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

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

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

2. **A Journey to Michigan** (8 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 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 should assume that the running time of `findSubsets` is proportional to the number of subsets it returns, and also that $k$ is a constant which is small compared to $n$.

   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.
Algorithm 1 First Attempt

1: function Foo(Array, k)
2:     if k > length(Array) then
3:         Error “Too few books”
4:     end if
5:     subsets = findSubsets(Array, k)
6:     minFairness = 1000000
7:     for each S in Subsets do
8:         min = maxValue
9:         max = 0
10:        for x in S do
11:            if x < min then
12:                min = x
13:            end if
14:            if x > max then
15:                max = x
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 (10 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.

In line 31, assume that it takes constant time to divide up the array and pass the half-size arrays to the recursive calls. (Whether this is actually true depends on the details of how we are storing arrays and passing them to the functions.)

Hint: you might want to use a second (helper) function in your definition of $T(n)$.
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 \quad \triangleright \text{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 = NumBetween(x_1, x_2, \ldots, x_n, low, 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 

21: function NumBetween(x_1, x_2, \ldots, x_n, l, k) 
22:   if n \leq 2 then 
23:     rv := 0 
24:     for i between 1 and n do 
25:       if l \leq x_i \leq h then 
26:         rv := rv + 1 
27:       end if 
28:     end for 
29:   else 
30:     m := floor(n/2) 
31:     return NumBetween(x_1, x_2, \ldots, x_m) + NumBetween(x_{m+1}, x_2, \ldots, x_n) 
32:   end if 
33: end function
4. Michigan yet again (8 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 of length \( p \) and sorts it in \( O(p \log p) \) time, leaving the sorted values in the same array.

**Algorithm 3** Smart version

```plaintext
1: function Foo3(\( x_1, x_2, \ldots, x_n, k \) )
2:     if \( k > n \) then
3:         Error “Too few books”
4:     end if
5:     sort(Array) \( \triangleright \) Put elements of Array into increasing order
6:     min = 1000000 \( \triangleright \) 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
```