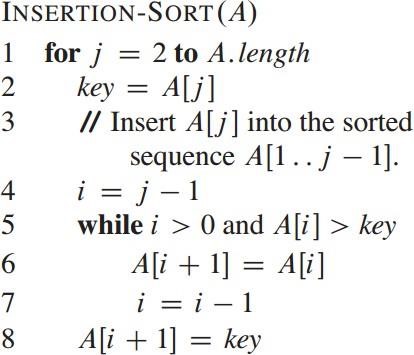
**COSC 3304 – Algorithms Design and Analysis**

**Assignment 1**

**Due: 23:59:00pm, Jan. 24, 2024 (Wednesday) - Andrew Kalathra**

1. Why is the worst case very useful for algorithm efficiency analysis? (30 points)
   1. The worst case is useful because we know that nothing can be worse than that. Using average is very close, but is impossible to collect due to its very nature (needing to test every possible input). Best case just tells you the fastest it can be run, but does not give actual data on if the algorithm could be improved. In this sense, the worst case gives the most representative analysis of efficiency of the algorithm.

1. When all elements in the input array ***A*** are the same, is it the best case or worst case for the INSERTION-SORT function below? Why? (35 points)



* 1. This would be the best case for the INSERTION-SORT function shown. This is because the while statement will always be false, so the for loop will continue till the end of the array, which would happen as the best case of the INSERTION-SORT function.
  2. For an example, we can take the arrays A = [4, 4, 4, 4], B = [4, 3, 2, 1], and C = [3, 1, 2, 4]. In this we see that function will have to first set j as the second element and check the element behind it. If the element behind it is greater than j, then an addition step is needed. In this way, A would not ever need this extra step, so it would be the best case. B would always need this extra step, so this would be the worst case. C would occasionally need this extra step, so it is closer to the average case (though that is impossible to obtain).

A math equations on a white background

Description automatically generated

1. Between the following two input arrays, A=[1, 2, 3, 4, 5, 6, 7, 8] and A=[8, 7, 6, 5, 4, 3, 2, 1], which requires more steps to run the MERGE-SORT function below? Why? (35 points)
   1. Modification: the ascending array is A1; the descending array is A2
   2. The first step is dividing the array into half recursively until 1 element is reached. This first step will cost the same amount of steps for both arrays since they contain the same amount of elements.
   3. The Merge portion of the code includes two steps: comparing, and if not in the right order, swapping. Both A1 and A2 will have to do the same amount of comparing steps. A1, however, has no need of swapping, whereas A2 needs to swap every element. This means that A2 requires more steps than A1.
   4. In conclusion, A1 requires less steps than A2 using the MERGE-SORT function.

1. What is the time complexity of the following program by assuming N=2k (\*\*Bonus\*\*) (please show detailed steps for full credits):

A close up of a math equation

Description automatically generated with medium confidence

* 1. The outer for loop will have a O(n)=N.
  2. For the inner loop, we can work through a few cases

|  |  |  |
| --- | --- | --- |
| i | Steps | N |
| 1 | 1 | 1 |
| 0 |  |  |
| 2 | 2 | 2, 3 |
| 1 | 3 |  |
| 0 |  |  |
| 4 | 4 | 4, 5, 6, 7 |
| 2 | 5 |  |
| 1 | 6 |  |
| 0 |  |  |
| 8 | 7 | 8, 9, 10, 11, 12, 13, 14, 15 |
| 4 | 8 |  |
| 2 | 9 |  |
| 1 | 10 |  |
| 0 |  |  |

* 1. As we can see, as N increases, the number of steps increases as well. Isolating the inner loop, we can see that (for example), a number such as N = 8 gives us 4 steps (steps 7 – 10), or N = 4 gives us 3 steps (steps 4 – 6).
  2. We can denote this inner loop as log2N + 1.
     1. Solving this out more thoroughly, we can take an example.
     2. Using logarithmic properties, we can rewrite log2N = x as 2x = N
     3. If we take N = 8, we can see from the chart that it would correspond to 4 steps. However, 2x = 8 would mean that x = 3 != 4, so the simple solution would just be to add one. This intuitively makes sense as when N = 1, it must run 1 time, so 2x = 1 would mean x = 0, then adding 1 would mean x = 1, so it would take 1 step (which is correct).
     4. So modifying our log, we can say that x (the number of steps) can be found (in the inner loop) through log2N + 1.
  3. This makes our inner loop be O(n) = logN (stripping away unnecessary terms).
  4. Combining our outer and inner loops, we get O(k) = NlogN
  5. This is not complete, however, as we do need to replace the N = 2k
  6. 2klog2k is quite repetitive and does not make immediate sense, but using what we had initially taken out (log2) along with logarithmic properties, we can rearrange it as shown:
     1. 2k \* k \* log22
        1. log22 = 1
     2. k2k
  7. So our final O(n) of the code shown is O(k) = 2k (because our k coefficient does not make that big of a difference in the face of the exponential portion)