

Faculty of Science

**Course**: CSCI 3070U: Design and Analysis of Algorithms

**Tutorial:** #2

**Topic:** Complexity

1. Write as an expression without summation, in terms of n.

(02 – 1), (12 – 1), (22 – 1), (32 – 1), …, (n2 – 1)

-1, 0, 3, 8, …, (n2 – 1)

Let x = (j2-1),

x(x-1) (j2-1)((j2-1) – 1) (j2-1) (j2-2)

-------- = ------------------- = -------------

2 2 2

1. Consider the following well-known algorithm for bubblesort. On p.40 of your textbook, this algorithm is proven correct. Using the iteration technique discussed in the lecture, determine the worst-case asymptotic complexity in the form Θ(f(n)).

BUBBLESORT(A)

* 1. for i = 1 to A.length – 1 n
  2. for j = A.length downto i+1 n (worst case)
  3. if A[j] < A[j-1] k
  4. temp = A[j] k
  5. A[j] = A[j-1] k
  6. A[j-1] = temp k

Since inside the inner loop is k \* n \* n, the asymptotic complexity is thus Θ(n2).

1. Consider the binary search algorithm given below.
   1. Identify one or more loop invariants that (collectively) are sufficient to describe the correctness of this algorithm’s ability to find a number within a sorted list of numbers.

At the start of each iteration, either:

1. X is not in A
2. X is in A[start..end]
   1. Describe the best case, average case, and worst case inputs for this algorithm.

Best: The mid point (as calculated on line 1) is the location of X in A

Worst: X is not in A

Average: X is at another position and will require between 1 and log n iterations to be found

* 1. Informally evaluate the worst-case asymptotic complexity in terms of Θ(f(n)). You don’t have to be formal about this, but I want you to justify your answer.

BINARYSEARCH(A, X, start, end)

* 1. mid = floor(end – start / 2)
  2. while (end – start) > 1
  3. if A[mid] = X
  4. return True
  5. else if A[mid] < X
  6. start = mid + 1
  7. else
  8. end = mid - 1
  9. return False

Binary search is Θ(log n). Intuitively, we should realize that since we are not subdividing the list based on any input (we divide the list before we look at any of its contents), the list is divided as closely in half as we can (excluding floor/ceiling). Thus, each iteration should result in a factor of ½. This progression naturally points to log2n.

1. Consider the following algorithm to reverse the order of a list. Write the recurrence equation for the running time (i.e. T(n)) for this algorithm. Draw a recursion tree for this recurrence and estimate the complexity of this function.

REVERSE(List)

1. **if** (List == []) **then**

2. **return** []

3. Rest = List[2..]

4. **return** append(REVERSE(Rest), List[1])

Where:

* List[2..] returns a list with the same elements as List, but with the first element removed
* append(x, y) takes a list (x) and an element (y), and appends y to the end of x

T(n) = T(n-1) + c1 + c2 + c3\*n + c4\*n

The actual drawing of the recursion tree is left as an exercise. At each level, i, the amount of (non-recursive) work done is Θ(n). There should be n levels in the worst case. The complexity of this algorithm is thus Θ(n2).