COMP 3804 Assignment 1:

1) Dulika Gamage 101263208

2) Is FIBBEIBER correct?:

Proof by Induction:

→ Base cases:

→ Base case for $n=0$:

- FibBeiber(0) initializes the array so $f(0)=-1$
- FibBeiber(0) calls Beiber(0)
- Beiber(0) makes $m=0$, then checks if $m=0$ and since it is, it sets $f(0)=0$
- FibBeiber(0) then returns $f(0)=0$, which is equal to $F_0=0$

→ Base case for $n=1$:

- FibBeiber(1) inits the array $f(0)=-1$, $f(1)=-1$
- FibBeiber(1) calls Beiber(1)
- Beiber(1) makes $m=1$, then checks if $m=0$, since it isn't, it checks $m=1$ and since it is it sets $f(0)=0$ and $f(1)=1$
- FibBeiber(1) then returns $f(1)=1$, which is equal to $F_1=1$

→ Inductive step:

hyp:

Assume the algorithm works for $n=k$ ⇒ Assume it computes the correct fibonacci number for $0 \le n \le k$.

For each $n \le k$, the algorithm computes F_n. We need to show $F_{k+1}=f(k+1)$ when $n=k+1$

Case: $n=m=k+1$

- FibBeiber(k+1) inits an array $f(k+1)$ where all elements are set to -1.
- Beiber(k+1) is called:
- checks $f(k-1)$ and $f(k)$ are already computed (not -1). If they aren't computed it recursively calls Beiber(k) and Beiber(k-1)
- after $f(k-1)$ and $f(k)$ are computed it computes: $f(k+1)=f(k-1)+f(k)$
- which is equal to $F_{k+1}=F_{k-1}+F_k$ (by definition of fibonacci sequence)

$\boxed{\therefore f(k+1)=F_{k+1} \rightarrow \text{the inductive step holds, and FibBeiber(n) correctly computes the } n^{th} \text{ fibonacci number.}}$

2) What's the runtime of FibBeiber(n)

- init array: $O(n)$
- the algorithm ensures each value is only computed once

↳ the # calls to Beiber(n) is proportional to n (because of this)

↳ each call to Beiber(n) is constant amount of work

$\boxed{\therefore \text{the runtime is } O(n)}$

(we saw this example in class)

↳ (w/ memoization)

3) Is FibSwift correct?

Proof by induction:

→ Base Cases:

→ Base Case for $n=0$

- inits the array so $f(0) = -1$
- FibSwift(0) calls Swift(0)
- swift(0) makes $m=0$, then checks if $m=0$ and since it is it sets $f(0)=0$
- FibSwift then returns 0 because $f(0)=0$

→ Base Case for $n=1$

- inits the array so $f(0)=-1$ and $f(1)=-1$
- FibSwift(1) calls Swift(1)
- swift(1) makes $m=1$, then checks if $m=0$, since it isnt it checks $m=1$, since it is it makes $f(0)=0$ and $f(1)=1$
- FibSwift then returns 1 because $f(1)=1$

→ Inductive Step:

hyp:
Assume the algorithm works for $n=k \Rightarrow$ Assume it computes the correct fibonacci number for $0 \leq n \leq k$.

For each $n \leq k$, the algorithm computes F_n. We need to show $F_{k+1} = f(k+1)$ when $n=k+1$

Case: $n = m = k+1$

- FibSwift(k+1) inits an array $f(k+1)$ where all elements are set to -1.
- Swift(k+1) is called:
- for $k+1 \geq 2$, Swift recursively calls Swift(k)
- after $f(k)$ (and ∴ Swift(k-1) since it's recursive) is computed it calculates $f(k+1) = f(k) + f(k-1)$
- which is equal to $F_{k+1} = F_{k-1} + F_k$ (by definition of fibonacci sequence)

$\boxed{\therefore\ f(k+1) = F_{k+1} \rightarrow \text{the inductive step holds, and FibSwift(n) correctly computes the } n^{th} \text{ fibonacci number.}}$

3) What's the runtime of FibSwift?

- init array: $O(n)$
- the algorithm ensures each value is only computed once (like FibBeiber)
  - ↳ the # calls to FibSwift(n) is proportional to n (because of this)
  - ↳ each call to FibSwift(n) is constant amount of work

$\boxed{\therefore \text{ the runtime is } O(n)}$

(we saw this example in class)
↳ (w/ memorization)

4) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ n^3 + 12 \cdot T(\frac{n}{7}) & \text{if } n \geq 7 \end{cases}$

Unfolding: $7^k = n$

$T(n) = n^3 + 12 \cdot T(n/7)$

$\leq n^3 + 12 \cdot \left((\frac{n}{7})^3 + 12 \cdot T(n/7^2) \right) \rightarrow T(n/7)$

$= n^3 + 12 \cdot n^3/7^3 + 12^2 \cdot T(n/7^2)$

$\leq n^3 + 12 \cdot \frac{n^3}{7^3} + 12^2 \cdot \left((n/7^2)^3 + 12 \cdot T(n/7^3) \right) \rightarrow T(n/7^2)$

$= n^3 + 12 \cdot \frac{n^3}{7^3} + 12^2 \cdot n^3/7^6 + 12^3 \cdot T(n/7^3)$

$\leq n^3 + 12 \cdot \frac{n^3}{7^3} + 12^2 \cdot n^3/7^6 + 12^3 \cdot \left((n/7^3)^3 + 12 \cdot T(n/7^4) \right) \rightarrow T(n/7^3)$

$= n^3 + 12 \cdot n^3/7^3 + 12^2 \cdot n^3/7^6 + 12^3 \cdot n^3/7^9 + 12^4 \cdot T(n/7^4)$

$= \left(1 + \frac{12}{7^3} + \frac{12^2}{7^6} + \frac{12^3}{7^9} \right) n^3 + 12^4 \cdot T(n/7^4)$

$= \left(1 + (\frac{12}{7^3})^1 + (12/7^3)^2 + (12/7^3)^3 \right) n^3 + 12^4 \cdot T(n/7^4)$

.
.
.

$= \underbrace{\left(1 + (12/7^3)^1 + (12/7^3)^2 + (12/7^3)^3 + \ldots + (12/7^3)^{k-1} \right)}_{\frac{(\frac{12}{7^3})^k - 1}{\frac{12}{7^3} - 1}} \cdot \underbrace{n^3}_{n^3 = 7^{3k}} + 12^k \cdot \underbrace{T(n/7^k)}_{T(1) = 1}$

$= \left(\frac{(\frac{12}{7^3})^k - 1}{\frac{12}{7^3} - 1} \right) \cdot n^3 + 12^k$

$\left[\begin{array}{l} 7^k = n \\ k = \log_7 n \end{array} \right.$

$T(n) \leq \left(\frac{(\frac{12}{7^3})^{\log_7 n} - 1}{\frac{12}{7^3} - 1} \right) \cdot n^3 + 12^{\log_7 n}$

$\boxed{\text{Since } n^3 > 12^{\log_7 n} \text{ we can say } T(n) = O(n^3)}$

4) Master Theorem

$T(n) = a \cdot (\frac{n}{b}) + n^d \rightarrow a = 12, \ b = 7, \ d = 3$

$\log_7 12 = \frac{\log 12}{\log 7} \approx 1.277$ $\quad \boxed{3 > 1.277 \ \therefore \ T(n) = O(n^3)}$

5) Algorithm in $O(\log n)$ time

→ getLargest logic: → assume index starts at 1 (not 0)

Base case:

- $n=1$: return $A[1]$

Recursive: $(n \geq 2)$

- let ℓ = left index & r = right index (starts as $\ell = 1$ $r = n$)
- find the middle of the 2 indeces $m = \lfloor (\ell + r)/2 \rfloor$
- check $A[m] < A[m+1]$

↳ if yes, call alg. with $m+1 = \ell$

↳ if no, call alg. with $m = r$

example runthrough:

A = 1, 2, 3, 4, 5, 6, 0

1 2 3 4 5 6 7

$\ell = 1$, $r = n = 7$, $m = 4$

$A[4] < A[5]$ → yes

$\ell = 5$, $r = 7$, $m = 6$

$A[6] < A[7]$ → no

$\ell = 5$, $r = 6$, $m = 5$

$A[5] < A[6]$ → yes

$\ell = 6$, $r = 6$

$\ell == r$ ∴ largest $A[6] = 6$

→ Pseudo code:

```
getLargest(A, l, r):
if l==r: return A[l];
if n≥2:
    m= floor((l+r)/2);
    if A[m] < A[m+1]: return getLargest(A, m+1, r);
    else: return getLargest(A, l, m);
```

→ Correctness: Proof by Induction

→ Base case: $l == r$ then $A[\ell]$ is returned because there's only 1 element.

→ Inductive step: hyp: The alg will return the largest element in any subarray $[\ell, \dots, r]$

- the array property ensures we only have 1 largest number
- the alg. compares $A[m]$ (middle of subarray) with $A[m+1]$ (element to the right of the middle) at each rec. call.

↳ if $A[m] < A[m+1]$ we know the largest num must be on the right of $A[m]$ because of the array property.

↳ if $A[m] > A[m+1]$ we know it must be on the left of $A[m]$ because of the array property.

↳ the alg. always halves the subarray at each call

↳ this reduces the subarray until there is only 1 element left (base case)

∴ We know that since the alg takes the largest element of the subarray at each step & reduces until there's 1 element, the last element will be the largest element in the array.

→ Time Complexity:

work per step when $n \geq 2$: - calculate m: addition - constant $O(1)$

- $A[m] < A[m+1]$: comparison: constant $O(1)$
- make recursive call: constant $O(1)$

recursive call: each time we do the recursive call we half n so $\frac{n}{2}$

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

Master Theorem: $a = 1$, $b = 2$, $d = 0$

$\log_2 1 = 0$

$d = 0 = 0$

$$\boxed{\therefore \; T(n) = O(1 \cdot \log n) = O(\log n)}$$

6) Algorithm in O(nlogn) time:

→ Base Case: (n ≤ 1)

- n ≤ 1, return 0, because no inversions on 0 or 1 elements.

→ Recursive: (n ≥ 2)

- split array into halves & rec. count out-of-order pairs in each half
- merge two sorted halves & count inversions

↳ count when element from right half is smaller than left half → then that element & every left upcoming element have inversions

because they're all greater & appear before the element from the right

→ Pseudocode:

```
InversionCount(A):
    n = len(A);
    temp_A = [0] * n;
    return mergeCount(A, temp_A, 0, n-1);

mergeCount(A, temp_A, l, r):
    if l == r: return 0;
    else:
        m = floor((l+r)/2);
        count = 0;
        count += mergeCount(A, temp_A, l, m);
        count += mergeCount(A, temp_A, m+1, r);
        count += merge(A, temp_A, l, m, r);
        return count;

merge(A, temp_A, l, m, r):
    count = 0;
    i, k = l;    # index for left subarray (i) / merge (k)
    j = m+1;     # index for right subarray
    while i ≤ m and j ≤ r:
        if A[i] ≤ A[j]:
            temp_A[k] = A[i];
            i++;
        else:
            temp_A[k] = A[j],
            count += (m - i + 1);   # all remaining are greater
            j++;
        k++;
    while i ≤ m:
        temp_A[k] = A[i];
        i++; k++;
    while j ≤ r:
        temp_A[k] = A[j];
        j++; k++;
    for c in range(l, r+1):
        A[c] = temp_A[c];
    return count;
```

b) → Correctness: Proof by Induction:

→ Base Case: $n=1$ $n=0$

- the alg will return 0 because there's no inversions

→ Inductive Step:

hyp:
Assume the alg. correctly counts out-of-order pairs for any array of size k : $k \geq 1$. Prove for $k+1$:

- mergeCount: $A = [a_1, a_2, \ldots, a_{k+1}]$ Left subarray $= [a_1, a_2, \ldots a_{\frac{k+1}{2}}]$ Right subarray $= [a_{\frac{k+1}{2}}, \ldots, a_{k+1}]$
  $L = $ " " $R = $ " "
- mergeCount: by hyp. we assume it correctly counts inversions for K
  ↳ it correctly counts for right & left subarray
- mergeCount: calls merge to merge the subarrays
- merge: compare elements of L & R (subarrays)
  ↳ if $L[i] \leq R[j]$, no inversion
  ↳ if $L[i] > R[j]$, yes inversion because violates the property
    ↳ every remaining element in L will also be greater than $R[j]$ (because sorted), so counts those inversions
    ↱ $L[i] - L[\frac{k+1}{2}]$
- mergeCount: then the inversion count is the sum of inversions in L, inversions in R & inversions between L & R (when merged)

$\boxed{\therefore \text{ by induction, we have proven that the alg. correctly computes inversion for } k+1 \text{ \& therefore } n.}$

→ Time Complexity:

Since we used Merge-sort for the solution, we know from class it's $O(n \log n)$.

- div. into L & R $O(1)$
- we recursively call the alg. on $\frac{n}{2}$ so $T(\frac{n}{2})$
- merging 2 sorted arrays takes $O(n)$

$T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$ $a=2,\ b=2,\ d=1$

$\log_2 2 = 1 = d$ $\boxed{\therefore\ T(n) = O(n \log n)}$

7) Algorithm for finding k-closest elements

→ Base Case: $l = r$:

- subarray is one element, no need to partition further.
- return A[l]

→ Recursive:

Compute Differences.

- calc difference between each number & 2025
- create a list where each element is a pair: ex. D[i] = (| A[i] - 2025 |, A[i]) → first val is difference & second val is the number

Find k-th Smallest Difference Using QuickSelect:

QuickSelect: (comparison-based)

- choose pivot randomly → otherwise may not be O(n), as proven in class
- partition list so nums smaller than pivot go to left and nums larger go right
- recursing one half of array
  - ↳ if pivot is at index k-1, we found the k-th smallest difference
  - ↳ if pivot is too large, we recurse into left
  - ↳ if pivot is too small, we recurse into right

- the k-th smallest val in D gives us a threshold T → k-closest nums are ≤ T
  - ↳ select those until we have k elements
- return the array w/ k elements

Pseudocode:

```
quickselect(A, l, r, k):
    if l==r: return A[l]; # base case
    p = random(l,r) #random so O(n)
    A[p], A[r] = A[r], A[p];
    p_new = partition(A, l, r);
    if k == p_new: return A[k]; # at the pivots pos
    elif k < p_new: return quickselect(A, l, p_new-1, k); #on the left
    else: return quickselect(A, p_new+1, r, k); #on the right

closestK(A, k):
    D = [(|(A[i] -2025)|, i) for i in range(len(A))]; #difference
    quickselect(D, 0, len(D)-1, k-1);
    C = [A[D[i][1]] for i in range(k)]; # k-closest elements
    return C;

partition(A, l, r).
    p = A[r];
    i = l-1;
    for j in range(l, r):
        if A[j] < pivot:
            i++;
            A[i], A[j] = A[j], A[i];
    A[i+1], A[r] = A[r], A[i+1];
    return i+1;
```

7)

→ Correctness: Proof by Induction

→ Base case: $l == r$:

- return $A[l]$ since this is the one and only element

→ Inductive Step

hyp:

Assume quickselect works for subarray of size n where $n < N$ (finds k-th smallest element in any subarray with $< N$ elements)

Prove it works for size N.

- calling quickselect on array w/ size N picks a random pivot between 0 - N and partitions
  - ↳ anything smaller = left of pivot
  - ↳ anything bigger = right of pivot

Case 1: K = pivot:

- pivot is the k-th smallest element & alg returns the pivot.

Case 2: K < pivot:

- recurse left subarray (since k is smaller & smaller elements are stored in the left)
- by inductive hyp., the alg will correctly find k-th smallest element in the left partition.

Case 3: K > pivot:

- recurse right subarray (since k is bigger & bigger elements are stored in the right)
- by inductive hyp., the alg will correctly find k-th smallest element in the right partition

∴ By induction, since each rec. works on a smaller subarray & the alg correctly narrows to the k^{th} smallest element, we can say the alg works on array of size N.

→ Time-Complexity:

- we saw in class that in a "good case" where p = median the runtime = $O(n)$
  - ↳ to get the "good case" use random p, since on average this gives something close to the median

To further prove:

If p = median:

- partitioning takes $O(n)$
- we recurse on a subarray of $n/2$

$T(n) = O(n) + T(n/2)$

Master Theorem: $a = 1$, $b = 2$, $d = 1$

$\log_2 1 = 0 \quad d > 0$

$\boxed{\therefore \ T(n) = O(n)}$

8) $T(n) = 1 + T(\lfloor \sqrt{n} \rfloor)$

↳ you can also write $\sqrt{n}$ as $n^{1/2}$

$T(n) = 1 + T(\lfloor n^{1/2} \rfloor)$ → unfolding

$\leq 1 + [1 + T(\lfloor (\lfloor n^{1/2} \rfloor)^{1/2} \rfloor)$ → $T(\lfloor n^{1/2} \rfloor)$

$= 2 + T(\lfloor (\lfloor n^{1/2} \rfloor)^{1/2} \rfloor)$

$\approx 2 + T(n^{1/2^2})$

$\leq 2 + [1 + T(\lfloor (\lfloor (\lfloor n^{1/2} \rfloor)^{1/2} \rfloor)^{1/2} \rfloor)$ → $T(\lfloor (\lfloor n^{1/2} \rfloor)^{1/2} \rfloor)$

$= 3 + T(\lfloor (\lfloor (\lfloor n^{1/2} \rfloor)^{1/2} \rfloor)^{1/2} \rfloor)$

$\approx 3 + T(n^{1/2^3})$

.

.

.

$= k + T(n^{1/2^k})$

↳ we want to find how many rec. steps (k) it takes to reach 1 (base case)

↳ but since we have the floor function, the recurrence actually terminates when <2 (because anything <2 floors to 1)

$n^{1/2^k} < 2$

$(\frac{1}{2})^k \log n < \log 2$

$(\frac{1}{2})^k < \log 2 / \log n$

$k \log(\frac{1}{2}) < \log\log 2 / \log\log n$ $\qquad \log\log 2 = \log 1 = 0$

$k < 0 - \log\log n / \log(\frac{1}{2})$ $\qquad \log\frac{1}{2} = \log 1 - \log 2 = -1$

$k < -\log\log n / -1$

$k < \log\log n$

$\boxed{\therefore \text{ the recurrence time complexity is } O(\log\log n).}$