CHAPTER 02

Exercises

Ex 2.1

Q1. Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array A = {31,41,59,26,41,58}.

Ans.

Iteration(j)	Iteration(i)	Item1	Item2	Item3	Item4	Item5	Item6	Key	Comments
		31	41	59	26	41	58	N/A	Initial condition
2	1	31	41	59	26	41	58	41	Loop passed as 41>31
3	2	31	41	59	26	41	58	59	Loop passed as 59>41
4	3	31	41	59	26	41	58	26	Loop Enters
4	3	31	41	59	59	41	58	26	A[i+1] = A[i]
4	2	31	41	41	59	41	58	26	A[i+1] = A[i]
4	1	31	31	41	59	41	58	26	A[i+1] = A[i]
4	0	31	31	41	59	41	58	26	Loop Terminates
4	0	26	31	41	59	41	58	26	A[i+1] = key
5	4	26	31	41	59	41	58	41	Loop Enters
5	4	26	31	41	59	59	58	41	A[i+1] = A[i]
5	3	26	31	41	59	59	58	41	Loop Terminates
5	3	26	31	41	41	59	58	41	A[i+1] = key
6	5	26	31	41	41	59	58	58	Loop Enters
6	5	26	31	41	41	59	59	58	A[i+1] = A[i]
6	4	26	31	41	41	59	59	58	Loop Terminates
6	4	26	31	41	41	58	59	58	A[i+1] = key
		26	31	41	41	58	59	N/A	Final Sorted array.

Other way to understand above process is given under. Analyze whichever you want.

31/i	41/j	59	26	41	58

↓(No inversion as A[j] > A[i])

31	41/i	59/j	26	41	58

↓(No inversion as A[j] > A[i])

31	41	59/i	26/j	41	58

↓(3 forward insertion)

26	31	41	59/i	41/j	58

↓(1 inversion)

26	31	41	41	59/i	58/j

↓(1 inversion)

26	31	41	41	58	59

Q2. Rewrite the INSERTION-SORT procedure to sort into non-increasing instead of non- decreasing order.

Ans. Pseudo Code:-

```
INSERTION-SORT(A)
  for j = 2 to A.length
    key = A[j]
    i = j - 1
    while i > 0 and A[i] < key
      A[i + 1] = A[i]
      i = i - 1
    A[i + 1] = key
```

Q3. Consider the searching problem:

Input: A sequence of n numbers **A = <a_1 ,a_2,a_3,a_4 ……a_n>** 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.

Ans.

Pseudocode:

```
LINEAR-SEARCH(A, v)
  for i = 1 to A.length
    if A[i] == v
      return i
  return NIL
```

Loop invariant :- $\forall$ j <i, A[j] ≠ v.

Speaking informally – All the items that proceeds the item positioned at i , is not equal to v.

Initialization:- Before the start of loop , the subarray is empty. Hence the invariant holds.

Maintenance:- During each loop iteration we compare **v** with **A[i]** . If they are same we return it which is true result. Hence it's the ith elements that matches but the array A[1..i-1] still has item which doesn't equal with v. If we assume that algorithm has completed it means line 3 has not been executed. Then A[i] ≠ v is true and our invariant still holds.

A more formal approach will be by contradiction. Let us assume $\forall$ k <i , A[k] = v. Since we are in ith iteration and k is less than i. Line 3 must not have executed because it leads to termination. Since 3rd line is not executed means Line 2 must have been a false statement i.e. A[k] ≠ v. But this contradicts our first assumption. Hence it proves our invariant.

Termination:- Our algorithm can terminate in two ways.

a. The loop terminates when i = A.length+1 . At that time i would be n+1. Substituting it in A[1…i-1], we find that A[1…n] contains elements which are different from v. Thus we return **nil** as per requirement. Hence our algorithm is correct.

b. If line 3 executes successfully. In that case A[i] must have been equal to v. Hence we return i, which is correct answer. Hence our algorithm is correct.

Q4. 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.

Ans. Let us first state the problem formally.

Input : Two n-bit arrays, A= {$a_1,a_2,\dots,a_n$} and B = {$b_1,b_2,\dots,b_n$} and 1 (n+1) bit array C = {$c_1,c_2,\dots,c_n$}

where a_i , b_i , c_i are bits i.e. a_i , b_i , $c_i \in \{0,1\}$

Output : Modified C array, $c_i \in \{0,1\}$ and C holds the binary addition of A and B.

Pseudocodes:-

1.

Input:- Two arrays A and B containing binary digits of two number **a** and **b**

Output :- an (n+1) element array C containing a+b in binary form.

```
ADD-BINARY(A, B)
  C = new integer[A.length + 1]
  carry = 0
  for i = 1 to A.length
    C[i] = (A[i] + B[i] + carry) % 2  // remainder
    carry = (A[i] + B[i] + carry) / 2 // quotient
  C[i + 1] = carry
  return C
```

2.

Input:- Two arrays A and B containing binary digits of two number **a** and **b**

Output :- an (n+1) element array C containing a+b in binary form.

```
ADD-BINARY(A, B)
  $C = \text{new integer}[A.length + 1]$
  $carry = 0$
  for $i = 1$ to $A.length$
    $carry = (A[i]\ \mathbf{XOR}\ B[i]\ \mathbf{XOR}\ carry)$
    $C[i] = (A[i]\ \mathbf{AND}\ B[i])\ \mathbf{XOR}\ (A[i]\ \mathbf{AND}\ carry)\ \mathbf{XOR}\ (B[i]\ \mathbf{AND}\ carry)$
  $C[i] = carry$
  return C
```

Note : AND, XOR are basic bitwise operator.

Ex 2.2

Q1. Express the function **$n^3/1000-100n^2-100n+3$** in terms of Θ-notation.

Ans. From our understanding we know that while calculating Θ-notations. We mostly worry about the terms can dominates the expression whenever input size grow to infinity. We care this way because other smaller terms becomes insignificant when compared with the larger term.

$$f(n) = \Theta(g(n)) \ \Rightarrow \exists\ c_1, c_2, n_0 : \forall n \geq n_0 \quad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

To take it further

Let $f(n) = 2^n + 2n$.

Now consider what is going to happen when n → 100.

2^{100} will be **1,267,650,600,228,229,401,496,703,205,376**.

While the other being **200**.

I think now it's clear why we care about larger terms.

Coming to our original problem.

If we **ignore** the **low order terms** and **constant factors**. It will reduce to **$\Theta(n^3)$**.

Q2. 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 Θ-notation.

Ans. **Pseudocode:-**

```
n = A.length
for i = 1 to n - 1
   smallest_pos = i
   for j = i + 1 to n
      if A[j] < A[smallest_pos]
         smallest_pos = j
   swap(A[i], A[smallest_pos])
```

Loop invariant :- **∀ k < i , ∀ j ≥i , A[1] ≤ A[2] ≤ …. ≤ A[j] ≤ A[j]**

Our invariant says that all the item which proceeds the i^{th} are in order(sorted) and they are smaller than any element whose index is not less than i .

Initialization:- Before the start of iteration of loop , the subarray is hence, so the variant holds.

Maintenance:- During each loop iteration we compare mark smallest element's position as i's positon. Then we iterate over remaining array A[i+1…n]. If we find any element smaller than marked smallest we modify the marker. In the end of inner loop we swap. If marked element was already smallest we swap same element otherwise we insert A[j] in it proper position by swap in A[1…i-1].

Termination:- The loop terminates when i > A.length-1. At that time A[1…n-1] will have smallest i-1 elements but in sorted order. Hence remaining n_{th} will be largest among all. Hence we terminate the loop with assurance that A[1…n] is now sorted.

As mentioned above after loop termination we have n-1 smallest element in the A[1..n-1]. It itself implies that $\mathbf{n_{th}}$ is **largest** among all. Hence we run only it n-1 times.

Analysis :

We must note that in any case we must swap n-1 times (7 lines) .

Hence total no comparisons is :-

$$(n\text{-}1) + (n\text{-}2) + \ldots + 1 = \sum_{1}^{n\text{-}1}$$

By using AP

$\sum_1^{n-1} = (n-1)+1/2 * (n-1) = ½n(n-1) = ½(n^2-n)$ [(n-1) because last element is already sorted.]

Ignoring constant we get complexity is

$\Theta(n^2)$

Q3. 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 Θ-notation? Justify your answers.

Ans.

No. of checks :

Average case

Let us assume that all the elements are equally likely. Hence on average we need to check n/2.

This is because any element has the chance to be the target.

Thus for 1 element, no of steps will be 1

For 2nd element it would be 2.

Since we have n case result would be 1 + 2+ …+ n divided by n.

$\Rightarrow \frac{n(n+1)}{2n} \Rightarrow \frac{(n+1)}{2}$

Worst case

In Worst case, element will not be in array. Hence it's easy to count to the no. of check which is **n**.

Running time

Since we can see that time depend on no. of checks. But since we are looking for Θ-notation, which discards all low order terms and constant.

Discarding 2 in from average case check we get,

Running time in both average and worst case is Θ(n).

Q4. How can we modify almost any algorithm to have a good best-case running time?

Ans. There are mainly two cases

- When we don't care about correctness of the algorithm. In that case we can return some valid solution for some cases.

E.g. In case of sorting we simply return the inputted array and do nothing. Since we did nothing it's Θ(1) solution. We will rely on the fact that the input is already sorted.

- When we care about correctness. In that case we can pre-compute some valid solutions for some inputs. And whenever we receive any input firstly we can check that, inputted input matches with the pre-computed solution. If that's the case we can simply return our **hard-coded** , pre-computed solution.
  E.g.
  1. In case of exponent calculation. We can pre-compute 2^{100} and for every input if input matches with 2^{100} we can return the cached solution. Otherwise we can run the normal algorithm for solution.
  2. Another analogy can be sorting. For every array we can first check if it's already sorted. If it is simply return it. Checking it would take Θ(n). Otherwise run the normal sort procedure.

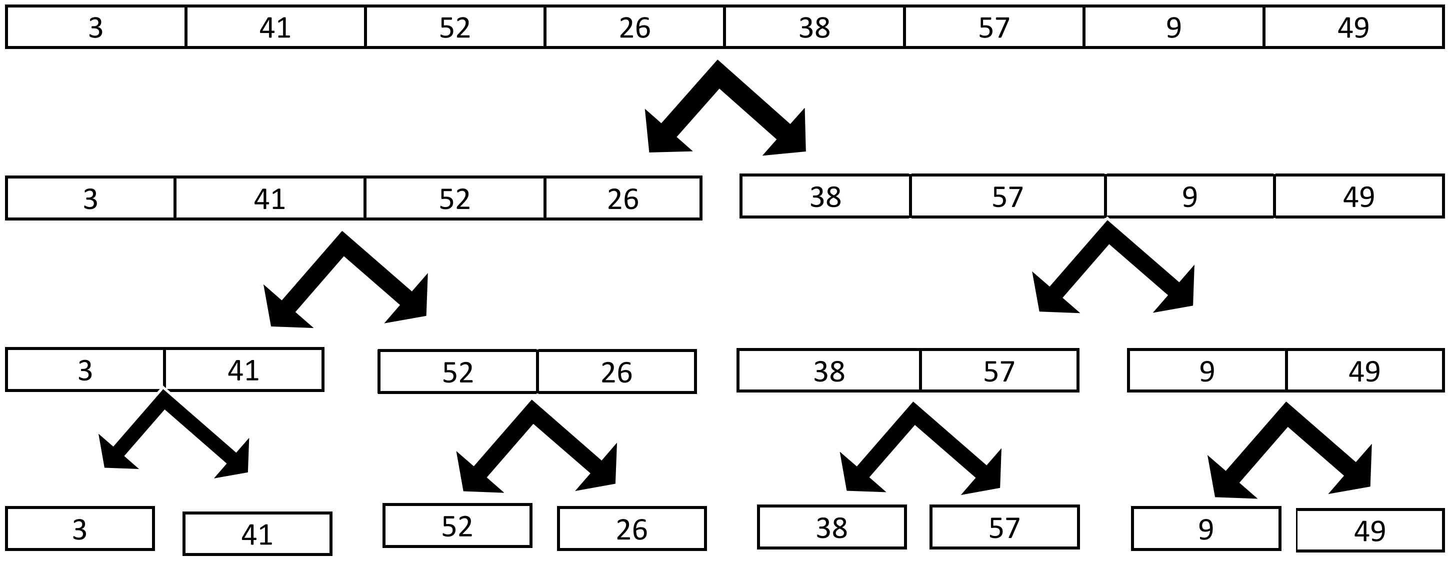
Ex 2.3

Q1. Using Figure 2.4 as a model, illustrate the operation of merge sort on the array A{3,41,52,26,38,57,9,49}.

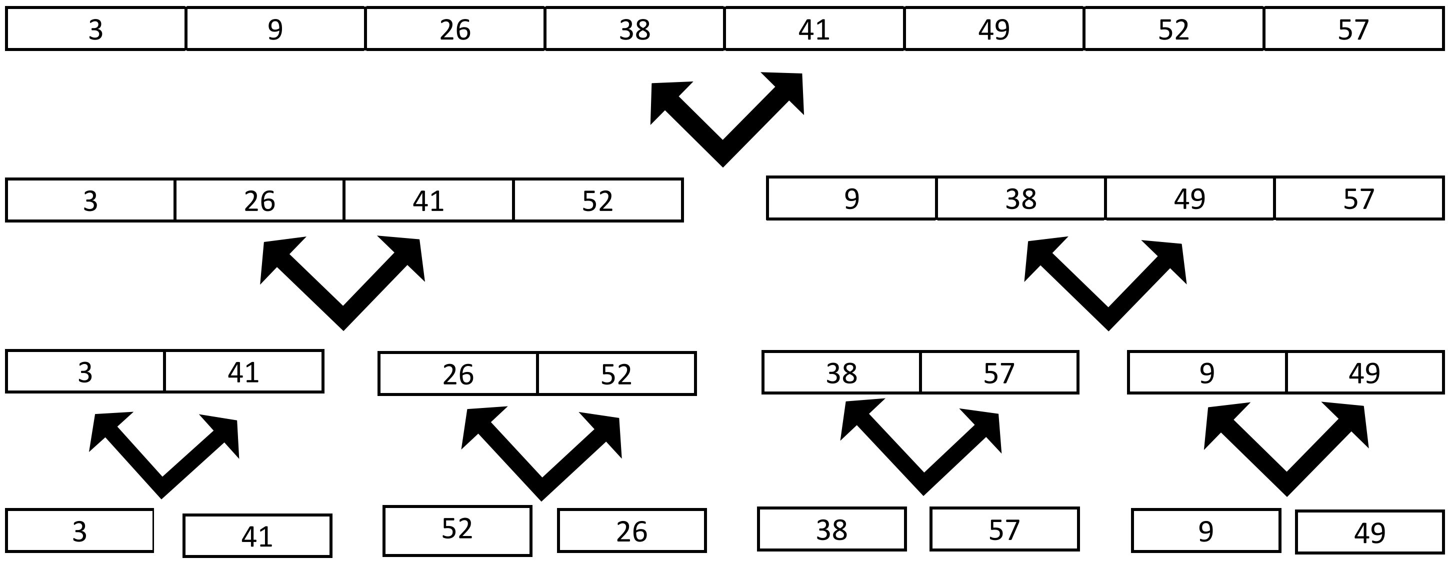
Ans.

DIVIDE STEP

Ans.



MERGE STEP



Q2. 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.

Ans.

Pseudocode:-

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

```
        j = j + 1
    else if j > n2
        A[k] = L[i]
        i = i + 1
    else if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
```

Q3. 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.

Ans.

- Base Case

  $$\text{For } n = 2^1, T(n) = 2 \lg 2 = 2 \quad [\because \lg_2 2 = 1]$$

- Let us assume it hold for some constant k

  $$\Rightarrow T(n) = n \lg n = 2^k \lg 2^k = k2^k \text{ [Eq-1]}$$

- Let us try to prove for k+1

  $$T(2^{k+1}) = 2\,T(2^{k+1}/2) + 2^{k+1}$$
  $$= 2\,T(2^k) + 2^{k+1}$$
  From eqn 1
  $$= 2\,(k\,2^k) + 2^{k+1}$$
  $$= k\,2^{k+1} + 2^{k+1}$$
  $$= 2^{k+1}(k+1)$$
  $$= 2^{k+1} \lg 2^{k+1} \; [a = \lg_2 2^a]$$
  $$= n \lg n$$

Q4. 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.

Ans.

Let us derive a recurrence for insertion sort.

Divide : Divide the n-element array into two sub-arrays with one part containing 1 element and other with n-1

Conquer : Sort each sub-array recursively; the single element is already sorted.

Combine : Combine the two sub-arrays by inserting the single element into it's appropriate position that keeps the array sorted.

Let T(n) : Worst running time to sort n-elements.

D(n) : Time required to divide the sub-problem. Calculating index

C(n) : Time required to combine the solutions. Insertion element in its proper position.

- Recurrence :

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n-1) + T(1) + D(n) + C(n) & n > 1 \end{cases}$$

Since division step only do index calculation it's Θ(1) and so do sorting a single element. Hence recurrence reduces to

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

Since in combine step we need to find appropriate position of the element. Hence we have to traverse the array and in worst case we may need to traverse n-1 items. Hence C(n) = Θ(n).

$$T(n) = \begin{cases} \Theta(1) & \text{If } n = 1 \\ T(n-1) + \Theta(n) & n > 1 \end{cases}$$

Q5. 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 Θ(lg n).

Ans.

- Iterative:

```
ITERATIVE-BINARY-SEARCH(A, v, low, high)
   while low ≤ high
      mid = floor((low + high) / 2)
      if v == A[mid]
         return mid
      else if v > A[mid]
         low = mid + 1
      else high = mid - 1
   return NIL
```

- Recursive:

```
RECURSIVE-BINARY-SEARCH(A, v, low, high)
   if low > high
      return NIL
   mid = floor((low + high) / 2)
   if v == A[mid]
      return mid
   else if v > A[mid]
      return RECURSIVE-BINARY-SEARCH(A, v, mid + 1, high)
   else return RECURSIVE-BINARY-SEARCH(A, v, low, mid - 1)
```

Recurrence :

Let us derive a recurrence for insertion sort.

Divide : Divide the n-element array into two sub-arrays with one part containing 1 element and other with n-1

Conquer : Sort each sub-array recursively; the single element is already sorted.

Combine : Combine the two sub-arrays by inserting the single element into it's appropriate position that keeps the array sorted.

Let T(n) : Worst running time to sort n-elements.

D(n) : Time required to divide the sub-problem. Index calculation (calculation of mid)

C(n) : Time required to combine the solutions. Return the solution.

D(n) we only need to calculate the mid which takes constant time. Hence D(n) is $\Theta(1)$.

C(n) : In combine step or better say. In resulting step all we need to do is return the index if we find the value which is independent of the size of array. Hence it's $\Theta(1)$.

T(n) : In recursive step we work with half part (Either A[low….mid] or A[mid+1….high]). Hence though it divides our problem in two part from which we consider only **one** hence a = 1 .

By recurrence relation

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 2, \\ aT(n/b) + D(n) + C(n) & \text{if } n = 2^k \text{, for } k > 1 \end{cases}$$
{where **a** is no. of sub-problems each of size **1/b**}.

Above statement recurrence reduces to

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ T(\frac{[n-1]}{2}) + \Theta(1) & n > 1 \end{cases}$$

$$T(n) \leq \begin{cases} \Theta(1) & +n = 1 \\ T(\frac{n}{2}) + \Theta(1) & n > 1 \end{cases}$$

Since there are many methods to solve this but we haven't studies them yet. Let's us try to solve this through from our previous knowledge.

T(n) $\leq$

$$T(\tfrac{n}{2}) + \Theta(1), \qquad 1$$

$$T(\tfrac{n}{4}) + \Theta(1) + \Theta(1), \qquad 2$$

$$T(\tfrac{n}{8}) + \Theta(1) + \Theta(1) + \Theta(1), \qquad 3$$

………………………………………………………..

$$T(2) + \Theta(1) + \Theta(1) + \ldots\ldots\ldots + \Theta(1) \qquad k\text{--}1$$

$$T(1) + \Theta(1) + \Theta(1) + \ldots\ldots\ldots + \Theta(1) \qquad k$$

$$\Theta(1) + \Theta(1) + \Theta(1) + \ldots\ldots\ldots + \Theta(1) \qquad k+1$$

From above steps we can derive relation

$$T(n) \leq \Theta(1) + \Theta(1) + \Theta(1) + \ldots\ldots\ldots + \Theta(1)$$

Now all we need to do is to calculate the no. of Θ(1) terms. Let us see how we can do that.

Notice that there is only 1 such term after step 1 and the denominator of problem size become 2 = 2^1

In step 2 there are two such terms and the denominator of problem size becomes 4= $2^{2.}$

Same referring to 3 we can see the denominator size is 8 = 2^3 (3 being the no. of Θ terms.)

Hence in kth step .

Denominator will be 2^k , using this statement we get

$$T(n) \leq T(\tfrac{n}{2^k}) + \Theta(1) + \ldots + \Theta(1)$$

In order to have size 1 as we get from k +1 step.

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$\lg n = \lg 2^k$$

$$\lg n = k \lg 2$$

$$k = \lg n$$

Substituting back we get

$$T(n) \leq T(\tfrac{n}{2^k}) \underbrace{+ \Theta(1) + \ldots}_{\lg n \text{ terms}} + \Theta(1)$$

$$T(n) \leq (\lg n + 1)\, \Theta(1)$$

$$\mathbf{T(n) = \Theta(\lg n)}$$

Q6. 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 Θ(n lg n)?

Ans. We can use binary search **but** it **won't** help us in any way. In insertion sort we need to copy element to its neighbor. Binary search can tell us how many shift we need to do but it won't get rid us of copying we need to do.

Q7. Describe a (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.

Ans.

Solution 1-

Explanation :
We first sort the array. After sorting we take two pointer and point them to the first and the last element. Since array is sorted we are sure that 1st will point to smallest element and 2nd to largest element.
Now we compute the sum of value pointed by pointer.

- If sum happens to be **desired result** we simply return it.
- **If Sum is smaller**. Since we are already summing the smallest and largest element. To produce a bigger result. We either need to point to 2nd to a bigger element or 1st to bigger element. Since increasing 2nd will lead to out of bound. We increase 1st pointer to point just bigger element. If that matches we return.
- **If sum is larger**. We either need to decrease the 1st to point a smaller element or 2nd to point a smaller element. Since 1st already point to smallest , decreasing it will lead to out of bound. Hence we reduce the 2nd pointer to point just smaller element. If that matches we return otherwise will do the same process again and again. We do this process while 2nd pointer pointing position is greater than 1st 's location. Once 1st crosses second we simply return false (or null as per implementation) to indicate that sum pair doesn't exist.

```
Use Merge Sort to sort the array A in time Θ(n lg(n))
i = 1
j = n
while i < j do
    if A[j] + A[j] = S
        return true
    if A[i] + A[j] < S
        i = i + 1
    if A[i] + A[j] > S then
        j = j − 1
return false
```

Solution 2

```
Use Merge Sort to sort the Array A in time Θ(n lg n)
for A[i] in A[1,n]
    Complement = sum – x;
    if binarySearch(A,complement,i,n)
```

```
                return true
```

First, sort S, which takes Θ(n lgn). Then, for each element s_i in S , i = 1.….,n, search A[i+1….n] for $s_i' = x – s_i$ by **binary search**, which takes **Θ(lgn).**

- If s_i' is found, return its position;
- otherwise, continue for next iteration.

$$T(n) = \underbrace{\Theta(n \lg n)}_{\text{Merge Sort Time}} + \underbrace{\Theta(\lg n) + \ldots + \Theta(\lg n)}_{\text{Time to perform binary search}}$$

$$T(n) = \Theta(n \lg n) + \underbrace{n\ \Theta(\lg n)}_{\text{In Worst case there will be n searches.}}$$

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

Problems

Q1. **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 sub problems become sufficiently small. Consider a modification to merge sort in which n/k sub lists 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 sub lists, each of length k, in $\Theta(nk)$ worst-case time.

b. Show how to merge the sub lists in $\Theta(n \lg (\frac{n}{k}))$ worst-case time.

c. Given that the modified algorithm runs in $\Theta(nk + n \lg(\frac{n}{k}))$ 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 Θ-notation?

d. How should we choose k in practice?

Ans.

Let us first state the new modification.

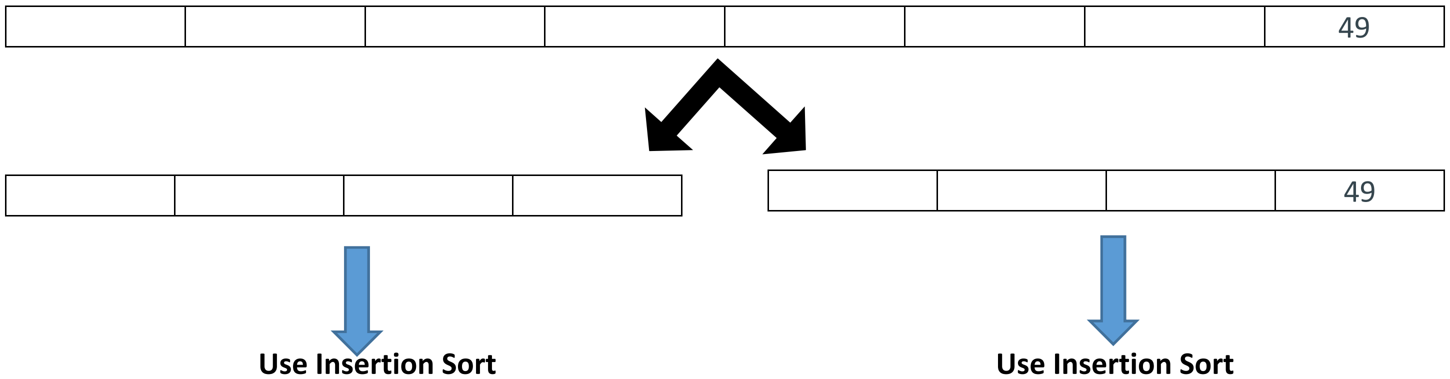
To do merge sort, we check the problem size and if it is below some **LEAF-SIZE** we can perform insertion sort instead of further dividing it to perform of merge sort. Because insertion sort works better for smaller sizes.

When size will reach k we should have n/k problems.

If you still didn't get why n/k? Go back to check merge sort. Let us go to merge sort. In given array. Let us assume that if size reduces to 4(k). We will stop dividing and use insertion sort. Now when we divide 8 we get 2 sub problems. Now we reached size 4.

We can see that no of sub-problems are 8/4 = 2 (n/k)

Ans.



Use Insertion Sort **Use Insertion Sort**

a. Insertion Sort sorts a sequence of k items in T(k) = $\Theta(k^2)$. Now we have n/k sequences, the overall running time will be

$$\frac{n}{k}\,\mathrm{T(k)} = \frac{n}{k}\,\Theta(\mathrm{k}^2) = \frac{n}{k}\,k^2 = \Theta(\mathrm{nk}).$$

Note : Don’t think that **k** constant and we should discard it. K is not a constant.

b. Remember that merge process takes Θ(n) to merge two sub-sequences producing n sorted elements.

Let us assume that at rth level we have k elements. So time we need to merge elements will be

$$\mathrm{T(k)} = \frac{\mathrm{n}}{\mathrm{k}}\,\Theta(\mathrm{k}) = \Theta(\mathrm{k})\ [\text{ we have n/k lists}]$$

At r-1th level

$$\mathrm{T}(\tfrac{k}{2}) = \frac{2\mathrm{n}}{\mathrm{k}}\,\Theta(\mathrm{k}) = \Theta(\mathrm{n})$$

Since we used in our analysis that we have lg($\frac{\mathrm{n}}{\mathrm{k}}$) levels. In original merge sort we have **lg n** levels. But in modified algorithm we stop at k. If we won’t have discarded at k it would make a tree with lg n leaves. Hence these k could have made an addition of m(no. of levels)

$$\frac{k}{2^m} = 1$$

$$\Rightarrow \mathrm{m} = \lg \mathrm{k}$$

Hence new height of the tree is **lg n – lg k = lg($\frac{n}{k}$)**

Since we saw above that each level takes Θ(n) time.

$\lg(\frac{n}{k})$ will take

$$\lg(\tfrac{n}{k}) * \Theta(\mathrm{n}) = \Theta(\mathrm{n}\lg(\tfrac{n}{k})$$

c. Let us first see how we get Θ(nk + n lg ($\frac{n}{k}$))

$$\underbrace{\Theta(\mathrm{nk})}_{\text{Insertion sort time}} + \underbrace{\Theta(\mathrm{n}\lg(\tfrac{n}{k}))}_{\text{Recursive merging}} = \underbrace{\Theta(\mathrm{nk} + \mathrm{n}\lg(\tfrac{n}{k}))}$$

As per problem we want to know largest k such that

$$\Theta(nk + n \lg (\frac{n}{k})) \quad = \Theta(n \lg n\ )$$

$$\Theta(nk + n \lg n - n \lg k\ ) = \Theta(n \lg n)$$

In order to make the equation true.

n lg n on left side must dominate nk.

Hence $\Theta(nk) = \Theta(n \lg n\ )$

Hence **k ∈ Θ(lg n)**

d. To choose best k , we must run the two algorithm on required machine. And should note the output. Then we compare the noted data and try to choose the best k.

Q2. **Correctness of bubblesort**

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

```
for i=1 to A.length-1
    for j= A.length down to i+1
        if A[j] < A[j-1]
        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]≤ A[2]≤ A[3]…… A[n] where n = A.length. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

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 in-equality (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?

Ans.

a. We also need to prove that elements in the array are same and then array has same no of elements. This can be easily proved as only modification algorithm does is **swap** which surely don't change the array's element. So the resulting array is just a permutation of original array but with the above constrain.

b**. Loop invariant** :- At the start of each iteration, the position of the smallest element of A[i….n] is at most j.

- Initialization:- This is clearly true prior to the first iteration because the position of any element is at most A.length.
- Maintenance:- Suppose that the position of smallest element is at most k and j = k . Then we compare A[k] to A[k − 1]. If A[k] < A[k − 1] then A[k − 1] is not the smallest element of A[i..n], so when we swap A[k] and A[k − 1] we know that the smallest element of A[i..n] must occur in the first k − 1 positions of the subarray, the maintaining the invariant. On the other hand, if A[k] ≥ A[k − 1] then the smallest element can't be A[k]. Since we do nothing, we conclude that the smallest element has position at most k − 1. Upon termination, the smallest element of A[i..n] is in position i.
- Termination :- At termination the j will be having an exchange in i and i+1 hence smallest element of A[i..n] is in position i which is within the bound of j.

c. **Loop invariant**:- At the start of each iteration A[1..i-1] contains the i-1 elements but in sorted order.

- Initialization:- At start i= 1. And hence the first 1-0 element are trivially sorted.
- Maintenance:- Let us assume that A[1..i-1] contains sorted elements. Now in above part we shows that at the end of line 4, **i** will be having smallest element. Since i-1 elements are already sorted , A[i] must be ith smallest element. Therefore A[1..i] contains sorted elements.
- Termination:- Upon termination i = A.length -1 . This means A[1….n-1] elements are already sorted and hence the nth remaining element must be nth smallest. Hence A[1…n] contains sorted elements upon termination.

d. Assume the worst case time is denoted by T(n) and S(n) denotes the worst running due to exchange and comparing in line 3 and 4 which takes Θ(1).

$$\mathrm{T(n)} = \underbrace{\sum_{i=1}^{n-1}}_{\text{Outer loop}} \left(\underbrace{\sum_{i+1}^{n}}_{\text{Inner loop}} (S(n)) \right)$$

$$\mathrm{T(n)} = \sum_{i=1}^{n-1} \left(\sum_{i+1}^{n} \Theta(1)\right)$$

$$\mathrm{T(n)} = \Theta(1) \sum_{i=1}^{n-1} \left(\sum_{i+1}^{n} (1)\right)$$

$$\mathrm{T(n)} = \Theta(1) \sum_{i=1}^{n-1} (n-1)$$

$$\mathrm{T(n)} = \Theta(1)\left(\sum_{i=1}^{n-1} (n) - \sum_{i=1}^{n-1} (i)\right)$$

$$\mathrm{T(n)} = \Theta 1 \left(n(n\text{-}1) - \frac{n(n-1)}{2} \right)$$

$$\mathrm{T(n)} = \Theta 1 \left(\frac{n(n-1)}{2}\right)$$

$$\mathrm{T(n)} = \Theta(n^2)$$

Therefore, as we can see that bubble sort have the same asymptotic time of Θ(n^2). But if we consider best case, insertion sort has best case when array is sorted has it has best running time of Θ(n) but in case of bubble sort there is not any termination till the loop is completed. Hence it has best case of Θ(n^2).

We could add a condition to enhance it to detect best case where there are no swaps. But as a general requirement we don't rely on best case. Hence it's not worth much effort.

Q3. **Correctness of Horner's rule**

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

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

Given the coefficient $a_0, a_1 \ldots a_n$ and a value for x:

```
$y = 0$
for $i = n$ down to 0
$y = a_i + x*y$
```

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 naïve 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 for loop of line 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 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 c_n.

Ans.

a. If we assume that the trivial arithmetic calls run in constant time. Then the turn of iterations are n hence it's Θ(n).

b.

```
y = 0
for i=0 to n do
    k = 1
    for j=1 to i do
        k = k*x
    y = y + ai*k
return y
```

Analysis :

$$T(n) = \underbrace{\sum_{i=0}^{n} (\Theta(1)}_{\text{Outer loop}} + \underbrace{\sum_{j=1}^{i} \Theta(1))}_{\text{Inner loop}}$$

$$T(n) = \sum_{i=0}^{n} (\sum_{j=1}^{i} \Theta(1))$$

$$T(n) = \Theta(1) \sum_{i=0}^{n} (\sum_{j=1}^{i} (1))$$

$$T(n) = \Theta(1) \sum_{i=0}^{n} (i)$$

$$T(n) = \Theta(1)(\frac{n(n+1)}{2})$$

$$T(n) = \Theta(n^2)$$

This code has runtime Θ(n2). This is much slower than Horner's rule.

However we write another native yet efficient approach.

```
y = 0
k = 1
for i=0 to n do
    y = y + $a_i$ k
    k = k*x
return y
```

c.

Initialization : at the start of first iteration. i = n: therefore, the R.H.S upper limit evaluates to

n-i+1 = -1 which leads to 0 summation which true as y = 0 .

Maintenance:

Let us try to prove this through induction. We denote y_x to be the value of y when i = x. In this case let us assume that above variant holds for some i=j .

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

Now as inductive step we need to prove it for j-1.

Now after next iteration

$y_{j-1} = a_j + x * y_j$

Substituting y_i , we get

$$y_{j-1} = a_j + x * \left[\sum_{k=0}^{n-(j+1)} a_{k+j+1}x^k\right] \qquad \text{-----}(1)$$

$$y_{j-1} = a_j + \sum_{k=0}^{n-(j+1)} a_{k+j+1}x^{k+1}$$

$$y_{j-1} = a_j + \sum_{u=1}^{n-j} a_{u+j}x^u \qquad [\text{ Substituting } u = k+1]$$

At u = 0 we can notice that summation will yield a_j . Hence we can combine two terms and make u start from 0 in summation.

$$= \sum_{u=0}^{n-j} a_{u+j}x^u$$

Above summation can be written as by substituting k = u , we get

$$= \sum_{u=0}^{n-((j-1)+1)} a_{u+(j-1)+1}x^u$$

Note the similarity between above and 1st eqn.

Hence our invariant holds at i.

Termination : In termination step i = - 1, substituting it to the eqn we get our desired result.

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

d. From the termination step of above part we can get that the above algorithm at termination evaluates the polynomial.

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

Q4. **Inversion**

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 five inversions of array {2,3,8,6,1}

b. What array with elements from the set {1,2,3…n} has the most no. of inversion? 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 inversion in any permutation on n elements in Θ(n lg n) worst-case time.(Hint: Modify merge sort.)

Ans.

a. The five inversion are (2,1) , (3,1) , (8,1) , (8,6) , (6,1) . Since in all these cases i < j and A[i] > A[j].

b. Let us develop intuition before answering. If the first element of the array will be the largest. It need to be swap through all remaining elements.(n-1). If the second element is largest element it need to be swap through the remaining elements on it right side. (n-2). As may have guessed now that if we continue above argument. In order to generate most no. of inversion. 1st element must be largest , 2nd must be 2nd biggest and so on. Hence a reverse sorted array(n,n-1,n-2….1) has most no. of inversion. And the no. of inversions are

$$S_n = n-1 + n-2 + n-3 + .. + 1$$

$$= \sum_{i=1}^{n-1}(n-i)$$

$$= \sum_{i=1}^{n-1}(n) - \sum_{i=1}^{n-1}(i)$$

$$= n(n-1) - \frac{n(n-1)}{2}$$

$$= \frac{n(n-1)}{2}$$

c. We know that the inner while loop of insertion sort shift the elements to left to their right position. So if there is more inversion in an array, then we need to shift more elements. Hence as the number of inversions increases, running time of insertion sort increases. Hence if g(n) denotes the no. of inversion then

$T(n) = \Theta(g(n))$, where n is the size of array.

Justification:

The running time of insertion sort is a constant times the number of inversions. Let I(i) denote the number of j < i such that A[j] > A[i]. Then $\sum^{n}_{i=1} I(i)$ equals the number of inversions in A. Now consider the while loop on lines 5-7 of the insertion sort algorithm. The loop will execute once for each element of A which has index less than j is larger than A[j]. Thus, it will execute I(j) times. We reach this while loop once for each iteration of the for loop, so the number of constant time steps of insertion sort is $\sum^{n}_{i=1} I(j)$ which is exactly the inversion number of A.

d. Explanation:

We will use our regular merge sort but with minor modification to meet our requirements. We will call our main procedure COUNT-INVERSIONS. Since during counting original array will be destroyed. We can create a copy of array before starting our main process. (I have not mentioned it in code because it depends on you. If you want to preserve it or not).

To count the total no. of inversion we need to count the inversion from right , left and then add them up to give total. This give the reason why we use Divide and Conquer.

Now an inversion occurs when we copy something from right sub-array while there are elements in left subarray. And the no. of inversions will be no. of elements left in left sub-array because only those elements will be greater than that right sub-array element.

Since there is no extra task other than trivial arithmetic its runtime is same that of merge sort.

$$T(n) = \Theta(n) + \Theta(n \lg n) = \Theta(n \lg n)$$

Pseudocode:-

```
COUNT-INVERSIONS(A, p, r)
  if p < r
    q = floor((p + r) / 2)
    left = COUNT-INVERSIONS(A, p, q)
    right = COUNT-INVERSIONS(A, q + 1, r)
    inversions = MERGE-INVERSIONS(A, p, q, r) + left + right
    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] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
inversions = 0
for k = p to r
   if L[i] <= R[j]
      A[k] = L[i]
      i = i + 1
   else
      inversions = inversions + n1 - i + 1  //Important step to be noted.
       // When the right is smaller, there are elements on the left this indicates inversion.
     // each remained element in the left makes an inversion pair.
      A[k] = R[j]
      j = j + 1
return inversions
```

Since we need to count the inversion where i<j & A[i]>A[j]. We can easily count this during the merge process. Since in merge when we compare left and right sub-array if at any step we find that element in the right subarray is smaller than the element in the left. We perform swap **or inversion** are there. Now to calculate **How** much inversion.

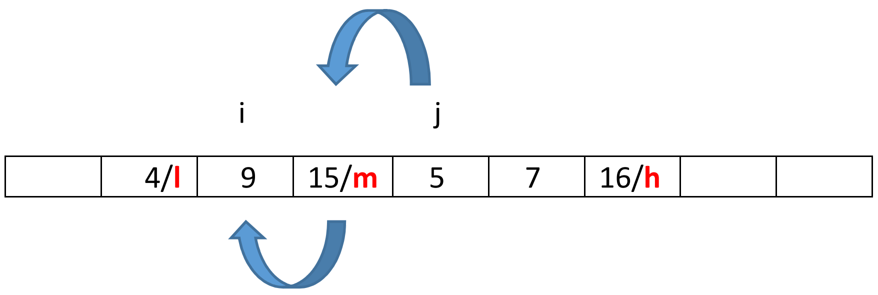
E.g.

Imagine this scenario

L = low

M = mid

H = high



Let us assume that A[l….m] is left array and A[m+1……h] is right array. Initially **i** and **j** are at 4 and 5 respectively. Since j>I and A[j] > A[i] there won't be any inversion. So we insert 4 in original array and increase

i. Now j is at 5 and i at 9. Since j > i **but** A[i] > A[j], there will be an inversion. To calculate much many inversion. We can calculate that 5 will be at the position of 9 and hence it will swap with 15 and then 9 to reach its position.

No. of swap of will be no. of elements he need to cross in left array for swap. Since we have already moved i position total no. of swaps will be, swap = (size1 – i +1). Extra one because j will be at i's positon.

If we want we can also print the number which will be swapped. Swapped number will be (15,5) and (9,5).

To visualize this add following lines in the else part(where inversion count is calculates)

```
for l= i to size1
        print (left[l] and right[j]))
         l = l +1;
```

For implementations. Click here.