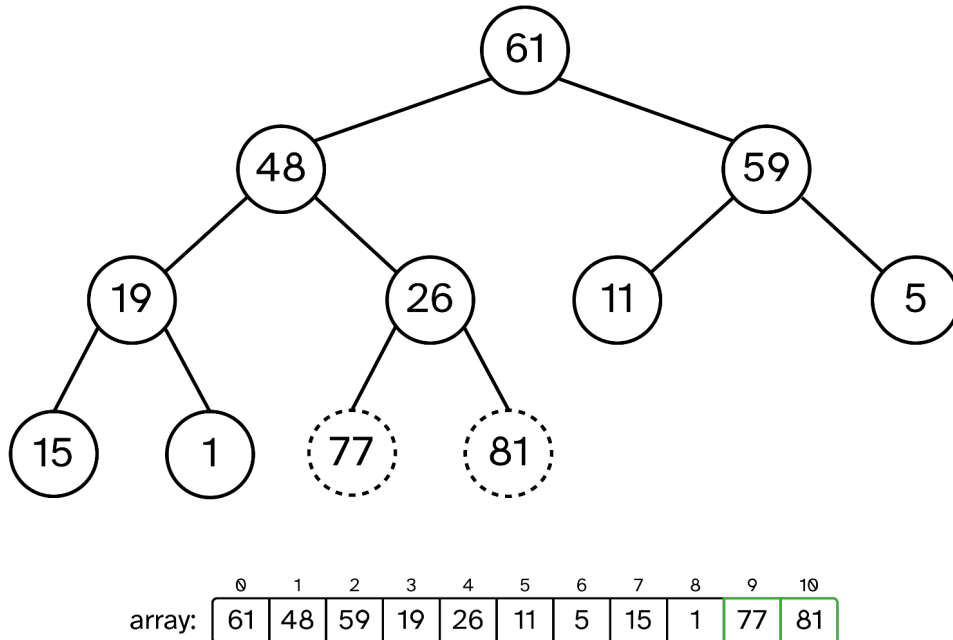
0	1	2	3	4	5	6	7	8	9	10
81	61	77	48	26	11	59	15	1	19	5

c) Continue with the sort for two iterations **past** the initial heap. What does the heap look like?

The first iteration, where 81 is removed and placed as a sorted value:



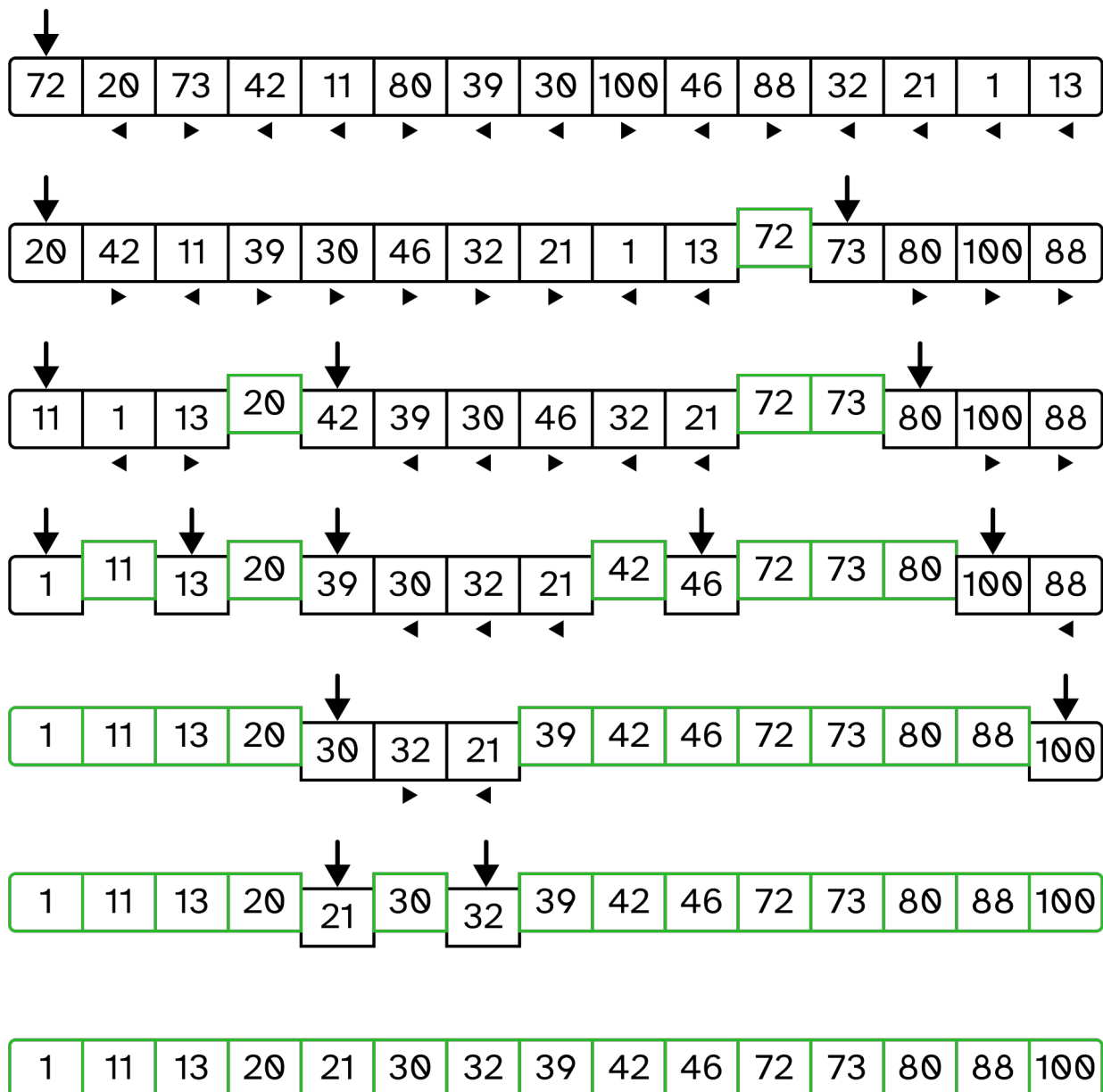
The second iteration, where 77 is removed and placed as a sorted value:



5. Use **quicksort** to sort the following array using the first element as the pivot element:

72, 20, 73, 42, 11, 80, 39, 30, 100, 46, 88, 32, 21, 1, 13

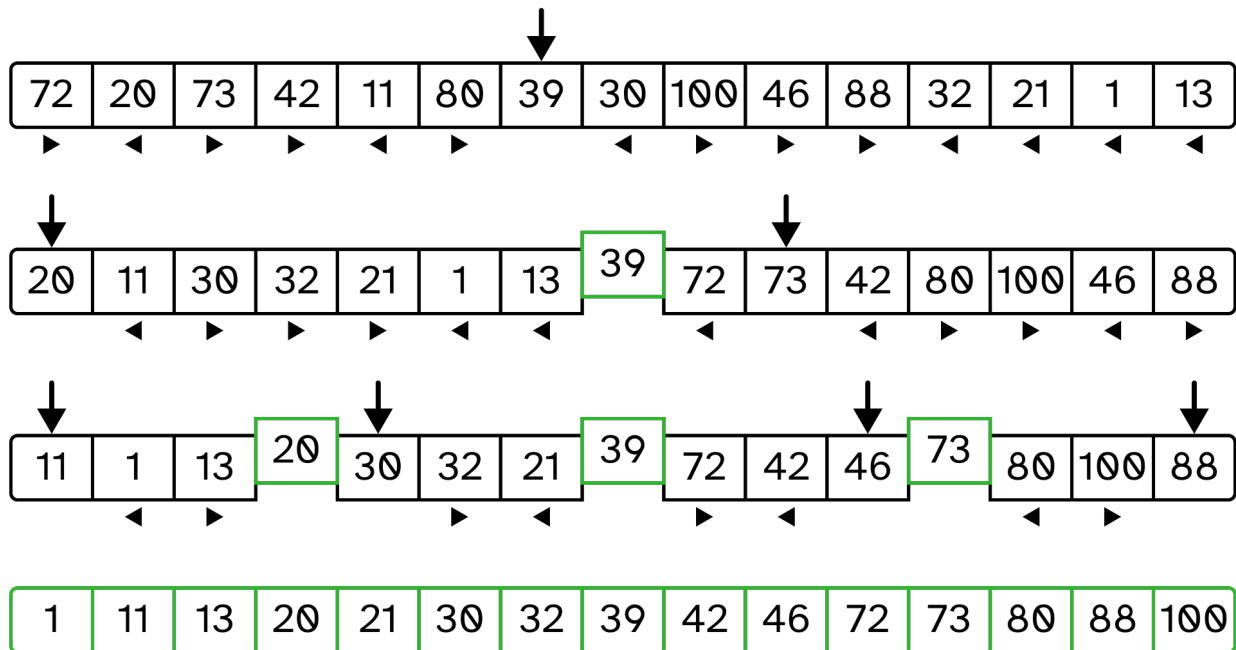
a) Continue sorting the entire array until totally sorted. During the entire sort – what was the maximum number of inversions removed by a single comparison and swap?



The maximum number of inversions were removed on the first swap, where 10 inversions were removed by placing 72 in the fifth to last spot.

b) What would be the sequence of the “best” pivot elements for this particular case?

The sequence of pivot elements 39, 20, 73, 11, 30, 46, 88 would be the best for this particular case as it would most efficiently split the list in half each iteration. As a result, the least number of iterations would need to occur to sort all the elements.



c) Does a balanced Binary Search Tree help?

Yes, because an in-order traversal of a Binary Search Tree yields the sorted order of the list, the binary tree would be able to produce the sorted array after the fifteen elements are added to the tree. This would be more efficient than the array using quicksort to produce the sorted list.