Aleksandr Itskovich (ai1221)
Collaborator: Elsie Kenyon

Assignment 1

Question 1: Asymptotic Notation

a)

Function	$\Theta(g(n))$	Explanation	Type
$\sqrt{\log n + (\log n)^2}$	$\log n$	$(\log n)^2$ grows faster than $\log n$. As n becomes huge, the term can be simplified to just $\sqrt{(\log n)^2}$ which just becomes $\log n$	Logarithmic
$n \log(n^{0.2n})$	$n^2 \log n$		Linear Logarithmic
$10!\, n!$	$n!$	The leading $10!$ term doesn't matter for large n. Simplifies to just $n!$	Factorial
$n^{0.2}(\log n)^2$	$n^{0.2}(\log n)^2$		Fractional Power, Poly Logarithmic
n^{2n}	n^n	As n becomes huge, the 2 in front doesn't really matter.	Linear Exponential
$(2^n + n)(\sqrt{n} + 2^n)$	4^n	$(2^n + n)$ simplifies to just 2^n because the exponential term grows so much faster than the linear term. Similarly, $(\sqrt{n} + 2^n)$ simplifies to just 2^n because the exponential term grows so much more quickly than the polynomial term. As n becomes huge we just have $(2^n)(2^n)$ which becomes 4^n.	Exponential
$n\,(\log n)^3$	$n\,(\log n)^3$		Linear Poly Logarithmic
$\frac{n^3 \log n}{(\log n)^2 + n}$	$n^3 \log n$	As n becomes huge, the multiplicative polynomial n^3 combined with the $\log n$ term grow much more quickly than the terms in the denominator. So we can simplify it to just the term in the numerator.	Polynomial Logarithmic
$\sqrt{\log n + 1}$	$(\log n)^{0.5}$	As n becomes huge, $\log n + 1$ simplifies to just $\log n$ leaving $(\log n)^{0.5}$	Fractional Poly Logarithmic

Correct Order:

$\sqrt{\log n + 1}, \sqrt{\log n + (\log n)^2}\,, n^{0.2}(\log n)^2, n\,(\log n)^3, n \log(n^{0.2n})\,, \frac{n^3 \log n}{(\log n)^2 + n}\,, (2^n + n)(\sqrt{n} + 2^n), 10!\, n!, n^{2n}$

b)

- Execute SwapSort1 on array A = [7, 6, 5, 4, 3, 2, 1] indexed from s = 1 to f = 7.

First Pass:

i		i + 1		f - 2		
7	6	5	4	3	2	1

	i		i + 2	f - 2		
5	6	7	4	3	2	1

		i		f – 2, i + 2		
5	4	7	6	3	2	1

			i	f – 2	i + 2	
5	4	3	6	7	2	1

				i, f – 2		i + 2
5	4	3	2	7	6	1

				f – 2	i	
5	4	3	2	1	6	7

Second Pass:

i		i + 2		f – 2		
5	4	3	2	1	6	7

	i		i + 2	f – 2		
3	4	5	2	1	6	7

		i		f – 2, i + 2		
3	2	5	4	1	6	7

			i	f – 2	i + 2	
3	2	1	4	5	6	7

				f – 2, i		i + 2
3	2	1	4	5	6	7

				f – 2	i	
3	2	1	4	5	6	7

Third Pass:

i		i + 2		f – 2		
3	2	1	4	5	6	7

	i		i + 2	f – 2		
1	2	3	4	5	6	7

		i		f – 2, i + 2		
1	2	3	4	5	6	7

			i	f – 2	i + 2	
1	2	3	4	5	6	7

				f – 2, i		i + 2
1	2	3	4	5	6	7

				f – 2	i	
1	2	3	4	5	6	7

Fourth Pass:

On the 4th pass no more swaps are performed because the array is already sorted; after this pass the algorithm ends.

- Which of the above algorithms is correct?

SwapSort1 is incorrect because it does not work on all arrays. For example:

i, f - 2		i + 2
1	100	3

According to the way this algorithm is written, it would end after a single pass without actually sorting the array. This makes the algorithm *incorrect* because in order to be correct, an algorithm must work on *all inputs*.

Because it is always swapping i with i + 2, SwapSort1 has a "blind spot" in the i + 1 position, such that two elements might never end up in sorted order for certain array configurations. Here is another example:

First Pass:

i	f - 2	i + 1	
2	3	3	1

	f – 2, i		i + 1
4	3	2	1

	f – 2	i	

4	1	2	3

Second Pass:

i	f – 2	i + 2	
4	1	2	3

In the second pass, SwapSort1 will compare 4 with 2 and 1 with 3, and will not perform any swaps in either case. Then the array will end, without returning a sorted result.

Unlike SwapSort1, SwapSort2 *is* correct. In fact, it is just a modification on Bubble Sort. There are two while-loops we need to worry about. The key thing to know is that the first while-loop continues until there are no more swaps; because the swaps only ever move larger values towards the end of the array, we know that we will eventually reach a state where the loop will terminate because there will be no more larger values to swap with smaller values (all the largest values will be in the back).

The second while loop is the same as Bubble Sort, which we know to be correct. Therefore, whatever the state of the array at the end of the first loop, we know that the second *while-loop* will successfully sort it and terminate.

- What is the worst-case number of comparisons made by SwapSort2 when the input array A has length n and is sorted in reverse order?

In this algorithm there are 2 *while* loops, each of which carries out a *for-loop*.

The first for-loop ranges from 1 to n – 2. A comparison is performed at every step of the iteration, therefore each full run of the for-loop performs **n – 2** comparison total.

The question then is, how many times does the first *while-loop* repeat?

In the case when the input array is sorted in reverse order, we have the situation where all of the *largest* values are in the front of the array. With every pass of the *first* while loop, we bubble up the first *two* largest values in the unsorted portion of the array into the *final* two slots of the array. Once the two largest values are occupying positions *n* and *n – 1* in the array, we know that there are no long any larger values that they are going to be swapped with in future passes. In other words, we can think of the last two positions of the array as already being in their *final position*, immune to any more swaps. Therefore, we are still only going to have swaps in the remaining portion of the array, which is now size *n – 2*.

Because each pass of the fist *while-loop* reduces the size of the array in which we still have things to swap by 2, we're going to need to run at least the first *while-loop* at least **n / 2** times before there are no more swaps. Therefore, the number of comparisons in the first *while-loop* is given by:

$$Number\ of\ comparisons\ (first\ while-loop) = \frac{n}{2}(n-2)$$

In other words, we perform *n – 2* comparisons *n / 2* times.

The second for-loop ranges from 1 to n – 1. Similar to the case above, a comparison is performed at each iteration and so the total number of comparisons performed each time the for-loop runs is (n – 1).

Because we know that the array is already given in reverse sorted order, we know that the worst-case state of the array before the second *while-loop* runs looks something like:

n_1	$n_1 - 1$	n_2	$n_2 - 1$	n_3	$n_3 - 1$

We know this because of the way the numbers "leap-frog" over each other during the swaps, and that larger pairs traverse the list together towards the end of the array. In other words, after the first *while-loop* completes, we have pairs of values in the array that are either in *sorted* or *reverse order*, with the latter state of course corresponding to the worst-case.

In that worst-case, we would have to run the second *while-loop* two times: once to swap all the reversed pairs, and once to check that the array is sorted, no more swaps are needed, and the loop can end.

Therefore, the worst-case number of comparisons for an array in reverse sorted order for the second *while-loop* is given by:

$$Number\ of\ comparisons\ (first\ while-loop) = 2(n-1)$$

Putting it all together, the total number of comparisons carried out by SwapSort2 is given by:

$$Total\ comparisons\ (first\ while-loop) = \frac{n}{2}(n-2) + 2(n-1)$$

- Does your result from above represent the worst-case number of comparisons made by SwapSort2? Justify your answer.

SwapSort2 is essentially just a modification of Bubble Sort. We know from class that the worst-case scenario from bubble sort is when the *smallest* element is last in the array, because then it needs to be swapped all the way to the front of the array in successive passes; i.e. it's the scenario that requires the greatest number of passes. The case where the array is sorted in reverse-order corresponds to that worst-case scenario, so yes, it does represent the worst-case number of comparisons that need to be made overall.

- Justify that the worst-case runtime of SwapSort2 is of the form $T(n) = an^2 + bn + c$ for constants a, b, and c.

Generalizing from the equation for the total number of comparisons carried out at each step above, the number of operations performed by SwapSort2 is given by:

$$T(n) = \frac{n}{2}(n-2) + 2(n-1) = \frac{n^2}{2} - n + 2n - 2 = \frac{1}{2}n^2 + n - 2$$

Therefore, $T(n)$ is of the form $an^2 + bn + c$ where a = ½, b = 1 and c = 2.

c)

The child is carrying out **insertion sort** from class. In insertion sort, we iterate through the array. The portion of the array we've already looped through represents the sorted portion. If we encounter a value larger than the rightmost value in the sorted portion, we continuously swap it to the left until it encounters a value smaller than itself, or we run out of values. Then we proceed again to the right. This is what the child is essentially doing.

d)

- $\boldsymbol{f(n) = n^{1.5}\log 2n + n^2\log(n^2) + \sqrt{n}}$

Note: the dominant term is $n^2 \log(n^2)$, therefore:

$$\textit{Goal: show that } f(n) \leq C \cdot n^2 \log n$$

$$f(n) = n^{1.5}\log 2n + n^2\log(n^2) + \sqrt{n}$$

$$= n^{1.5}(\log 2 + \log n) + 2n^2 \log n + n^{0.5}$$

$$= n^{1.5}(1 + \log n) + 2n^2 \log n + n^{0.5}$$

$$= n^{1.5} + n^{1.5}\log n + 2n^2 \log n + n^{0.5}$$

$$\leq n^2 + n^{1.5}\log n + 2n^2 \log n + n^{0.5} \mid since\ n^{1.5} \leq n^2\ for\ n \geq 0$$

$$\leq n^2 \log n + n^{1.5}\log n + 2n^2 \log n + n^{0.5} \mid since\ n^2 \leq n^2 \log n\ for\ n \geq 0$$

$$\leq n^2 \log n + n^2 \log n + 2n^2 \log n + n^{0.5} \mid since\ n^{1.5}\log n \leq n^2 \log n\ for\ n \geq 0$$

$$\leq n^2 \log n + n^2 \log n + 2n^2 \log n + n^2 \mid since\ n^{0.5} \leq n^2\ for\ n \geq 0$$

$$\leq n^2 \log n + n^2 \log n + 2n^2 \log n + n^2 \log n \mid since\ n^2 \leq n^2 \log n\ for\ n \geq 0$$

$$\leq 5n^2 \log n$$

Therefore, $f(n) \leq 5n^2 \log n$ *where* $C = 5$, so $f(n)$ is $O(n^2 \log n)$.

$$f(n) = n^{1.5}\log 2n + n^2\log(n^2) + \sqrt{n} \geq n^2\log(n^2)$$

Therefore, $f(n)$ is $\Omega(n^2 \log n)$.

- $\boldsymbol{f(n) = 2^n \cdot n^2 + 100 \cdot 3^n + n^4}$

Note: the dominant term is 3^n, therefore:

$$\textit{Goal: show that that } f(n) \leq C \cdot 3^n$$

$$f(n) = 2^n \cdot n^2 + 100 \cdot 3^n + n^4$$

$$\leq 2^n \cdot n^2 + 100 \cdot 3^n + 3^n \mid since\ n^4 \leq 3^n\ for\ n \geq 7$$

$$\leq 2^n \cdot 1.5^n + 100 \cdot 3^n + 3^n \mid since\ n^2 \leq 1.5^n\ for\ n \geq 13$$

$$= (2 * 1.5)^n + 100 \cdot 3^n + 3^n \mid since\ a^b c^b = (a \cdot c)^n$$

$$\leq 3^n + 100 \cdot 3^n + 3^n$$

$$= 102 \cdot 3^n$$

Therefore, $f(n) \leq 102 \cdot 3^n$ where $C = 102$, so $f(n)$ is $O(3^n)$.

$$f(n) = 2^n \cdot n^2 + 100 \cdot 3^n + n^4 \geq 3^n$$

Therefore, $f(n)$ is $\Omega(3^n)$.

Question 2: Recurrences

a)

Practice1

Examining *Practice1* first: essentially what the algorithm is doing during each recursive call is running a binary search on half of the array, then passing the other half of the array into a subsequent recursive call. Along the way it performs some constant-time operations such as comparisons, calculations, variable assignments and return statements.

The main functional work of each recursive call is performing the binary search, which we know from class takes $\log n$ work to complete.

We can describe *Practice1* using the recurrence relation:

$$T(n) = T\left(\frac{n}{2}\right) + c \log n$$

Because it is in the appropriate form, we can use the Master Method to determine the runtime for the given relation using the constants:

$$a = 1, b = 2$$

$$k = \log_b a = \log_2 1 = 0$$

$$n^k = n^0 = 1$$

We therefore have $f(n) = \log n \geq n^k = 1$. Because $f(n)$ is asymptotically larger than n^k, our recurrence relations falls into Case 3 of the Master method, from which is follows that $T(n)$ is $\Theta\big(f(n)\big) = \Theta(\log n)$.

Practice2

In this algorithm, we recursively perform binary search on arrays sequences decreasing by 1 with each recursive call.

The worst-case scenario corresponds to the case when all binary searches are unsuccessful. Because we are decreasing the array by one element with each recursive call, we end up making n recursive calls in total, and perform a binary search for (n -1) calls (because the final call hits the base-case and returns immediately. Therefore the total work done is $(n-1) + c \log n$.

The recurrence relation for this algorithm is given by:

$$T(n) = T(n-1) + c \log\ n$$

We want to show that $T(n)\ is\ \theta(n \log n)$.

Goal: show that $T(n) \leq cn \log n$

For the sake of induction, let us assume that:

$$T(n-1) \leq d\ (n-1) \log(n-1)$$

Next, we can use our induction hypothesis to make a substation in the original recurrence relation:

$$T(n) = T(n-1) + c \log\ n$$

$$\leq\ d\ (n-1)\log(n-1) + c\log n$$

$$= (dn - d)\log(n-1) + c\log n$$

$$= dn\log(n-1) - d\log(n-1)\ + c\log n$$

$$= dn\log(n-1) - (d\log(n-1) - c\log n)$$

$$\leq\ n\log n$$

$$when\ d > c\ such\ that\ d\log(n-1) - c\log n\ \ is\ positive.$$

When d is larger than c such that the subtractive term is positive, we are reducing $dn\log(n-1)$, so that it ends up being asymptotically smaller than $n\log n$.

Therefore, $T(n)\ is\ \theta(n\log n)$

b)

$$\boldsymbol{T(n) = T\left(\frac{n}{3}\right) + n \log n}$$

$$a = 1, b = 3, k = \log_b a = \log_3 1 = 0$$

$$n^k = n^0 = 1$$

$$f(n) = n \log n > n^k = 1$$

Because $f(n)$ is asymptotically larger than n^k we have Case 3 in the Master Method. $f(n)$ is $\Omega(n^{0+e})$ and $T(n)$ is $\theta\big(f(n)\big) = \theta(n \log n)$.

$$\boldsymbol{T(n) = 16T\left(\frac{n}{4}\right) + n^{1.5} \log n}$$

$$a = 16, b = 4, k = \log_b a = \log_4 16 = 2$$

$$n^k = n^2$$

Because we know that polynomials terms dominate in both $f(n)$ and n^k we can compare the functions directly and see that n^k is asymptotically larger than $f(n)$. Therefore we have Case 1 of the Master Method, when $f(n)$ is $O(n^{2-e})$ and $T(n)$ is $\theta\big(n^k\big) = \theta(n^2)$.

$$\boldsymbol{T(n) = 4T\left(\frac{n}{16}\right) + \sqrt{n}}$$

$$a = 4, b = 16, k = \log_b a = \log_{16} 4 = 0.5$$

$$n^k = n^{0.5}$$

We see that n^k is asymptotically the same than $f(n)$ because $n^{0.5}$ is equal to $n^{0.5}$. Therefore we have Case 2 of the Master Method, when $f(n)$ is $O(n^k \log n)$ and $T(n)$ is $\theta\big(\sqrt{n} \log n\big)$.

c)

```
FindK(A, s, f, k)
        if s < f
                if f = s + 1
                        if k = A[s] return s
                        if k = A[f] return f
                        return -1
                else
                        q1 = ⌊(2s + f)/3⌋
                        q2 = ⌊(q1 + 1 + f)/3⌋
                        r1 = FindK(A, s, q1, k)
                        r2 = FindK(A, q1 + 1, q2, k)
                        r3 = FindK(A, q2 + 1, f, k)
                        if (r1 > -1) return r1
                        if (r2 > -1) return r2
                        if (r3 > -1) return r3
                        return -1
        else
                if s = f and k = A[s]
                        return s
                else
                        return -1;
```

- Write and justify a recurrence for the runtime of $T(n)$ of the above algorithm.

In the algorithm above, we make **3 successive recurrent calls**. Each call operates on roughly **1/3 of the size of the input array, or n/3**. The rest of the operations all take **constant time**, including comparisons, mathematical operations such as addition and division, return statements, etc. So the two components contributing to the runtime of the algorithm are the 3 recursive calls, and the sum of the constant-time operations which can be lumped together in a single constant term.

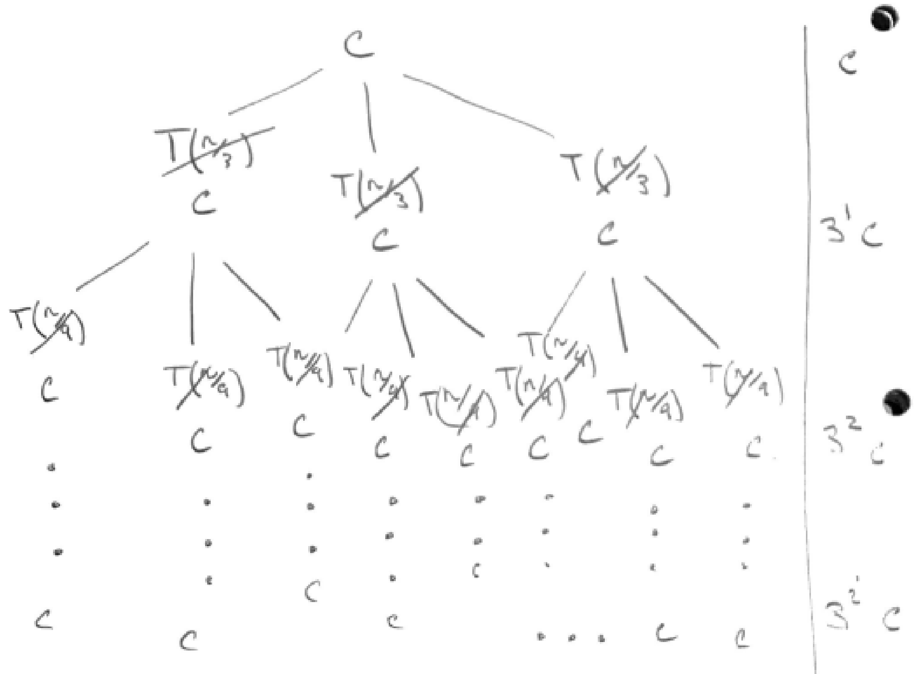
The final recurrence relation for the algorithm above should be:

$$T(n) = 3T\left(\frac{n}{3}\right) + c$$

The 3 in front of $T\left(\frac{n}{3}\right)$ indicates that we are making the recursive call three times, and the $\frac{n}{3}$ indicates that each recursive call operates on about 1/3 of the input array, as discussed.

- Use the recursion tree to show that the algorithm runs in time $O(n)$.

Recursion Tree for: $T(n) = 3T\left(\frac{n}{3}\right) + c$



$$T(n) = 3^1c + 3^2c + 3^3c + 3^4c + \dots 3^i c$$

$$= c\sum_{i=0}^{\log_3(n)} 3^i$$

$$= 3^1c + 3^2c + 3^3c + \dots + 3^{\log_3 n}c$$

$$= \underbrace{3^1c + 3c + 3^3c + \dots +}_{d} nc$$

$\therefore\ T(n) = cn + d$, so $T(n)$ is $O(n)$

Question 3: Hashing

a)

Final Result

0	1	2	3	4	5	6	7	8	9	10	11	12
22		12	16	37	18	19	7		29		50	38

Steps:

0	1	2	3	4	5	6	7	8	9	10	11	12

Insertion Order: 38, 16, 50, 7, 18, 19, 12, 37, 29, 22

Insert 38:

$$38\ mod\ 13 = 12$$

0	1	2	3	4	5	6	7	8	9	10	11	12
												38

Insertion Order: 16, 50, 7, 18, 19, 12, 37, 29, 22

Insert 16:

$$16\ mod\ 13 = 3$$

0	1	2	3	4	5	6	7	8	9	10	11	12
			16									38

Insertion Order: 50, 7, 18, 19, 12, 37, 29, 22

Insert 50:

$$50\ mod\ 13 = 11$$

0	1	2	3	4	5	6	7	8	9	10	11	12
			16								50	38

Insertion Order: 7, 18, 19, 12, 37, 29, 22

Insert 7:

$$7\ mod\ 13 = 7$$

0	1	2	3	4	5	6	7	8	9	10	11	12
			16				7				50	38

Insertion Order: 18, 19, 12, 37, 29, 22

Insert 18:

$$18\ mod\ 13 = 5$$

0	1	2	3	4	5	6	7	8	9	10	11	12
			16		18		7				50	38

Insertion Order: 19, 12, 37, 29, 22

Insert 19:

$$19\ mod\ 13 = 6$$

0	1	2	3	4	5	6	7	8	9	10	11	12
			16		18	19	7				50	38

Insertion Order: 12, 37, 29, 22

Insert 12:

$$12\ mod\ 13 = 12$$

Probe Attempt #1:

$$h(12,1) = \left(12 + 4(1) + 2(1^2)\right)\ mod\ 13 = 18\ mod\ 13 = 5$$

Probe Attempt #2:

$$h(12,2) = \left(12 + 4(2) + 2(2^2)\right)\ mod\ 13 = 28\ mod\ 13 = 2$$

0	1	2	3	4	5	6	7	8	9	10	11	12
		12	16		18	19	7				50	38

Insertion Order: 37, 29, 22

Insert 37:

$$37\ mod\ 13 = 11$$

Probe Attempt #1:

$$h(37,1) = \left(37 + 4(1) + 2(1^2)\right)\ mod\ 13 = 43\ mod\ 13 = 4$$

0	1	2	3	4	5	6	7	8	9	10	11	12
		12	16	37	18	19	7				50	38

Insertion Order: 29, 22

Insert 29:

$$29\ mod\ 13 = 3$$

Probe Attempt #1:

$$h(29, 1) = \left(29 + 4(1) + 2(1^2)\right)\ mod\ 13 = 35\ mod\ 13 = 9$$

0	1	2	3	4	5	6	7	8	9	10	11	12
		12	16	37	18	19	7		29		50	38

Insertion Order: 22

Insert 22:

$$22\ mod\ 13 = 11$$

Probe Attempt #1:

$$h(22, 1) = \left(22 + 4(1) + 2(1^2)\right)\ mod\ 13 = 28\ mod\ 13 = 9$$

Probe Attempt #2:

$$h(22, 2) = \left(22 + 4(2) + 2(2^2)\right)\ mod\ 13 = 38\ mod\ 13 = 12$$

Probe Attempt #3:

$$h(22, 3) = \left(22 + 4(3) + 2(3^2)\right)\ mod\ 13 = 52\ mod\ 13 = 0$$

Final Result

0	1	2	3	4	5	6	7	8	9	10	11	12
22		12	16	37	18	19	7		29		50	38

b)

```
HashInsert(T, k)
        m = k mod 13
        if (T[m] = nil)
                T[m] = k
                return true
        else
                for i = 1 to 13
                        m = (k + (4 * i) + (2 * i * i)) mod 13
                        if (T[m] = nil)
                                T[m] = k
                                return true
                return false
```

c)

Assume that the arrays A and B have sizes (i.e. *n*) large enough to contain all possible 3-digit PIN number combinations. In other words, they are large enough to contains values from 000 through 999, with each PIN number occupying one cell of each array.

Assume also that array B uses a simplistic hash implementation where the hash function is the *identity function*. In other words, pin 555 would be stored at index 555 in the array; pin 002 would be stored in index 2, etc.

Assume that array A is a regular array where the numbers are entered as they appear (i.e. in no particular order, without gaps).

The algorithm to output all safe passwords (i.e. passwords that *are not in* Bank B) from bank A would therefore be:

```
FindSafePins(A, B, s, f)
        for i = s to f                              (0 to n)
                pinA = A[i]                         (constant)
                if (B[pinA] = nil) // not in bank B (constant)
                        print(pinA)                 (constant)
```

In the algorithm above we are looping over the entire length of array A once. For each iteration of the loop, we perform a constant amount of work. This is because accessing an element of an array is a constant-time operation, we perform two array accesses: once for A to get the PIN from A, and once in B to see if B contains the same PIN as A. Finally, there is the print operation to output the PIN from A if B does not contain the PIN in A.

In the worst-case array A is completely filled, i.e. it has size *n*. Therefore we can describe the runtime of the algorithm as:

$$T(n) = an + b$$

Where *n* represents the number of iterations, and *a* represents the constant work performed per operation, and *b* represents any additional constant work associated with invoking the function.

$$\therefore\ f(n)\ is\ O(n)$$

Question 4: Selection

a)

Target Rank: $k = 19$

56	78	34	19	67	32	13	12	90	92	50	51	30	1	99	58	43	42	24	65	21	25	68	69	**101**

1) Divide the array into groups of size 5:

56	78	34	19	67	32	13	12	90	92	50	51	30	1	99	58	43	42	24	65	21	25	68	69	**101**

2) Sort each group and find the median of each group:

19	34	56	67	78	12	13	32	90	92	1	30	50	51	99	24	42	43	58	65	21	25	68	69	**101**

3) Find the median of all the medians:

32	43	50	56	68

$x = 56$

4) Partition the elements comparing them to *x*, determining the rank of x along the way,

34	19	32	13	12	30	1	43	42	24	21	25	50	56	78	67	90	92	51	99	58	65	68	69	101
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

5) Rank of 50 is 13. We are looking for 19. Recurse on the *larger* portion of the array.

56	78	67	90	92	51	99	58	65	68	69	101

New target rank = k – x = 19 – 13 = 6.

1) Divide the array into groups of size 5:

56	78	67	90	92	51	99	58	65	68	69	101

2) Sort each group, and find its median:

56	67	78	90	92	51	58	65	68	99	69	101

3) Find the median of the medians, x:

65	78	101

x = 78 (*assuming median of two elements is the second element*)

4) Partition the elements around x, determining the rank of x along the way:

56	67	51	58	65	68	69	78	90	92	99	101

Rank of 78 is 8. The target rank is 6, so recurse on the left subarray.

56	67	51	58	65	68	69

1) Divide into groups of 5

56	67	51	58	65	68	69

2) Sort each group

51	56	58	65	67	68	69

3) Find the median of the medians

58	69

4) Partition around the median

56	67	51	58	65	68	69

Rank of median is 7. We're looking for the rank 6, so recurse on the left subarray.

56	67	51	58	65	68

1) Split array into groups of 5

56	67	51	58	65	68

2) Sort each group, finding the median:

51	56	58	65	67	68

3) Get median of medians

58	68

4) Partition around the median

56	67	51	58	65	68

68 is rank 6. We are looking for rank 6, so return 68.

b)

Target Rank: $k = 19$

Initial Array:

56	78	34	19	67	32	13	12	90	92	50	51	30	1	99	58	43	42	24	65	21	25	68	69	**101**

Pivot: 101. After sorting around 101:

56	78	34	19	67	32	13	12	90	92	50	51	30	1	99	58	43	42	24	65	21	25	68	69	**101**

Pivot rank: 25. Recurse on left subarray:

56	78	34	19	67	32	13	12	90	92	50	51	30	1	99	58	43	42	24	65	21	25	68	**69**

Pivot: 69. After sorting around 69:

56	34	19	67	32	13	12	50	51	30	1	58	43	42	24	65	21	25	68	**69**	78	90	92	99

Pivot rank: 20. Recurse on left subarray.

56	34	19	67	32	13	12	50	51	30	1	58	43	42	24	65	21	25	**68**

Pivot: 68. After sorting around 68:

56	34	19	67	32	13	12	50	51	30	1	58	43	42	24	65	21	25	**68**

Pivot rank: 19. The pivot rank equals the target rank ($k = 19$) so return 68 as the 19^{th} rank value.

c)

Given an array of size *n*, our goal is to find the quartile of elements *smaller* than a given rank (faster times). In other works, given some rank *r* in the array, we want to find return elements ranked from *r - 1* through *r - n/4* (inclusive). In this case, we are not given the target rank *r*; however, we know the exact value 10.57 from which we can easily derive its rank. (Note: this is assuming that the fastest runner has rank 1, *i.e.* is the smallest value in the array).

Given array of size *n* and value v, the steps are as follows:

- Step 1: Determine the rank *r1* of *v1* using partitioning
    - This is the same partitioning step we've seen in the selection algorithms where we put all values smaller than *v1* on the left, and all values larger than *v1* on the right, thus determining the rank of *v1* itself
- Step 2: Determine the value *v1* of rank *r2*, where *r2 = r1 - n/4* using a linear selection algorithm
    - Note: we can be certain that *r2* is not out of bounds in our array because we are told explicitly that *v1* has *at least* rank *n/2*, since *at least* half of the runners were faster
- Step 3: Now, having both values *v1* and *v2*, return all elements between *v1 - 1 1*and *v2* (inclusive)
- The runtime of each step should be $O(n)$; therefore the entire runtime is expected to be $O(n)$

FasterSprinterQuartile(A, s, f)

```
r1= 1
// determine the rank of 10.57 by counting how many values are less than 10.57
for i = s to f
        if (A[i] < 10.57)
                r1 = r1 + 1
r2 = r1 - (f / 4)
v2 = getKthOrderStatistic(A, s, f, r2)
for i = s to f
        if (A[i] >= v2 and A[i] < 10.57)
                print A[i]
```

In this algorithm we see the main iterative work is performed by 2 for-loops and one call to the selection algorithm. Both for-loops traverse the array once and always run in linear time, $O(n)$. We also know that we can use a selection algorithm that runs in *worst-case* linear time, also $O(n)$. Everything else is constant work.

Putting it all together, we see that $f(n) = 3dn + c$; ignoring the constants we see that it is overall $O(n)$.

Question 5: Heaps

a)

```
VerifyHeap(A)  // take in heap as parameter
        if (A.heapsize = 1)
                return true  // all leaf nodes are valid heaps
        currNode = A[1]
        leftChild = nil
        if (2 * i < A.heapsize)
                leftChild = A[floor(2 *i)]
        rightChild = nil
        if (2 * i + 1 < A.heapsize)
                rightChild = A[floor(2 * i + 1)]
        // check if either the left or right child violate the heap condition
        if (leftChild != nil AND leftChild > currNode)
                return false
        if (rightChild != nil AND rightChild > currNode)
                return false
        // make recursive calls on the left and right subheaps
        leftHeapIsValid = leftChild = nil OR VerifyHeap(getLeftSubHeap(A))
        rightHeapIsValid = rightChild = nil OR VerifyHeap(getRightSubHeap(A))

        return (leftHeapIsValid AND rightHeapIsValid)
```

- Recurrence Relation

$$T(n) = 2T\left(\frac{n}{2}\right) + c$$

In the algorithm above we see that there are two recursive calls: one on the *left sub-heap* of the current node, and one on the *right sub-heap* of the current node. Therefore, we needed to include the runtime of *2* recursive calls. Moreover, each recursive call operates on a heap of *roughly half the size* of the original heap, since we are moving down half of the binary tree with each recursive call. Therefore, the recursive work is performed on *n /2* elements from the original array.

The rest of the work is constant: performing mathematical operations, comparisons, assignments, return statements, etc. This is all added as a constant term *c* in the recurrence relation.

- *Worst-case Runtime Analysis*

The worst case runtime corresponds to the scenario where the heap *is* valid and every level of the heap is complete. In that case, we will need to check every single node in the heap in order to be certain that the heap really is correct.

Intuitively, we know that if we have an array of size *n* and we have to look at every element in the array *exactly once*, then we are going to perform *n* iterations. This gives us a runtime of $f(n) = O(n)$.

We can verify this using the Master Method on the recurrence relation above:

$$T(n) = 2T\left(\frac{n}{2}\right) + c$$

$$a = 2 \qquad b = 2$$
$$k = \log_b a = \log_2 2 = 1$$
$$n^k = n^1 = n$$
$$f(n) = c$$

$$Because\ n^k\ is\ asymptotically\ larger\ than\ f(n)\ where\ f(n) is\ O(n^{1-e)})$$

$$T(n)\ is\ \Theta\left(n^k\right) = \Theta(n)\ and\ therefore\ O(n)$$

- *Best-case Runtime Analysis*

The best-case scenario is where the entire heap is composed of just one node. That means that the root is also a leaf node and we know that all leaf nodes are valid heaps. In that case, the functions returns immediately and the runtime is constant or $O(1)$.

b)

We have a situation where a node is missing somewhere in heap A. Any node in a heap is the root node of some subheap. Therefore, the missing node is also the root of an associated subtree. In order to repair the subtree, all we need to do is to delete the missing node following the same algorithm we would use to delete the maximum of the entire heap. Specifically, we're going to swap the missing node with the last element of the subheap and bubble it up. Since the position of the missing element, *k*, is already given, and we don't have to go searching for it, the operation will run in $O(\log n)$ time.

Note: I am assuming that the missing node *is not* the last node in the heap. In that special case, all we would have to do is decrement the size of the heap by 1 and do no further work. However, I don't think that was the point of the problem so I am ignoring this edge-case in the pseudocode below.

RepairHeap(A, k)

```
last = A.heapsize
Swap A[k] and A[last]
Bubble-down(A, k)
A.heapsize = A.heapsize - 1
```

- *Worst-case Runtime Analysis*

The worst case scenario corresponds to the case when the missing element is at the root position. This is because we may have to bubble-down the swapped node all the way down the height of the tree. We already know that the Bubble-down algorithm runs in $O(\log n)$ time, and all other work down by this algorithm is *constant time*. Therefore the entire algorithm is worst-case $O(\log n)$.

- *Best-case Runtime Analysis*

The best-case scenario is when the missing node *k* is a leaf node, or when the missing element only has a single child. In those cases, Bubble-down will return immediately without performing any more work, and all the other operations run in constant time.

Therefore, in the best-case scenario the runtime is $O(1)$.

c)

Intuitively we know that for a valid minheap, the smallest element of the entire heap is at the root position. We have a min heap for each of the 5 heats. Each heat's minheap contains the fastest time for the heat in the root position. That is, the minimum of a given heap is the fastest time *for that heat*. The minimums of the 5 heaps therefore represent the *fastest times across all 5 heats*. Therefore, the minimum of those minimums represents the *fastest time across all heaps*.

To find the 40 fastest times across all heaps them, all we need to do is:

1) Find the minimum of minimums, i.e. the minimum of the roots of all 5 heaps.
2) Output that minimum value as the current fastest time across all heats.
3) Delete that minimum value from the heap from which it came.
4) Repeat steps 1-3 another 39 times.

Top40(H1, H2, H3, H4, H5)

```
for i = 1 to 40
        minTime = H1[1]
        minTimeHeap = H1
        if (H2[1] < minTime)
                minTime = H2[1]
                minTimeHeap = H2
        if (H3[1] < minTime)
                minTime = H3[1]
                minTimeHeap = H3
        if (H4[1] < minTime)
                minTime = H4[1]
                minTimeHeap = H4
        if (H5[1] < minTime)
                minTime = H5[1]
                minTimeHeap = H5
        delete(minTimeHeap, 1)  // delete the current minimum (root) from its associate heap
        print(minTime)
```

- Runtime Justification:

In the algorithm above, we carry out exactly 40 iterations. In each iterations, we perform a series of constant time comparisons and assignments to figure out which of the 5 heaps contains the current fastest time of all of them. This all takes constant time.

Then, at the end, before moving on to the next iteration, we remove the current fastest time from its heap. We know from class that deleting the minimum element of a minheap takes $O(\log n)$ time.

Therefore, the work done by out algorithm can be expressed as:

$$f(n) = 40(\log n + c) = 40\log(n) + d$$

We can then see that $f(n)$ is $O(\log n)$.

Question 6: Lower Bounds and Linear Time Sorting

a)

Counting Sort: we know that the runtime of counting sort is typically given by $O(n + k)$ where n represents the input of numbers to sort, and *k* is the maximum integer of the input.

In this case, we are told we have an input of *n* natural numbers (that is, 1, 2, 3, …) that range from *0* to $\sqrt{n}$. Therefore *k* is equal to $\sqrt{n}$ and the total runtime is:

$$O\left(n + \sqrt{n}\right) \text{ or simply, } O(n) \text{ because } n \geq \sqrt{n}$$

However, we can be more explicit in working out the runtime. Counting sort requires 3 steps to run.

Step 1: loops over the input elements and counts the occurrence of each element, placing the count into a separate array, C. This step takes $O(n)$ times to run because we have to count each input element.

Step 2: loops over array C. C contains the count of all original elements; the size of C is $\sqrt{n}$ because we know that that is the maximum value. This step takes time $O\left(\sqrt{n}\right)$.

Step 3: loops over the original set of elements again, this time using the values in array C to place them into final position in a new output array. This step also takes $O(n)$ time.

Altogether the time taken is $O\left(2n + \sqrt{n}\right)$ which turn simplifies to $O\left(n + \sqrt{n}\right)$ or simply, $O(n)$ because $n \geq \sqrt{n}$, as stated above.

Radix Sort:

The runtime of radix sort is determined by the size of the input array, the maximum number of possible digits *d*, and the number of types of digits, *r*. Taken together the runtime is $O(d(n + r))$.

Because we are dealing with natural numbers, the possible values for a given digit are 0 through 9, which means *r* = 10.

We also know that the maximum possible number of digits for a given group of numbers is given by $\log(n)$. We also know that the maximum possible value is $\sqrt{n}$, so the number of digits necessary is at most $\log_{10}\sqrt{n} = \log_{10} n^{0.5} = 0.5\log_{10} n$, or just $\log_{10} n$.

Therefore, the runtime for radix sort is $O\left(d(n + r)\right) = O(0.5\log_{10} n(n + 10)) = O(0.5n\log_{10} n + 5\log_{10} n)$. This ultimately simplifies to $O(n\log n)$.

Bucket Sort:

We are running bucket sort using 10 buckets. Because we have 10 elements, and all the elements are uniformly distributed, each bucket stores $n/10$ elements. Using insertion sort, the time it takes to sort each bucket will be $O(e_i^2) = O\left(\frac{n^2}{100}\right) = O(n^2)$.

The first step of bucket sort is to scatter all elements across the 10 buckets. Because we have *n* elements, this will take time $O(n)$.

The second step of bucket sort is to go through and sort each bucket. Because we have exactly 10 buckets, and we've seen from above that the time taken to sort each bucket is $O(n^2)$, the total time taken for this step is $O(10n^2)$. We can drop the constant to have just $O(n^2)$.

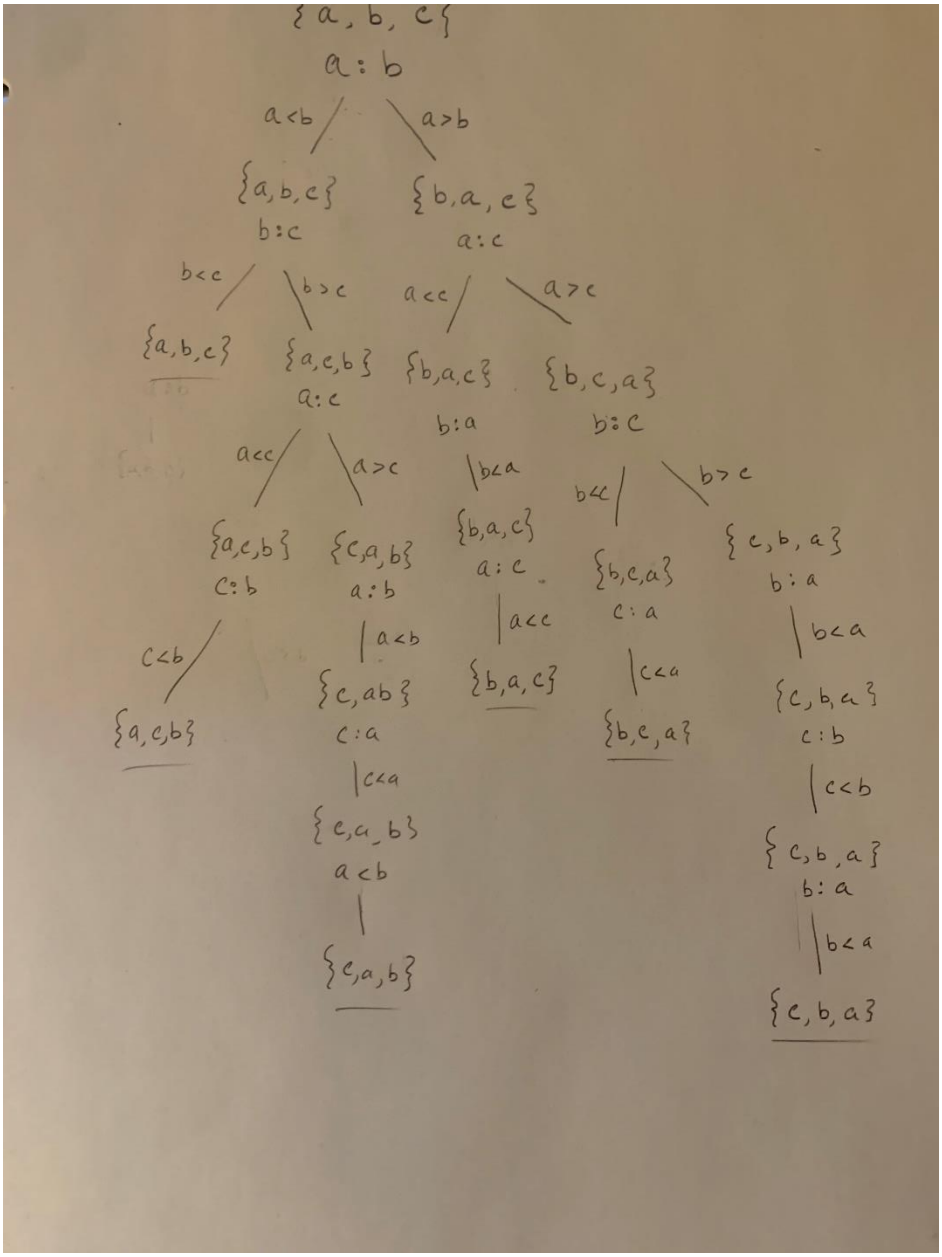
Finally the last gather step is to go through each bucket and output the elements. Because we will have to output every element, this step also takes time $O(n)$.

Altogether the steps take time $O(2n + n^2)$ which simplifies to $O(n^2)$.

We see that **counting sort** has best runtime.

b)

Decision Tree:



The longest path of the bubble sort decision tree corresponds to the worst-case runtime of the algorithm. The worst-case runtime for bubble sort occurs when all the elements are sorted in reverse order. Each pass of the algorithm places exactly one element in its correct position (the largest element in the back of the array). That means that in the worst case, 2 passes are required to position all the

elements into their proper location, along with 1 final pass to check that the array is sorted. Altogether, this means 3 passes, with $n - 1$, or 2, comparisons on each pass, for a total of 6 comparisons. That is why we see the longest branch having a length of 6 in the decision tree above.