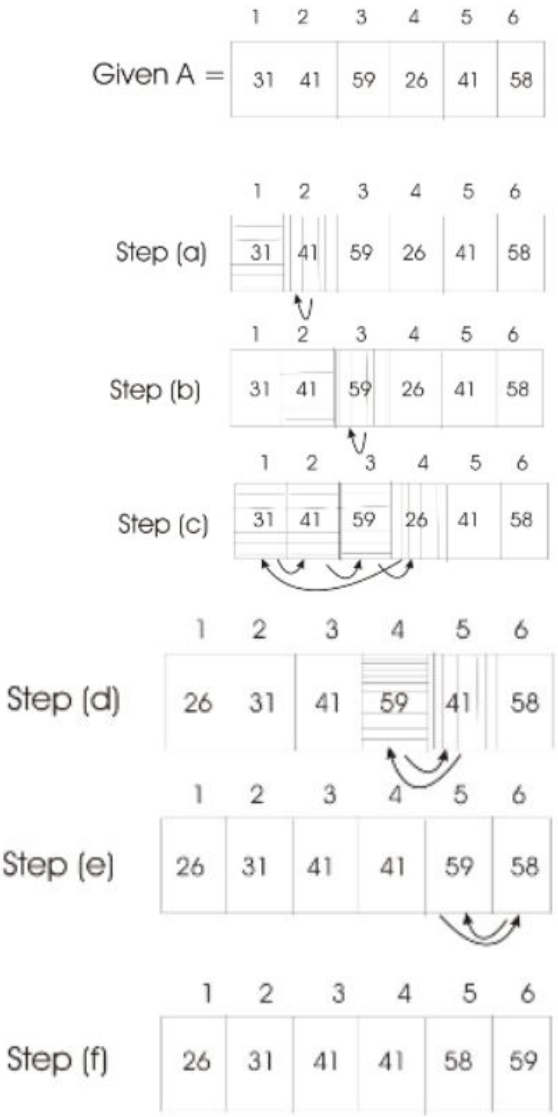
***2.1-1***

Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

**Step 1** of 1




***2.1-2***

Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of non-decreasing order.

Step 1 of 1

**INSERTION – SORT (A)**

```
1. for j = 2 to A. length
2.     key = A[j]
3. // Insert A[J] into the sorted sequence A[1.. j-1].
4.      i = j – 1
5.     while i > 0 and A[i] < key
6.        A[i+1] = A[i]
7.         i = i – 1
8.        A[i+1] = key
```

On Analysis the worst-case time complexity is $\Theta(n^2)$.

### *2.1-3*

Consider the ***searching problem***:

**Input:** A sequence of $n$ numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ and a value $v$.

**Output:** An index $i$ such that $v = A[i]$ or the special value NIL if $v$ does not appear in $A$.

Write pseudocode for ***linear search***, which scans through the sequence, looking for $v$. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

Step 1 of 2

**Pseudocode for Linear Search**

// the pseudocode for linear search, it scans the sequence for *v* using a loop invariant:

**Input:** A sequence of n numbers $A = \langle a_1, a_2, \ldots\ldots a_n \rangle$ and a value *v*.

**Output:** An index *i* such that *v* = *A[i]* or the special *NIL* if *v* does not appear in *A*.

**Linear Search *(A, v)***

```
1. for i =1 to n
2. {
3 if A[i] = v
4. return i
5. }
6. return NIL
```

### *2.1-4*

Consider the problem of adding two $n$-bit binary integers, stored in two $n$-element arrays $A$ and $B$. The sum of the two integers should be stored in binary form in

an $(n + 1)$-element array $C$. State the problem formally and write pseudocode for adding the two integers.

**Step 1** of 2

**Consider the data:**

1. Addition of two *n*-bit binary integers.

2. Binary integers should be stored in two n-element arrays *A* and *B*.

3. The sum of the two binary integers should be stored in $(n+1)-element\ array\ C$

**Explanation of the problem:**

This information can be expressed in the form table as shown below:

| | | | |
|---|---|---|---|
| $A[1]$ | ... | $A[n]$ | |
| $B[1]$ | ... | $B[n]$ | |
| $C[1]$ | ... | $C[n]$ | $C[n+1]$ |

Assume that initially the key value is 0, and then we get

$$C[1]=(A[1]+B[1]+key\ value)\ \mathrm{mod}\ 2$$
$$key\ value=(A[1]+B[1]+key\ value)\ div\ 2$$

These equations can be expressed for $n$ -terms are as follows:

$$C[i]=(A[i]+B[i]+key\ value)\ \mathrm{mod}\ 2$$
$$key\ value=(A[i]+B[i]+key\ value)\ div\ 2$$
$$C[n+1]=key\ value$$

**Step 2** of 2

**Pseudo code for addition of two n-bit binary integers is:**

**Algorithm** Binary integer-Addition $(A[n], B[n])$

**//Input:** Two $n$ -bit binary integers, sorted as a sequence of n numbers in array $A=[1...n]$ and a sequence of $n$ -numbers in array $B=[1...n]$.

**//Output:** The sum of the two integers, stored in binary form as a sequence of $(n+1)$ numbers in array $C=[1...n+1]$ .

1. key value = 0

2. **for** $i \leftarrow 1\ to\ n\ do$ //this loop will run *n*-times

3. $sum \leftarrow A[i]+B[i]+key$

4. $C[i] \leftarrow sum\ \mathrm{mod}\ 2$

5. $key \leftarrow sum\ \mathrm{div}\ 2$

6. $C[n+1]=key$

7. Return C

**Analyzing the running-time:**

The worst case running time of this algorithm is $\Theta(n)$

### *2.2-1*

Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of $\Theta$-notation.

**Step 1** of 1

The general formula for polynomial function can expressed in terms $\Theta$ as

$p(n)=\sum_{i=0}^{d} a_i n^i$ where $a_i$ are the constants and $a_d > 0$,

we have $P(n) = \Theta(n^d)$.

$f(n) = n^3/1000 - 100n^2 - 100n + 3$

$P(n) = \Theta(n^3)$.

### *2.2-2*

Consider sorting $n$ numbers stored in array $A$ by first finding the smallest element of $A$ and exchanging it with the element in $A[1]$. Then find the second smallest element of $A$, and exchange it with $A[2]$. Continue in this manner for the first $n - 1$ elements of $A$. Write pseudocode for this algorithm, which is known as ***selection sort***. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n - 1$ elements, rather than for all $n$ elements? Give the best-case and worst-case running times of selection sort in $\Theta$-notation.

**Step 1** of 2

**Selection-Sort(A)**

```
1. for i = 1 to n − 1
2.    index = i
3.  for j = i + 1 to n
4.     if A[j] < A[index]
5.      index = j
6.    if index ≠ i
7.      A[i] ↔ A[index] // exchange
```

**Step 2** of 2

The algorithm maintains the loop invariant that at the start of each iteration of the outer **for** loop, the subarray A[1....j-1] consists of the j-1 smallest elements in the array A[1.... n], and this subarray is in sorted order. After the first n-1 elements, the subarray A[1...n -1] contains the smallest n- 1 elements, sorted, and therefore element A[n] must be the largest element.

Observe that on the $i^{th}$ iteration of loop, the inner loop executes exactly $(n-i)$ times, and on each such execution exactly one array comparison is performed . Thus in all cases (best, worst, average) the number of comparisons done by SelectionSort is

***2.2-3***

Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in $\Theta$-notation? Justify your answers.

**Step 1** of 1

On average the number of elements $\frac{1}{n}\frac{n(n+1)}{2}=\frac{n+1}{2}$ being searched.

In worst cast it n and element may not be found also.

The average running time is $\frac{n+1}{2}=\Theta(n)$ since for all $n \geq 1$ we have

$$\frac{1}{2}n \leq \frac{n+1}{2} \leq n$$

The worst case running time is $\Theta(n)$.

***2.2-4***

How can we modify almost any algorithm to have a good best-case running time?

**Step 1** of 1

To improve the best case, all we have to do it to be able to solve one instance of each size efficiently. We could modify our algorithm to first test whether the input is the special instance we know how to solve, and then output the canned answer.

For sorting, we can check if the values are already ordered, and if so output them. For the traveling salesman, we can check if the points lie on a line, and if so output the points in that order.

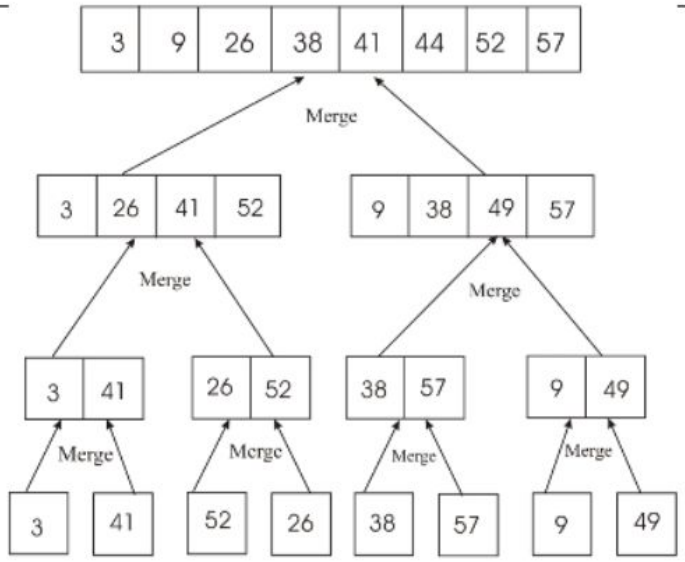
***2.3-1***

Using Figure 2.4 as a model, illustrate the operation of merge sort on the array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

**Step 1** of 1

Operation of Merge – sort on the array

A = < 3, 41, 52, 26, 38, 57, 9, 49 >




Initial sequence

***2.3-2***

Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array $L$ or $R$ has had all its elements copied back to $A$ and then copying the remainder of the other array back into $A$.

**Step 1** of 1

```
MERGE (A, p, q, r)

    //Calculate the size of the left array (nL) and the right array (nR)
1 nL = q-p+1
2 nR = r-q
   // Create the left array, L which has indexes from 1 to nL
3 L = Array(1...nL)
    // Create the right array, R which has indexes from 1 to nR
4 R = Array(1...nR)
    // Initialize the left and right arrays
5 for i = 0 to nL-1
6     L[i] = A[p+i]
7 for i = 0 to nR-1
8     R[i] = A[q+i]
   // Remember how many items we've used in each list
```

```
9  iL = 1
10 iR = 1
   // Merge the lists
11 for k = p to q
12   if iL > nL
13     A[k] = R[iR++]
14   else if iR > nR
15     A[k] = L[iL++]
16   else if L[iL] <= R[iR]
17     A[k] = L[iL++]
18   else
19     A[k] = R[iR++]
```

### *2.3-3*

Use mathematical induction to show that when $n$ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{for } k > 1 \end{cases}$$

is $T(n) = n \lg n$.

**Step 1** of 2

Consider the recurrence relation:

$$T(n) = \begin{cases} 2 & \text{if } n=2 \\ 2T\left(\frac{n}{2}\right)+n & \text{if } n = 2^k, \text{for } k > 1 \end{cases} = n \log n$$

**Base case:**

$T(n) = n \log n$, when n=2

$$T(n) = n \log n$$
$$T(2) = 2.\log 2$$
$$T(2) = 2.1 \qquad (\text{since } \log_2 2 = 1)$$
$$T(2) = 2$$

**Step 2** of 2

**Mathematical induction Hypothesis:**

$$T(n/2) = (n/2)\log(n/2)$$

Substituting n value then we get

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(n/2)\lg(n/2) + n \\ &= 2(n/2)(\lg n - \lg 2) + n \\ &= n \lg n - n + n \\ &= n \lg n \end{aligned}$$

Hence, $T(n) = n \log n$

### *2.3-4*

We can express insertion sort as a recursive procedure as follows. In order to sort $A[1..n]$, we recursively sort $A[1..n-1]$ and then insert $A[n]$ into the sorted array $A[1..n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

**Step 1** of 2

```
RECURSIVE-INSERTION-SORT (A,n)
1 if n>1
2   RECURSIVE-INSERTION-SORT (A,n-1)
3   INSERT(A,n)

INSERT(A,k)
// Insert A[k] to the sorted A[1..k-1]
// Use Insertion-Sort without outside loop
1 key=A[k]
2 i=k-1
3 while i>0 and A[i]>key
4   A[i+1] = A[i]
5   i--
6 A[i+1] = key
```

**Step 2** of 2

Since it takes $\Theta(n)$ time in the worst case to insert A[n] into the sorted array of A[1..n-1], let $T(n)$ be the total time it takes to insertion sort a list with $n$ elements, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n-1) + \Theta(n) & \text{otherwise.} \end{cases}$$

$$T(n) = T(n-1) + \Theta(n)$$

Therefore the recurrence is $T(n) = \Theta(n^2)$

### *2.3-5*

Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence $A$ is sorted, we can check the midpoint of the sequence against $v$ and eliminate half of the sequence from further consideration. The ***binary search*** algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

Step 1 of 2

**Iterative version of binary search:**

Binary Search $(A, v)$

1. $Low = 1$, $high = Alength$, found=false
2. while $p \le r$ and found
3. $\quad mid = \left\lfloor \frac{low + high}{2} \right\rfloor$
4. $\quad$ if $v = A[mid]$
5. $\quad\quad$ found=true
6. $\quad$ else if $v < A[mid]$
7. $\quad\quad$ $high = (mid - 1)$
8. $\quad$ else
9. $\quad\quad$ $low = (mid + 1)$
10. if found
11. $\quad$ return $mid$
12. else
13. $\quad$ return NIL

Step 2 of 2

Let $t_k$ denote the length of the sub array being searched on the $k^{th}$ iteration of loop 2-9. Since the length is halved on each iteration/invocation, we have $t_1 = n$ and $t_k = t_{k-1}/2$, and therefore $t_k = n/2^{k-1}$. In worst case the target $v$ is not in the list. The process stops when $t_k = 1$, i.e. $n/2^{k-1} = 1$. Therefore $k = \lg(n) + 1$,which gives the number of iterations (or the recursion depth) in worst case. The running time of Binary Search is therefore $\Theta(\lg n)$ in worst case.

### *2.3-6*

Observe that the **while** loop of lines 5–7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1 \,..\, j - 1]$. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

**Step 1** of 3

The INSERTION-SORT procedure uses a linear search to scan through the sorted sub array $A[1\ldots..j\text{-}1]$.

INSERTION-SORT ($A$)

1 **for** $j = 2$ **to** $A.\ length$

2 $key = A[j]$

3 //Insert $A[j]$ into the sorted sequence $A[1\ldots j-1]$.

4 $i = j-1$

**5 while $i > 0$ and $A[i] > key$**

**6 $A[i+1] = A[i]$**

**7 $i = i-1$**

8 $A[i+1] = key$

To locate the insertion position ($A[i] > key$) among the sorted sub-array $A[1\ldots j-1]$, the procedure INSERTION–SORT uses linear search technique.

Because the array is already sorted using loop invariant, we can select the middle of $A[1, j\text{-}1]$, check to see if value is greater or lower value, and recursively repeat this procedure until left with one element.

**Step 2** of 3

The pseudo code for such an algorithm is:

Binary-Insertion-Sort ($A$)

1 **for** $i = 0$ to $A.\ length$ **do**

2 $temp = A[i]$

3 $left = 0$

4 $right =$ i

5 **while** $left < right$ do

6 $middle = (left + right)/2$

7 **if** $temp \geq A[middle]$

8 $left = middle + 1$

9 **else** $right = middle$

10 **for** $j = i$ **to** $left$

11 **do**

12 $stemp = A[j-1]$

13 $A[j-1] = A[j]$

14 $A[j] = stemp$

15 $j = j - 1$

**Step 3** of 3

The number of comparisons required for binary search is calculated as follows:

$$1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots+\frac{1}{n-1}=\sum_{i=1}^{n-1}\frac{1}{n}$$
$$=\log_2 n$$

Therefore, binary search required $\lfloor \log_2 n \rfloor$ comparisons where $1 \le i < n$

The first for loop iterates $n$ times and the algorithm takes $n \times \lfloor \log_2 n \rfloor$ times.

Therefore, the running time of the algorithm is $\Theta(n \log n)$ .

The time required to find the element to be replaced is $O(\log n)$. In order to insert the element in the required position, remaining elements need to be shifted. The time required to shift the elements is $O(n)$ .

**The total time required to find and insert the element is $O(n \log n)$ .**

**Therefore, it is not possible to improve the overall worst-case running of insertion sort to $\Theta(n \log n)$ time even we use a binary search instead of linear search.**

## *2.3-7* ★

Describe a $\Theta(n \lg n)$-time algorithm that, given a set $S$ of $n$ integers and another integer $x$, determines whether or not there exist two elements in $S$ whose sum is exactly $x$.

**Step 1** of 2

**Algorithm to determine whether two Elements Exist in a Set**

**Algorithm**:

ELEMENT-x(*S*, *x*)

Input: A sorted array *S* and a value *x*

Output: Boolean value indicating *true*, if there exist two elements in *S*, otherwise *false*.

1 MERGE-SORT*(S,1,length(S))* // Sorting the elements in a set using mergesort.

2 v =REMOVE-MAX-ELEMENT(*S*) // Remove the last element (*v*) from the set.

3 **While**(*S*) // Searching for two elements *v, w*.

4 **do** w = BINARY-SEARCH(S,x-y,1,length(S))

5 **if** w+v=x // checking for two elements sum is equal to *x*.

6 **then return** true // if two elements sum is equal to x then return true.

i. // otherwise remove the next last element (*v*) from the set.

7 **else** v= REMOVE-MAX(S)

8 **return** NIL // If set is empty, so two elements sum is equal to x, then return *NIL*.

**Step 2** of 2

**Algorithm running time:**

The setp1 performs merge sort operation. Merge sort algorithm running at $\Theta(n \lg n)$ time. So, the step1 takes $\Theta(n \lg n)$ time. Step 2 performs remove an element. So it takes $O(1)$ time. Step 4 performs binary search operation. The binary search algorithm running at $\lg n$ time. So, the step 4 have need of at most $\lg n$ time. And finally steps 3 to 7 are repeated at most $n$ times. Therefore the total running time of this algorithm is $\boxed{\Theta(n \lg n)}$.

### *2-1 Insertion sort on small arrays in merge sort*

Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to ***coarsen*** the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which $n/k$ sublists of length $k$ are sorted using insertion sort and then merged using the standard merging mechanism, where $k$ is a value to be determined.

***a.*** Show that insertion sort can sort the $n/k$ sublists, each of length $k$, in $\Theta(nk)$ worst-case time.

***b.*** Show how to merge the sublists in $\Theta(n \lg(n/k))$ worst-case time.

***c.*** Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of $k$ as a function of $n$ for which the modified algorithm has the same running time as standard merge sort, in terms of $\Theta$-notation?

***d.*** How should we choose $k$ in practice?

**Step 1** of 4

a) Insertion sort takes $\Theta(k^2)$ time per k-element list in worst case time. Therefore sorting n/k such k-element lists, takes $\Theta(k^2 n/k) = \Theta(nk)$ worst case time.

**Step 2** of 4

b) The key idea is to merge the lists pair wise, and then merge the resulting lists pair wise until there is only one list. The pair wise merging requires $\Theta(n)$ work at each level, since we are still working on n elements, even if they are partitioned among sub lists. The number of levels, starting with n/k, k-element list and finishing with 1 n – element list, is $\lceil \lg(n/k) \rceil$, therefore the total running time for the merging is $\Theta(n \lg (n/k))$.

**Step 3** of 4

c) The largest asymptotic value of $k_1$ for which the modified algorithm has the same Asymptotic running time as standard merge sort, is $k = \Theta(\lg n)$. The combined running time is $\Theta(n \lg n + n \lg n - n \lg \lg n)$, which is $\Theta(n \lg n)$ for larger values of k the term nk would be larger than $\Theta(n \lg n)$.

**Step 4** of 4

d) In practice, k should be chosen such that it minimizes the running time of the combined algorithm. If you assign the number of data to x-axis and the consumed time to y-axis, you can draw $C_1 \cdot x \lg x$ and $C_2 \cdot x^2$ graphs. There should be an intersection point determined by $C_1$ and $C_2$, which represents the data number that the merge sort is firstly faster than insertion sort. So, it is natural to choose the x-value of the intersection point as k. It is well known that the insertion sort is the fastest one when $n \leq 16$, so we could say k is 16 experimentally.

### *2-2 Correctness of bubblesort*

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

BUBBLESORT($A$)

```
1  for i = 1 to A.length − 1
2      for j = A.length downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

***a.*** Let $A'$ denote the output of BUBBLESORT($A$). To prove that BUBBLESORT is correct, we need to prove that it terminates and that

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n] , \tag{2.3}$$

where $n = A.length$. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

The next two parts will prove inequality (2.3).

***b.*** State precisely a loop invariant for the **for** loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

***c.*** Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the **for** loop in lines 1–4 that will allow you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.

***d.*** What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

**Step 1** of 5

| | BUBBLE SORT (A ) | Execution |
|---|---|---|
| 1. | for i = 1 to A. length-1 | n + 1 |
| 2. | for j =A.length down to i+1 | n(n+1)/2 |
| 3. | if A[i] < A[j − 1] | n(n+1) /2 |
| 4. | then exchange A[j] with A[j-1] | 0 to n(n+1) /4 |

**Step 2** of 5

a) We need to show that the elements of A' form a permutation of the elements of A, i.e., they only come from A. Here an algorithm that takes an array A as input and terminates and produces an array A', but doesn't sort.

**Step 3** of 5

b) The loop invariant may be stated as follows:

At the start of each iteration of the for loop of lines 2–4,
$A[j] = \min\{A[k] | j \leq k \leq n\}$
and the sub array A[j…n] is a permutation of the values that were in A[j..n] at the time that the loop started.

Now the proof:

• Initially, j = n, and the sub array A[j..n] consists of single element A[n]. Thus, the loop invariant trivially holds.

• For the maintenance part, consider an iteration for a given value of j = j0, by the loop invariant, A[j0] is the smallest value in A[j0..n]. Lines 3–4 exchange A[j0] and A[j0 − 1] if A[j0] is less than A[j0 − 1], and so A[j0 −1] will be the smallest value in A[j0 − 1..n] afterward. Since the only change to the sub array A[j0−1..n] is this possible exchange, and the sub array A[j0..n] is a permutation of the values that were in A[j0..n] at the time that the loop started, we see that A[j0 − 1..n] is a permutation of the values that were in A[j0 − 1..n] at the time that the loop started. Decrementing j to j0 − 1 for the next iteration maintains the invariant.

• Finally, the loop terminates when j reaches i. By the statement of the loop invariant, $A[i] = \min\{A[k] | i \leq k \leq n\}$ and A[i..n] is a permutation of the values that were in A[i..n] at the time that the loop started.

Step 4 of 5

(C) We can have the following loop invariant for the for loop in lines 1–4:
At the start of each iteration of the this for loop, the sub-array A[1..i − 1] consists of the i − 1 smallest values originally in A[1..n], in sorted order, and A[i..n] consists of the n − i + 1 remaining values originally in A[1..n].

Now the proof:

• Before the first iteration of the loop, i = 1. The sub-array A[1..i − 1] is empty, and so the loop invariant trivially holds.

• Consider an iteration for a given value of i = i0. By the loop invariant, A[1..i0 − 1] consists of the i0 smallest values in A[1..n], in sorted order.
The above Part b showed that after executing the for loop of lines 2–4, A[i0] is the smallest value in A[i0..n], and so A[1..i0] is now the i0 smallest values originally in A[1..n], in sorted order. Moreover, since the for loop of lines 2–4 permutes A[i0..n], the sub-array A[i0+1..n] consists of the n−i0 remaining values originally in A[1..n].
Increment of i to i0 + 1 makes the loop invariant holds at the beginning of the next loop.

• Finally, for loop of lines 1–4 terminates when i = n+1, so that i−1 = n. By the statement of the loop invariant, A[1..i−1] is the entire array A[1..n], and it consists of the original array A[1..n], in sorted order.

Step 5 of 5

D) Worst-case cost of bubble sort.

$$= (n+1) + 3n(n+1)/2 = \tfrac{1}{2}(3n2 + 5n + 2)$$
$$= \Omega(n2)$$

The same worst case for insertion sort also. So both worst-cases are same.

### *2-3 Correctness of Horner's rule*

The following code fragment implements Horner's rule for evaluating a polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n)\cdots)) , \end{aligned}$$

given the coefficients $a_0, a_1, \ldots, a_n$ and a value for $x$:

```
1  y = 0
2  for i = n downto 0
3      y = a_i + x · y
```

***a.*** In terms of Θ-notation, what is the running time of this code fragment for Horner's rule?

***b.*** Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule?

***c.*** Consider the following loop invariant:

At the start of each iteration of the **for** loop of lines 2–3,

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k .$$

Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to show that, at termination, $y = \sum_{k=0}^{n} a_k x^k$.

***d.*** Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$.

**Step 1** of 9

$$P(x) = \sum_{k=0}^{n} a_k x^k$$

$$= a_0 + x(a_1 + x(a_2 + - - - - - - - + x(a_{n-1} + xa_n) - - - -)),$$

| | | |
|---|---|---|
| 1. | y = 0 | 1 |
| 2. | for i = n downto 0 | 1 |
| 3. | y = $a_i$ + x.y | n |
| | | n + 2 |

= n + 2

**Step 2** of 9

a) Simply the line 1 executes in the order $\Theta$ (1) time
Line 2 executes in he order '1' time.
Line 4 executes in the $\Theta$ (n) time
Thus, the total running time is (n+1) .

**Step 3** of 9

b) PSEUDOCODE for NAÏVE polynomial – Evaluation polynomial

$$P(x) = \sum_{k=0}^{n} a_k x^k$$

$$= a_0 + a_1 x + a_2 x^2 + - - - - - - - + a_{n-1} x^{n-1} + a_n x^n$$

| | | |
|---|---|---|
| 1. | y = 0 | 1 |
| 2. | for i = n downto 0 | 1 |
| 3. | y = $a_i$ + x.y | n |
| | | n + 2 |

= n + 2

**Step 4** of 9

The time complexity is same when compared Horner's rule that is O(n).
In Horner's rule we are computing the coefficients in a different manner mean that coefficients of x are manipulated in Horner's rule.

**Step 5** of 9

c) Loop invariant proof.

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

Initialization: Prior to the first iteration when y – 15 initialized to '0' we started with I = n. We have

$$y = \sum_{k=0}^{n-(n+1)} a_{k+n+1} x^k = \left[ a_{k+n+1} x^k \right]_0^1$$

$$= a_{n+1} x^0$$

**Step 6** of 9

Where $a_{n+1}$ is not given such prior to the iteration we can say that the loop invariant holds.
Maintenance: When we go through the loop, at each iteration from the higher term to $a_n x^n$ is calculated and is assigned to y and in the next loop iteration the value y is used as a coefficient to x according to the Horner's rule so this loop invariant holds.

**Step 7** of 9

Termination:-
When the loop terminates with a condition i < 0 when the last iteration finishes the 'i' value becomes less than '0', at that iteration the y value becomes.

$$y = \sum_{k=0}^{n} a_k x^k$$

Horner's rule also performs same polynomial addition but in a different approach. Thus after termination also the loop invariant holds. The value of y is nothing but the sum of all terms in a polynomial caution.

**Step 8** of 9

d) $$p(x) = \sum_{k=0}^{n} a_k x^k$$

$$= a_0 + x(a_1 + x(a_2 + - - - - - - - + x(a_{n-1} + xa_n) - - -))$$

To conclude that this evaluates a polynomial let take an example.

$$a_0 = 1,\ \ a_1 = 2,\ \ a_2 = 3,\ \ x = 4$$

According to Horner's rule the polynomial evaluation will be as follows.

Step 1: $a_n x = 4 \cdot 3$

$= 12$

Step 2: $a_{n-1} - x \cdot a_n = 2 + 12$

$= 14$

Step 3: $x(a_{n-1} + x \cdot a_n) = 4 \cdot 14$

$= 56$

Step 4: $a_0 + x(a_{n-1} + x \cdot a_n) = 56 + 1$

$= 57$

**Step 9** of 9

Usual Polynomial Method of Evaluation

$$y = a_0x^0 + a_1x^1 + a_2x^2$$

$$= 1 \cdot 1 + 2 \cdot 4 + 3 \cdot 4^2$$

$$= 1 + 8 + 48$$

$$= 57$$

Thus we can conclude that Horner's rule correctly evaluates a polynomial equation.

### *2-4 Inversions*

Let $A[1..n]$ be an array of $n$ distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair $(i, j)$ is called an ***inversion*** of $A$.

***a.*** List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

***b.*** What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

***c.*** What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

***d.*** Give an algorithm that determines the number of inversions in any permutation on $n$ elements in $\Theta(n \lg n)$ worst-case time. (*Hint:* Modify merge sort.)

**Step 1** of 5

a) The inversions are (2,1),(3,1), (81), (61), (86

Provide feedback (0)

**Step 2** of 5

b) The array with elements from {1, 2,…,n} with the most inversions is $\langle n, n-1, n-2, \ldots, 2, 1 \rangle$. For all $1 \le i < j \le n$, there is an inversion (i , j). The number of such inversions is $\binom{n}{2} = n(n-1)/2$.

**Step 3** of 5

c) Suppose that the array A starts out with an inversion (k, j). Then k>j and A[k]>A[j]. At the time that the outer for loop of lines 1-8 sets key = A[j], the value that started in A[k] is still somewhere to the left of A[j]. That is, it's in A[i], where $1 \le i \le j$, and so the inversion has become (i, j). Some iteration of the while loop of lines 5-7 moves A[i] one position to the right. Line 8 will eventually drop key to the left of this element, thus eliminating the inversion. Because line 5 moves only elements that are less than key, it moves only elements that correspond to inversions. In other words, each iteration of the while loop of lines5-7 corresponds to the elimination of one inversion.

**Step 4** of 5

d) We follow the hint and modify merge sort to count the number of inversions in $\Theta(n \lg n)$ time. To start, let us define a merge-inversion as a situation within the execution of merge sort in which the MERGE procedure, after copying A[p..q] to L and A[q+1..r] to R, has values x in L and y in R such that x>y. Consider an inversion(i,j), and let x=A[i] and y=A[j], so that i<j and x>y. We claim that if we were to run merge sort, there would be exactly one merge inversion involving x and y. To see why, observe that the only way in which array elements change their positions is within the MERGE procedure. Moreover, since MERGE keeps elements within L in the same relative order to each other, and corresponding for R, the only way in which two elements can change their ordering relative to each other is for the greater one to appear in L and the lesser one to appear in R. Thus, there is at least one merge-inversion involving x and y. To see that there is exactly one such merge-inversion, observe that after any call of MERGE that involves both x and y, they are in the same sorted sub array and will therefore both appear in L or both appear in R in any given call thereafter. Thus, we have proven the claim. We have shown that every inversion implies one merge-inversion. In fact, the correspondence between inversions and merge-inversions is one to one. Suppose we have a merge-inversion involving values x and y, where x originally was A[i] and y was originally A[j]. Since we have a merge-inversion, x>y. and since x is in L and y is in R, x must be within a sub array preceding the sub array containing y. Therefore x started out in a position I preceding y's original position j, and so (i, j) is an inversion. Having shown a one-to one correspondence

between inversions and merge-inversions, it suffices for us to count merge-inversions. Consider a merge-inversion involving y in R . Let z be the smallest value in L that is greater than y. At some point during the merge process z and y will be the exposed values in L and R, i.e., we will have z=L[i] and y=R[j] in line 13 of MERGE. At the time, there will be merge-inversions involving y and L[i],L[i+1],L[i+2],…,L[n1], and these $n_1 - i + 1$.Merge-inversions will be the only ones involving y. Therefore, we need to detect the first time that z and y become exposed during the MERGE procedure and add the value of $n_1 - i + 1$ at the time to our total count of merge-inversions.

```
COUNT-INVERSIONS(A, p, r)
   inversions = 0
      if p<r
      q=⌊(p+r)/2⌋
inversions = inversions + COUNT-INVERSIONS(A, p, r)
inversions = inversions + COUNT-INVERSIONS(A,q+1,r)
inversions = inversions + MERGE-INVERSIONS(A, p, q, r)
return inversions
```

```
MERGE-INVERSIONS(A, p, q, r)
   n1 = q-p+1
   n2 = r-q
let L[1.. n1+1] and R[1.. n2+1] be new arrays
   for i=1 to n1
    L[i] = A[p+i-1]
    for j=1 to n2
    R[j] = A[q+j]
    L[n1+1] = ∞
    L[n2+1] = ∞

    i=1
    j=1
inversions =0
counted = FALSE
  for k= p to r
   if counted == FALSE and R[j] < L[i]
   inversions = inversions + n1 - i + 1
  counted = TRUE
  if L[i] ≤ R[j]
  A[K] = L[i]
    i = i+1
  else A[k] = R[j]
    j= j+1
 counted = FALSE
  return inversions
```

**Step 5** of 5

The initial call is COUNT-Inversions(A,1,n).

In MERGE-INVERSIONS, the boolean variable counted indicates whether we have counted the merge-inversions involving R[j]. We count them the first time that both R[j] is exposed and a value greater than R[j] becomes exposed in L array. We set counted to FALSE upon each time that a new value becomes exposed in R. We don't have to worry about-inversions involving the sentinel $\infty$ in R, since no value in L will be greater than $\infty$.

Since we have added only a constant amount of additional work to each procedure call and to each iteration of the last for loop of merging procedure, the total running time of the above pseudocode is the same as for merge sort $\Theta(n \lg n)$.