**(3 points) Foundations of Algorithms Appendix A, Exercise Problem 5.**

**(5 points) Foundations of Algorithms Appendix A, Exercise Problem 6.**



**(2 points) Foundations of Algorithms Appendix A, Exercise Problem 11.**

Base Case:

Induction Hypothesis:

We need to show that,

There are n+1 occurrences of k(k!), therefore

**(2 points) Foundations of Algorithms Appendix A, Exercise Problem 12.**

Base Case:

Induction Hypothesis:

We need to show that:

**(2 points) Foundations of Algorithms Appendix A, Exercise Problem 13.**

Base Case:

Induction Hypothesis:

We need to show that:

**(2 points) Foundations of Algorithms Appendix A, Exercise Problem 14.**

Base Case:

Induction Hypothesis:

We need to show that:

This is equivalent to:

**(6 points) Foundations of Algorithms Chapter 1, Exercise 6.**

minmax(array a) {

min = 0;

max = 0;

if (a.length < 2) {

print("List not long enough for a min and max.");

}

if (a[0] > a[1]) {

min = a[1];

max = a[0];

} else {

min = a[0];

max = a[1];

}

for (i = 2; i < a.length-1; i+=2) {

if (a[i] > a[i+1]) {

min = Math.min(min, a[i+1]);

max = Math.max(max, a[i]);

} else {

min = Math.min(min, a[i]);

max = Math.max(max, a[i+1]);

}

}

if (a.length%2 == 1) {

min = Math.min(min, a[a.length-1]);

max = Math.max(max, a[a.length-1]);

}

print("Min: "+min+" | Max: "+max);

}

**(2 points) Foundations of Algorithms Chapter 1, Exercise 8.**

When the data to be searched is already sorted and is resorted not extremely often. Binary search is much more effective in this scenario.

**(2 points) Foundations of Algorithms Chapter 1, Exercise 9.**

Because exchange sort has O(n^2), applications such as sorting a database or other large data that are sorted often will be very inefficient when compared to other sorting algorithms.

**(2 points) Foundations of Algorithms Chapter 1, Exercise 10. You need to work on the exercise 6**

**algorithm only.**

Algorithm has a base where it checks if the length of the array is less than two. It has an if and else statement comparing the first two elements of the array to determine the initial min and max values. There is a for loop that starts at i = 2, up to the length of the array minus 1, and iterates by 2 each time. Inside the loop, there’s an if else statement that compares the element at i with the element at i+1. Inside each of those, there’s a Math.min() and Math.max() function performed based upon the comparison of the elements at i and i+1. After the loop, there is an if statement using mod to determine if the length of the array is odd. If it is odd, there is another Math.min() and Math.max() performed on the last element of the array. Lastly, the min and max values are printed.

The run time of this algorithm changes slightly if the array is of odd length or even length. If it is of even length, then there are 3n/2 – 2 array comparisons made. If it is of odd length, then there are 3n/2 array comparisons made. O(n)

(**1 point) Recursive Programming Chapter 1, Exercise 1.1**.

This function calculates n!, because it has a base case of 1 and successive multiplication by n, while calling the function with (n-1). For example, F(3) = 3\*2\*1 = 6; 3! = 6.

**(1 point) Recursive Programming Chapter 1, Exercise 1.2.**

term = 0;

for (i = 0; i < 4; i++) {

print(term);

term += 3;

}

term = 4;

for (i = 0; i < 4; i++) {

print(term);

term += 3;

}

**(2 points) Recursive Programming Chapter 1, Exercise 1.5.**

Is equivalent to:





