# CHAPTER 2 – Getting Started

## 2.1 – Insertion Sort

### 2.1-1

{31, 41, 59, 26, 41, 58}

{31, 41, 59, 26, 41, 58} : comparing 31 and 41, they are in the correct order.

{31, 41, 59, 26, 41, 58} : comparing 41 and 59, they are in the correct order.

{26, 31, 41, 59, 41, 58} : comparing 59 and 26, they are not in the right order, 26 moves to the front.

{26, 31, 41, 41, 59, 58} : comparing 59 and 41, they are not in the right order, 41 moves back one.

{26, 31, 41, 41, 58, 59} : comparing 59 and 58, they are not in the right order, 58 moves back one.

Sortation is complete

### 2.1-2

Initialization – when the loop starts, i < n, this is because I starts at 0 and n must be greater than or equal to 1. So the initialization always holds true prior to the first iteration of the loop given that the array is greater than 2 in length.

Maintenance – when the first iteration of the loop runs, at the end, i is incremented up. If i becomes equal to n it will stop, but this cannot happen until the step inside the loop has been completed. So, it will always complete the step.

Termination – Once i reached the value n, the loop will be terminated. n is our indicator that the loop would go out of bounds of the array given, so the algorithm finishes at the end bound of the array. The algorithm rely’s on evaluating all integers in the given bounds of the array, and knowing we looked at all of them, the algorithm is correct.

### 2.1-3

Reverse-Insertion-Sort(A, n)

1 for i = 2 to n

2 key = A[i]

3 j = i – 1

4 while j > 0 and A[j] < key

5 A[j+1] = A[j]

6 j = j -1

7 A[j+1] = key

### 2.1-4

Linear-Search(A, x, n)

1 while (i < n)

2 if x == A[i]

3 return i

4 i = i + 1

5 return NIL

Initialization – when the loop starts i is always less than n, so the while loop can always start

Maintenance – after we check if we can terminate the loop when the search condition is met, when it is failed we increment to the next value to check

Termination – upon reaching i = n, the while loop is left and we can assume that x does not occur in the array so we may return NIL

### 2.1-5

Add-Binary-Integers(A, B, n)

1 C[n]

2 carry = 0

3 for i = n-1 to 1

4 x = A[i] + B[i] + carry

5 carry = 0

6 C[i] = x % 2

7 if x = 2 or x = 3

8 carry = 1

## 2.2 – Analyzing Algorithms

### 2.2-1

is the leading factor, so;

### 2.2-2

Selection-Sort(A, n)

1 for i = 1 to n

2 key = A[i]

3 smallest = i

4 for j = i to n

5 if A[j] < A[i]

6 A[i] = A[j]

7 smallest = j

8 A[smallest] = key

The algorithm maintains a loop invariant of n-1 because we have n-1 elements to observe in the array.

Worst-case run time is . Best-case run time is , when the list is already sorted, it still needs to look through every other element and compare it to the ith element since it cannot assume it is done being sorted until the end.

### 2.2-3

Average case will be roughly . This is because on average sorts the the algorithm will run n/2 time. So, we simplify to n. The Worst-case scenario would have the same prediction, but instead, the reasoning is that we have to look through the whole array to either find or not find x, meaning we look through n units in the array.

### 2.2-4

Implement a check to see if the algorithm even needs to be used before using it.

## 2.3 – Designing Algorithms

### 2.3-1

{3, 41, 52, 26, 38, 57, 9, 49}

{3, 41, 52, 26} {38, 57, 9, 49}

{3, 41} {52, 26} {38, 57} {9, 49}

{3} {41} {52} {26} {38} {57} {9} {49}

{3, 41} {26, 52} {38, 57} {9, 49}

{3, 26, 41, 52} {9, 38, 49, 57}

{3, 9, 26, 38, 41, 49, 52, 57}

### 2.3-2

If p is entered as 1, then the only case in concern is when n = 1. This is because when n > 1, the question of p > r is always false. But if n = 1, then p >= r is true. So, if we say that p != r, we receive false back when n = 1 and we receive true back when n > 1. Which means it stops when the arrays are of length 1, but don’t stop when the arrays are longer than that length.

### 2.3-3

Initialization – The starting conditions to this while loop are that i is less than the length of the left array and j is less than the length of the right array. i will always be less than the length of the left array since it starts at 0 and the length is always greater than or equal to 1. The same goes for j and the right array.

Maintenance – The updates show that every time we look at I and j, one always gets incremented.

Termination – If we increment too far outside of the bounds of the while loop, or in other words, outside the bounds of one of the array parts, we end the algorithm.

Whatever is left over when one array is finished first, the right or the left, its counterpart will be completed by one of the other two while loops. So, in the end, every single number is entered into the newly sorted array. The solution is complete because there are no integers left to insert into the array.

### 2.3-4

Base step; n=2. When n = 2, T(2) = 2 and 2log2 = 2. Thus 2log2 = T(2)

Hypothesis step; Assuming T(n) = n log n is true if n = 2^k for some integer k > 0

Induction step; if n = 2^(k+1), then

T(2^(k+1))

= 2(T(2^(k+1)/2))+2^(k+1)

= 2(T(2^k)+2^(k+1)

= 2(2^klog2^k) + 2^(k+1)

= 2^(k+1)log2^k + 2^(k+1)

= 2^(k+1)(log(2^k)+1)

= 2^(k+1)((log(2^k)+log2)

= 2^(k+1)log(2^(k+1))

### 2.3.5

Recursive-Insertion-Sort(A, n)

If n <= 1

Recursive-Insertion-Sort(A, n-1)

Last = A[n-2]

J = n-2

While j >= 0 and A[j] > last

A[j+1] = A[j]

J = j – 1;

A[j+1] = last

Worst case should be theta(n^2) since we are running the while loop which as theta(n) complexity, n times.

### 2.3-6

Iterative-Binary-Search(A, v)

Right = A[1]

Left = A[A.length]

While (Right <= Left)

Middle = (right+left)/2

If v == A[middle]

Return middle

Else if v < A[middle]

Left = mid-1

Else

Right = mid+1

Return NIL

Recursive-Binary-Search(A, right, left, v)

If right > left

Return NIL

Middle = (right+left)/2

If v == A[middle]

Return middle

Else if v < A[middle]

Recursive-Binary-Search(A, right, middle-1, v)

Else

Recursive-Binary-Search(A, middle+1, left, v)

Intuitively we know that the worst case run time is nlogn because we run n times, except the subproblems always take logn time.

### 2.3-7

No, it would not improve the algorithm because the time would still take n times for each number in the array, and every time its inserted somewhere it takes n time to move elements over.

### 2.3-8

This algorithm would take the searched number, x, and find the difference between itself and a key in the array. Then it would decide if that difference is a key in the array or not. It would do this for all numbers in the array until it finds a solution. It would take nlogn time because we look through all numbers on the worst case, but we only need to compare it to numbers after it and not before because those comparisons were already done by previous iterations.

## CHAPTER 2 PROBLEMS

### 2-1

#### A

Assume that the lengths of each sub array are k, then insertion sort runs at k^2 time. That means that for the n/k sublists it can handle them in nk time because k^2\*n/k = nk.

#### B

Knowing that it takes nlogn time to sort n arrays of 1 array length sublists in normal merge sort, notice that there is no longer n sublists. Now there is n/k sublists so we can plug that in and see that it would take nlog(n/k) time to merge the sublists.

#### C

K can not grow faster than logn , because if it does the nk term will lead to the algorithm running worse that nlogn time. So lets assume that k = logn

nk + nlog(n/k) = nk + nlogn – nlogk

= nlogn + nlogn – nlog(logn)

= 2nlogn – nlog(logn)

= nlogn

So, k can only be as large as logn

#### D

Choose k to be logn.

### 2-2

#### A

We also need to prove that A’ has all of the elements in A.

#### B

Initiation – Always will be true as long as j and I are not at the same index. J can not be greater than n or less than i+1, and I can not be greater than n – 1 and less than i. The crossover between the ranges does not over lap as I + 1 is never bigger than I – 1 + 1 which is i. So j can never equal i.

Maintenance – if the elements share the necessary relation, they are swapped to be in the correct order.

Termination – this terminates when j reached i + 1 which is equal to n, so it stops when it compares the same indices together.

#### C

Initiation – will always start as long as the array holds more than one element to sort. Ie, n >= 2.

Maintenance – we go through the loop invariant stated in part b.

Termination – we end when we reach n-1 which is the end of the comparisons because j = n and that is the final thing to look at.

#### D

It’s n^2 which is the same worst case run time as insertion sort, but bubble sort is worse because insertion sort only swaps n times where as bubble sort can swap up to n^2 times. Insertion sort would run faster than bubble sort on the same input.

### 2-3

#### A

It should just be theta(n)

#### B

Naïve-Polynomial-Evaluation(A, n, x)

Total = 0

For I = 1 to n

Total = total + A[i]\*x^(i-1)

Return total

My algorithm runs in n^2 time because the power algorithm would take I time which is only limited by n.

Its worse than Horner’s rule.

#### C

I’m not doing this, it’s like proving that a summation is a summation. What’s the point, we defined it to be this way.

### 2-4

#### A

(1, 5) , (2, 5) , (3, 5), (4, 5) and (3, 4)

#### B

The array where the elements are ordered n, n-1, … , 2, 1.

It has n(n-1)/2 inversions.

#### C

It would be the same because insertion sort has to basically invert every time it inserts a number somewhere into the array.

#### D

The algorithm would split down the array like merge sort and it would count how many swaps it does, in other words it would just track the work merge sort does without changing the workings of the algorithm.