1. **Inequality induction** (14 points)

Use induction to prove the following formula, for all integers \( n \geq 1 \). Within your inductive step, be careful to arrange the steps in logical order.

\[
\sum_{k=1}^{n} \frac{1}{\sqrt{k}} > \sqrt{n}
\]

**Solution:**

First, we'll prove a lemma, whose use will be clear when we get into the inductive step.

**Lemma:** \( \frac{1}{\sqrt{j+1}} + \sqrt{j} > \sqrt{j+1} \) for \( j > 2 \)

**Proof of Lemma:** \( \sqrt{j^2 + j + 1} > j + 1 \)

\( \sqrt{j^2 + j + 1} > 0 \), so if we divide both sides with \( \sqrt{j+1} \) we get

\( \frac{\sqrt{j^2 + j + 1}}{\sqrt{j+1}} > \sqrt{j+1} \) The LHS is exactly \( \frac{1}{\sqrt{j+1}} + \sqrt{j} \)

Now, our main proof, by induction on variable \( n \).

**Base Case:** when \( n = 2 \):

\( LHS = 1 + \frac{1}{\sqrt{2}} > \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}} = \sqrt{2} = RHS \)

**IH:** \( \sum_{k=1}^{n} \frac{1}{\sqrt{k}} > \sqrt{n} \) for \( n = 1, 2, \ldots j \) where \( j \geq 1 \)

**IS:** Prove the case when \( n = j+1 \):

\[
\sum_{k=1}^{n} \frac{1}{\sqrt{k}} = \sum_{k=1}^{j+1} \frac{1}{\sqrt{k}} = \frac{1}{\sqrt{j+1}} + \sum_{k=1}^{j} \frac{1}{\sqrt{k}}
\]

By IH, \( \frac{1}{\sqrt{j+1}} + \sum_{k=1}^{j} \frac{1}{\sqrt{k}} > \frac{1}{\sqrt{j+1}} + \sqrt{j} \)

By the lemma, \( \frac{1}{\sqrt{j+1}} + \sqrt{j} > \sqrt{j+1} \). So \( \sum_{k=1}^{j+1} \frac{1}{\sqrt{k}} > \sqrt{j+1} \), which is what we needed to show.
2. A Journey to Michigan (12 points)

Margaret is on a recruiting trip to the University of Michigan. She has $n$ surplus algorithms books on her shelf. She would like to bring $k$ with her, so she can distribute a book to each of the $k$ best theory students at Michigan. However, different algorithm books contain different number of algorithms. To avoid a fight between the students, she would like to pick $k$ algorithm books with the smallest difference in coverage.

Specifically, suppose the $k$ books have $(x_1, x_2, x_3, ..., x_k)$ algorithms in them, where $x_i$ denote the number of algorithms in the $i_{th}$ book, then we define the unfairness as:

$$\max(x_1, x_2, ..., x_k) - \min(x_1, x_2, ..., x_k)$$

To keep the problem simple, we will assume that all the $x_i$ are stored in an array $Array[]$ of size $n$. Our goal is to compute the smallest possible unfairness.

First, consider the algorithm Foo, shown below. findSubsets returns all $k$-element subsets of Array. There are \( \binom{n}{k} \) such subsets, where \( \binom{n}{k} = \frac{n!}{k!(n-k)!} \)

You may assume that the running time of findSubsets is proportional to the number of subsets it returns.

What is the running time of Foo as a function of the input array length $n$? Express your answer in Big-O notation and briefly justify it.

Hint: assume $k$ is a constant which is small compared to $n$.

Solution for Algorithm 1

The running time is $O(n^k)$.

Brief analysis:

First, notice that \( \frac{n!}{m!(n-k)!} = \frac{1}{k} \frac{1}{k!} n(n-1) \ldots (n-k+1) \). If $k$ is small and constant, then \( \frac{1}{k} \frac{1}{k!} \) is a constant and the leading term of $n(n-1) \ldots (n-k+1)$ is $n^k$. So this is $O(n^k)$.

Edge Case Checking : $O(1)$.

Calling findSubsets() : $O\left(\frac{n!}{m!(n-k)!}\right)$, which is $O(n^k)$.

For Loop : In each iteration $O(k)$ operation, total $O\left(\frac{n!}{m!(n-k)!}\right)$ iterations

Checking for return Value and Return : $O(1)$.

So the final running time will mainly depends on the findSubsets() and For loop running time.

Since we assume that $k$ is small and constant, the running time should be $O(n^k)$

2
Algorithm 1 First Attempt

1: function Foo(Array, k )
2:   if $k > \text{length}(\text{Array})$ then
3:     Error “Too few books”
4:   end if
5:   subsets = findSubsets(Array, k )
6:   minFairness = 1000000 \triangleright \text{An absurdly large value}
7:   for each $S$ in Subsets do
8:     min = maxValue
9:     max = 0
10:    for $x_i$ in $S$ do
11:       if $x_i < \text{min}$ then
12:         min = $x_i$
13:       end if
14:       if $x_i > \text{max}$ then
15:         max = $x_i$
16:       end if
17:    end for
18:    currFairness = max - min
19:    if currFairness < minFairness then
20:       minFairness = currFairness
21:    end if
22:   end for
23:   return minFairness
24: end function
3. **Return to Michigan** (12 points)

   One of Robin’s students suggested that we could build a better algorithm using recursion, shown below as Foo2. Suppose that $T(n)$ is the running time of this algorithm on an array of length $n$. Write a recursive definition of $T(n)$ and use it to derive the Big-O running time of this algorithm. Show enough work that we can easily understand how you found your final answer.

```
Algorithm 2 Recursive version
1: function Foo2($x_1, x_2, \ldots, x_n, k$)
2:   if $k > length(Array)$ then
3:     Error “Too few books”
4:   end if
5:   min = 1000000 ▷ An absurdly large value
6:   for $i$ from 1 to $n$ do
7:     for $j$ from 1 to $n$ do
8:       if $x_i \leq x_j$ then
9:         low := $x_i$
10:        high := $x_j$
11:       if $k = \text{NumBetween}(x_1, x_2, \ldots, x_n, \text{low}, \text{high})$ and $min > x_j - x_i$ then
12:         min := $x_j - x_i$
13:       end if
14:     end if
15:   end for
16: end for
17: return min
18: end function

20: function NumBetween($x_1, x_2, \ldots, x_n, l, k$)
21:   if $n \leq 2$ then
22:     rv := 0
23:     for $i$ between 1 and $n$ do
24:       if $l \leq x_i \leq h$ then
25:         rv := rv+1
26:       end if
27:     end for
28:   else
29:     m := floor(n/2)
30:     return NumBetween($x_1, x_2, \ldots, x_m$) + NumBetween($x_{m+1}, x_2, \ldots, x_n$)
32:   end if
33: end function
```
Solution for Algorithm 2

First we specify the running time of NumBetween.

Base Case : O(1).

Recursive Case : S(n) = 2 * S(\frac{n}{2}) + c

With each layer of a recursion tree does a $2^h * c$ work and total height is $log n$. So we know that the running time of NumBetween is $O(n)$.

Then we go back to the function Foo2. Its For loop runs $O(n^2)$ times, each time doing $S(n)$ work. So

$$T(n) = O(n^2) \cdot S(n)$$

Since $S(n)$ is $O(n)$, the total running time is $O(n^3)$
4. **Michigan yet again** (12 points)

Finally, one of Jeff’s students proposed a faster algorithm, shown below as Foo3. What is the big-O running time of this method? Briefly justify your answer.

Hint: You may think of sort() as a black box which takes in an unsorted array and sorts it in O(n log n) time.

---

**Algorithm 3** Smart version

1. function Foo3(x₁, x₂, . . . , xₙ, k)
2.   if k > n then
3.      Error “Too few books”
4.   end if
5.   sort(Array) ▶ Put elements of Array into increasing order
6.   min = 1000000 ▶ An absurdly large value
7.   for i := 1 to (n - k) do
8.      temp := Array[i + k -1] - Array[i]
9.      if temp < min then
10.         min := temp
11.   end if
12.   end for
13.   return min
14. end function

---

**Solution for Algorithm 3** Running time is O(n log n)

Brief analysis:

Edge case checking : O(1).

Calling the sort() function : O(n log n).

For loop:

Since the array is in increasing order, we maintain a subarray of size k , each iteration we pop the number of the least value and add a number that is strictly bigger than the current maximum value calculate the new unfairness.

Each iteration takes O(1) time , total O(n-k) iterations. So total O(n-k) time.(worse case O(n)).

Return Operation : O(1) time.

So in our case, the running time will mostly depends on the sort() function, which is O(n log n) time.