**Question 1 (a)** Give a divide-and-conquer algorithm for the following problem:

• Input: Two arrays A[1 . . . n] and B[1 . . . n] of n, each array contains n elements which are

sorted in non-decreasing order.

• Output: The median (i.e., the n-th smallest element) of the 2n elements in the arrays.

**Solution**

Algorithm:

Compare the middle element of each array, i.e.

A[n/2] and B[n/2]

Throw out upper half of array with greater value and lower half of array with smaller value.

Keep doing this recursively until you are left with 2 arrays of size 1. Once you get 2 arrays containing a single element, pick the smallest one.

**Pseudocode:**

Given 2 sorted arrays of size n

s = p = 0

t = q = n

A[s..t] and B[p..q]

// Recursive calls will preserve the fact that sizeof(A[]) == sizeof(B[])

int fastMedian (int s, int t, int p, int q) {

// First find the rough middle of both arrays

int midA = (s+t) / 2;

int midB = (p+q+1) / 2;

// Base case when size of Arrays has reached 1 or 2

if (t-s <= 1)

return Median = smallest of A[midA] and B[midB]

// Compare medians; drop left half of smallest one & right half of biggest one

else if (A[midA] < B[minB])

return fastMedian(midA, t, p, midB);

else

return fastMedian(s, midA, midB, q);

}// End if function

**Question 1 (b) Show that your algorithm works correctly.**

Example

A[] = {1,2,3,7,9}

B[] = {0,1,4,9,10}

Answer: A&B sorted => {0,1,1,2,**3**,4,7,9,9,10}

Median = 3

Algorithm used to reach the answer:

(Array pointers s and p are in red, pointers t and q are in blue)

s= p = 0

t = q = n = 4

A[] = {1,2,3,7,9}

B[] = {0,1,4,9,10}

int fastMedian (int s, int t, int p, int q) {

// First find the rough middle of both arrays

int midA = (s+t) / 2 = (0+4)/2 = 2 A[] = {1,2,**3**,7,9}

int midB = (p+q+1) / 2 = (0+4+1)/2 = 2 B[] = {0,1,**4**,9,10}

// Compare medians; drop left half of smallest one & right half of biggest one

else if (A[midA] < B[minB]); (3 is < 4)

return fastMedian(midA, t, p, midB)

return fastMedian(2, 4, 0, 2)

// First find the rough middle of both arrays

New s = 2, t = 4, p = 0, q = 2

int midA = (s+t) / 2 = (2+4)/2 = 3 A[] = {1,2,3,**7**,9}

int midB = (p+q+1) / 2 = (0+2+1)/2 = 1 B[] = {0,**1**,4,9,10}

// Compare medians; drop left half of smallest one & right half of biggest one

else

return fastMedian(s, midA, midB, q);

return fastMedian(2, 3, 1, 2);

// First find the rough middle of both arrays

New s = 2, t = 3, p = 1, q = 2

int midA = (s+t) / 2 = (2+3)/2 = 2 A[] = {1,2,**3**,7,9}

int midB = (p+q+1) / 2 = (1+2+1)/2 = 2 B[] = {0,1,**4**,9,10}

// Reached base case when size of Arrays has reached 1 or 2

if (t-s <= 1) => 3 – 2 = 1 so condition is met

return Median = smallest of A[midA] and B[midB]

return Median = smallest of A[2] = 3 and B[2] = 4

return Median = 3;

}// End if function

**Question 2 Consider the following algorithm (call ZZZ here) for sorting an array A[1 . . . n] into non-decreasing order:**

**ZZZ-sort(A, i, j):**

1. if A[i] > A[j] do

2. swap A[i] and A[j]

3. if i + 1 ≥ j return

4. k ← ⌊(j − i + 1)/3⌋ % round down one third of the length of A[i . . . j]

5. ZZZ-sort(A, i + k, j) % call recursively on the last two-thirds

6. ZZZ-sort(A, i, j − k) % call recursively on the first two-thirds

7. ZZZ-sort(A, i + k, j) % call recursively on the last two-thirds again

**(a) Prove that ZZZ-sort(A, 1, n) correctly sorts the input array A[1 . . . n].**

**Proof by induction**

Let P(n): The algorithm sorts every list of size n >= 1

Base case with already sorted array of size n = 2: A[1, 2], i = 0 and j = 1

1. A[0] = 1 is not > A[1] = 2

3. i + 1 = 0 + 1 = 1 = j so return

A[1, 2] is now sorted

Base case with unsorted array of size n = 2: A[ 2, 1], i = 0 and j = 1

1. A[0] = 2 is > A[1] = 1

2. swap A[0] and A[1] leaving new array A[1,2]

3. i + 1 = 0 + 1 = 1 = j so return

A is now sorted

Induction step

Assume P(n) is true; the algorithm sorts correctly any array of size n >= 2 items

Step 5: sort the last 2/3 of the array. Assuming I.H., we have the following partitioning:

|  |  |  |
| --- | --- | --- |
| **Sub 1** | **Sub 2** | **Sub 3** |

**|----------Sub A: Unsorted-----|-----------------------Sub B: Sorted-------------------------------------|**

Assuming I.H., we can infer that EVERY element in Sub 2 < EVERY element in Sub 3.

Step 6: sort the first 2/3 of the array. Assuming I.H., we have the following partitioning:

|  |  |  |
| --- | --- | --- |
| **Sub 4** | **Sub 5** | **Sub 6** |

**|---------------------------Sub C: Sorted------------------------------------|-----Sub D: Sorted-----------|**

Assuming I.H., we can infer that EVERY element in Sub 4 < EVERY element in Sub 5. We can also deduct that every element in Sub 4 < every element in Sub 6. (worst case; after step 5, sub 2 was < than Sub 1, such that after step 6, Sub 4 = Sub 2)

Therefore, at this point, the first 1/3 of the whole array (Sub 4) is ‘absolutely’ sorted with regards to the rest of the array. i.e. all elements in Sub 4 are sorted and smaller than every other elements of the array.

Step 7: sort the last 2/3 of the array. Assuming I.H., we have the following partitioning:

|  |  |  |
| --- | --- | --- |
| **Sub 4** | **Sub 7** | **Sub 8** |

**|--Sub 4: Absolutely Sorted-|--------Sub E: Sorted, therefore also Absolutely Sorted-----|**

Assuming I.H., we can infer that EVERY element in Sub 7 < EVERY element in Sub 8. Since we deducted in Step 6 that sub 4 was absolutely sorted with regards to the whole array, now that Sub E is relatively sorted, it also has to be Absolutely sorted, since every element in Sub E > every element in Sun 4.

Therefore, the whole array is now sorted.

Since we call the function recursively for P(n/3), we will eventually reach the base case of P(2), which we proved as the base case.

**(b) Give a recurrence for the worst-case running time of ZZZ-sort on an array of length n, and use this to give a tight asymptotic (i.e., using \_-notation) bound on the worst-case running time.**

Worst case:

For every array Bx[], where Bx[] are sub arrays within A[] and B[], containing 3 elements, all the way down until the final Bx[a, b, c], where a > c.

i.e.

A[ B1[ B4[], B5[], B6[] ], B2[ B7[], B8[], B9[] ], B3[ B10[], B11[], B12[] ] ]

The worst case is found when every sub array Bx[] contains unsorted elements, forcing the recursive algorithm to run a switch at every sub level.

Running time analysis:

T[n] = 4 + 3 \* T[2n/3]

T[n] = 4 + 3 \* (4 + 3 \* T[22n/32]]

T[n] = 4 + 3 \* (4 + 3 \* (4 + 3 \* T[23n/33]))

T[n] = 4 + 3\*4 + 32\*4 + 33\*T[23n/33]

[…] T[n] = 4 + 3\*4 + 32\*4 + 33\*4 + […] + 3log3/2n\*T[1] Where T[1] = 3

T[n] = 4 (30 + 31 + 32 + 33 + … + 3log3/2n + 1)

T[n] = 4 ∑3i from i = 0 to i = log3/2n + 1

T[n] = 4 (1 – 3log3/2n+2)/(1 – 3)

T[n] = -2 (1 – 9 \* 3log3/2n)

T[n] = 18 \* 3log3/2n – 2

T[n] = 18 \* n \* 3log1/2n – 2

T[n] = k \* n \* ln(n)

So the algorithm is performed in O(n\*ln n) time