- The instructions are the same as in Homework 1, 2 and 3.

There are 6 questions for a total of 100 points.

---

1. (10 points) Given an array $A[1...n]$ containing n **distinct** integers sorted in increasing order, design an algorithm that determines if there is an index i such that $A[i] = i$. Give proof of correctness and running time analysis.

Solution:

Input: An array $A[1..n]$ containing n distinct integers sorted in increasing order.

Output: A boolean variable denoting, if there is an index i such that $A[i] = i$, or not.

Algorithm:

Procedure solve(A, n)

- Initialize two variables $low = 1$ and $high = n$.
- while $(low \leq high)$:
  - Initialize a variable $mid = low + (high - low)/2$
  - if $(A[mid] == mid)$:
    - return $true$
  - else if $(A[mid] < mid)$:
    - Set $low = mid + 1$
  - else:
    - Set $high = mid - 1$
- return $false$.

Note: In the above algorithm:

→ *= means assignment.*

→ *== means equality comparison operator.*

→ *Indentation, along with hyphen (-), denotes the scope.*

Running Time Analysis:

1. Initialization of variables takes $O(1)$ time.
2. In each iteration of the while loop, we either search in the left half or, in the right half of the array, thereby performing $O(\log_2 n)$ iterations in the worst case.
3. Returning answer takes $O(1)$ time.

Thus,

$$\begin{aligned} \Rightarrow \text{T.C.} &= O\left(1 + \log_2 n + 1\right) \\ &= O\left(c + \log n\right) \end{aligned}$$

Proof Of Correctness:

Termination Check: In each iteration of the while loop, either the value of low is increased or, the value of $high$ is decreased or, the answer is returned. Hence, the condition $low <= high$ eventually becomes $false$, and the program comes out of the while loop, thereby terminating.

Proof: The proof of correctness of our algorithm lies in the correctness of the following claim:

Claim: In any iteration if $arr[mid] = mid + k, k \in \mathbb{Z} - \{0\}$, then one half of the search space can be safely discarded, without any loss of generality.

case 1: $k < 0$

$\Rightarrow arr[mid] = mid - |k|$
$\therefore$ All the elements in the array $A[1...mid]$ will be at most $i - |k|$, $\forall\ 1 \leq i \leq mid$.
Therefore, an array element with a value equal to its index cannot exist in the left half of such an array.
Hence, we can safely discard the left half and continue our search only in the right half.

case 2: $k > 0$

$\Rightarrow arr[mid] = mid + |k|$
$\therefore$ All the elements in the array $A[mid...n]$ will be at least $i + |k|$, $\forall\ mid \leq i \leq n$.
Therefore, an array element with a value equal to its index cannot exist in the right half of such an array.
Hence, we can safely discard the right half and continue our search only in the left half.

$\therefore$ Our claim is correct.

Now our algorithm uses the above claim to discard either the left half or, the right half of the given array and uses binary search to search only one half of the array. So, the entire proof of correctness of our algorithm lies on the proof of correctness of the binary search algorithm.

Let $P(n)$ denote that the Binary Search algorithm gives correct output on an array of n integers.

For $n = 1$: $low = high = mid = 1$

case 1: $arr[1] = 1$

$\Rightarrow (arr[mid] = mid)$ condition becomes true.
$\therefore$ Our algorithm returns $true$ correctly.

case 2: $arr[1] \neq 1$

case 2a: $arr[1] < 1$

$\Rightarrow low$ gets set to 2.
$\Rightarrow (low \leq high)$ condition becomes false.
$\therefore$ Our algorithm returns $false$ correctly.

case 2b: $arr[1] > 1$

$\Rightarrow high$ gets set to 2.
$\Rightarrow (low \leq high)$ condition becomes false.
$\therefore$ Our algorithm returns $false$ correctly.

$\therefore P(1)$ holds.

Suppose that for $n = 1, 2, 3, ...k$, $P(k)$ holds.

Now,

For $n = k + 1$: Initially, $low = 1$, $high = (k + 1)$, and $mid = 1 + \lfloor \frac{k}{2} \rfloor$

case a: k is odd

$\Rightarrow mid = 1 + \frac{k-1}{2}$
$\Rightarrow mid = \frac{k+1}{2}$

case b: k is even

$\Rightarrow mid = 1 + \frac{k}{2}$
$\Rightarrow mid = \frac{k+2}{2}$

Now, there exist three possibilities:

case 1: $arr[mid] = mid$

$\Rightarrow (arr[mid] = mid)$ condition becomes true.
$\therefore$ Our algorithm returns *true* correctly.

case 2: $arr[mid] < mid$

$\Rightarrow low$ gets set to $mid + 1$.
$\Rightarrow$ Now, our code executes on $(high - mid)$ number of elements and,
$(high - mid) = \left[(k+1) - \left(\frac{k+1}{2}\right)\right]$ *or*, $\left[(k+1) - \left(\frac{k+2}{2}\right)\right]$
$= \left(\frac{k}{2} + \frac{1}{2}\right)$ *or*, $\left(\frac{k}{2} + \frac{1}{4}\right)$, both of which are less than $\left(\frac{k}{2} + 1\right)$ and thus lie in range $[1...k]$.
$\Rightarrow$ Our Algorithm returns correct output due to the Induction Hypothesis.

case 3: $arr[mid] > mid$

$\Rightarrow high$ gets set to $mid - 1$.
$\Rightarrow$ Now, our code executes on $(mid - low)$ number of elements and,
$(mid - low) = \left[\left(\frac{k+1}{2}\right) - 1\right]$ *or*, $\left[\left(\frac{k+2}{2}\right) - 1\right]$
$= \left(\frac{k}{2} - \frac{1}{2}\right)$ *or*, $\left(\frac{k}{2}\right)$, both of which are less than $\left(\frac{k}{2} + 1\right)$ and thus lie in range $[1...k]$.
$\Rightarrow$ Our Algorithm returns correct output due to the Induction Hypothesis.

$\therefore P(k+1)$ holds.

$\because [P(1) \wedge P(2) \wedge \ldots \wedge P(k)] \Rightarrow P(k+1)$
$\therefore$ P(n) is true, $\forall$ n ≥ 1
$\therefore$ Our algorithm is correct.

Q.E.D.

2. (15 points) Consider the following problem: You are given a pointer to the root r of a binary tree, where each vertex v has pointers $v.lc$ and $v.rc$ to the left and right child, and a value $Val(v) > 0$. The value NIL represents a null pointer, showing that v has no child of that type. You wish to find the path from r to some leaf that minimizes the total values of vertices along that path. Give an algorithm to find the minimum sum of vertices along such a path along with a proof of correctness and runtime analysis.

Solution:

Input: A pointer r, pointing to the root of a binary tree.

Output: An integer denoting the minimum sum of the value of nodes, in a path from root to some leaf, in the given binary tree.

Algorithm:

Procedure solve($*r$)

- if (r == NIL)
  - return 0
- if ($r \rightarrow$ lc == NIL and $r \rightarrow$ rc == NIL)
  - return $Val(r)$
- Declare two integer variables $leftSum$ and $rightSum$, both initialized to ∞.
- if ($r \rightarrow$ lc ! = NIL)
  - $leftSum = Val(r) + solve(r \rightarrow lc)$
- if ($r \rightarrow$ rc ! = NIL)
  - $rightSum = Val(r) + solve(r \rightarrow rc)$
- return min($leftSum, rightSum$).

Note: In the above algorithm:

- $\rightarrow$ *= means assignment.*
- $\rightarrow$ *== means equality comparison operator.*
- $\rightarrow$ *!= means inequality comparison operator.*
- $\rightarrow$ *Indentation, along with hyphen (-), denotes the scope.*
- $\rightarrow$ *min is a function that returns the minimum of two values in $O(1)$ time, using if-else statements.*

Running Time Analysis: Let $T(n)$ denote the running time of the above algorithm.

case 1: The binary tree is evenly balanced

So, the recurrence relation becomes:

$$\Rightarrow T(n) = 2T\left(\frac{n-1}{2}\right) + O(1)$$
$$\Rightarrow T(n) \approx 2T\left(\frac{n}{2}\right) + O(1)$$

Here,

$a = 2$, $b = 2$ and $d = 0$, which corresponds to bottom-heavy case of Master's Theorem

$$\Rightarrow T(n) = O(n^{\log_2 2})$$
$$\Rightarrow T(n) = O(n)$$

case 2: The binary tree is unbalanced

So, the recurrence relation becomes:

$$\Rightarrow T(n) = T(|L|) + T(|R|) + O(1)$$

where,

$|L|$ denotes the size of the left subtree and $|R|$ denotes the size of the right subtree

We guess that it would take $O(n)$. Now, let us try to prove this using induction.

Claim: $T(n) \leq c.n \ \forall \ n \geq 1$ and for some constant c that is bigger than $T(1)$ and bigger than the $O(1)$.

Proof:

For $n = 1$:

$$T(1) < c.(1)$$
$$c > T(1)$$

This is true by choice of c as we have chosen such a c, that satisfies the above claim.

Suppose that for some $n \geq 1$, $T(k) \leq c.k \ \forall \ k$, $1 \leq k < n$

Now,

$$T(n) = T(|L|) + T(|R|) + c.1 \leq c.|L| + c.|R| + c.1 \qquad \text{(By Induction Hypothesis)}$$
$$T(n) \leq c(|L| + |R| + 1)$$
$$T(n) \leq c.n$$

Also,

This analysis is tight as we have a worst-case input (full binary tree), which takes $O(n)$ time.

Thus,

$$T.C. = O(n)$$

Proof Of Correctness:

Termination Check: The given code recurses on each node of the binary tree exactly once. Once it reaches a leaf node, it returns its value to its parent and this way the recursion eventually terminates.

Proof: Let $P(n)$ denote that our algorithm is correct on a binary tree with n nodes.

For $n = 0$: $P(0)$ holds as our algorithm returns 0, which is correct.

Suppose that $P(0), P(1), P(2), ..., P(k)$ holds, i.e., our algorithm returns the correct output for a binary tree/subtree with v nodes, $0 \leq v \leq k$

Now,

For $n = k + 1$: Let r be the root node of a binary tree with $(k + 1)$ nodes.
Suppose the left subtree of such a binary tree has v nodes, $0 \leq v \leq k$

$\Rightarrow$ The right subtree of such a binary tree has $(k - v)$ nodes.
Now,

$$P(k+1) = Val(r) + P(\text{left binary subtree with } v \text{ nodes}) + P(\text{right binary subtree with } (k-v) \text{ nodes})$$

$$\Rightarrow P(k+1) = Val(r) + P(v) + P(k-v)$$

We know that v lies in the range $[0...k]$.
$\Rightarrow (k - v)$ also lies in the range $[0...k]$.
$\Rightarrow P(k+1)$ holds as our algorithm gives correct result for $P(v)$ and $P(k-v)$ (Induction Hypothesis).
$\therefore$ Our algorithm is correct.

Q.E.D.

3. (15 points) One ordered pair $v = (v_1, v_2)$ dominates another ordered pair $u = (u_1, u_2)$ if $v_1 \geq u_1$ and $v_2 \geq u_2$. Given a set S of ordered pairs, an ordered pair $u \in S$ is called Pareto optimal for S if there is no $v \in S$ such that v dominates u. Give an efficient algorithm that takes as input a list of n ordered pairs and outputs the subset of all Pareto-optimal pairs in S. Provide a proof of correctness along with the runtime analysis.

Solution:

Input: A set S *(an array of structures or, a vector of pairs)*, $S[0..n-1]$ containing n ordered pairs.

Output: A subset ans *(an array of structures or, a vector of pairs)* of set S, containing only pareto-optimal pairs present in S.

Algorithm:

Procedure solve1(S, n)

- Sort elements of S in non-decreasing order. *(If the first element of pair $u \in S$ and $v \in S$, $u \neq v$, is same, then sort in non-decreasing order of the second element of u and v).*
- Return Call $solve2(S, 0, n-1)$.

Procedure solve2$(v, low, high)$

- if $(low == high)$
  - Declare a dynamic data structure *(such as a vector of pairs in C++)* called $result$.
  - Add $v[low]$ to $result$.
  - Return $result$.
- Initialize a variable $mid = low + (high - low)/2$.
- Declare two dynamic data structures *(such as a vector of pairs in C++)* called $left$ and $right$ respectively,.
- $left$ = Call $solve2(v, low, mid)$.
- $right$ = Call $solve2(v, mid + 1, high)$.
- Return Call $solve3(left, right)$.

Procedure solve3$(left, right)$

- Declare a dynamic data structure *(such as a vector of pairs in C++)* called tmp.
- Add $left$ to tmp.
- Add $right$ to tmp.
- Declare a dynamic data structure *(such as a vector of pairs in C++)* called ans.
- Initialize an integer element called $currMax$ to $-\infty$.
- for each pair (u, v) in tmp *(from last to first)*:
  - if $(v >= currMax)$:
    - Add (u, v) to ans.
    - Update $currMax$ to v.
- Reverse the array of structures/vector of pairs ans.

- Return ans.

Note: In the above algorithm:

→ *= means assignment.*

→ *== means equality comparison operator.*

→ *Indentation, along with hyphen (-), denotes the scope.*

Running Time Analysis:

1. Excluding the time required for the *solve*2 procedure call, the *solve*1 procedure simply sorts the given set S of size n which takes $O(n.logn)$ time.

2. In the *solve*2 procedure call we have two steps - the "Divide" step and the "Combine" step. The "Divide" step keeps on dividing the set of size n, into equal halves until it contains only a single pair and then returns it in $O(1)$ time. The "Combine" step takes time equal to the time taken by the *solve*3 procedure call.

   Let $T(n)$ denote the time required for the *solve*2 procedure call.

   So,

$$T(n) = 2T\left(\frac{n}{2}\right) + O(\text{time taken by } solve3)$$

3. In the *solve*3 procedure call, we just iterate over the pairs of set S of size n once, and keep adding Pareto-optimal pairs to our answer, which takes $O(n)$ time in the worst case.

   Hence,

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

   Here,
   $a = 2$, $b = 2$ and $d = 1$, which corresponds to the steady-state case of Master's Theorem

$$\Rightarrow T(n) = O(n^d . \log_b n)$$
$$\Rightarrow T(n) = O(n . \log n)$$

   Thus,

$$\Rightarrow \text{T.C.} = O\left(n . \log n + n . \log n\right)$$
$$= O\left(n . \log n\right)$$

Proof Of Correctness:

Termination Check: The algorithm keeps on dividing the array/vector of n pairs, until it becomes an array/vector with a single pair. Once this happens, it simply combines these n arrays/vectors of single pairs into a single array/vector of Pareto-optimal pairs, thereby eventually terminating.

Proof: The proof of correctness of our algorithm lies in the correctness of the following claims:

Claim: In a set, sorted in non-decreasing order, if (a, b), $(c, d) \in S$ be two ordered pairs at indices i and j respectively, $0 \leq i < j \leq (n-1)$, then (a, b) can never dominate over (c, d).

Since the set S is sorted in non-decreasing order, hence the possible cases are:

case 1: $a < c$ **and** $b \leq d$

Clearly in this case, (a, b) cannot dominate over (c, d) as it follows from definition

case 2: $a = c$ **and** $b < d$

Clearly in this case, (a, b) cannot dominate over (c, d) as it follows from definition

case 3: $a = c$ **and** $b = d$

This case is impossible as S is a set.

$\therefore$ Our claim is correct.

Claim: In a set, sorted in non-decreasing order, if (a, b), $(c, d) \in S$ be two ordered pairs at indices i and j respectively, $0 \leq i < j \leq (n-1)$, then (a, b) will not be dominated by (c, d) iff $a < c$ but $b > d$.

This claim is correct trivially via the definition of Pareto-optimal pairs.

Hence, only in this case (a, b) can form a Pareto-optimal pair.

Now our algorithm uses the above claims to first find Pareto-optimal pairs for the left half and the right half of the array. Then, it finds Pareto-optimal pairs of the original array using the above claims by traversing the array in the reverse direction.

Let $P(n)$ denote that our algorithm gives correct output on an array of n integers.

For $n = 1$: $low = high = 0$
$\because$ Every pair is a Pareto-optimal pair in itself.
$\therefore$ Our algorithm returns $v[0]$ which is correct.
$\therefore P(1)$ holds.

Suppose that for $n = 1, 2, 3, 4, ...k$, $P(k)$ holds.

Now,

For $n = k + 1$: Initially, $low = 0$, $high = k$, and $mid = \lfloor \frac{k}{2} \rfloor$

case 1: k **is odd**

$\Rightarrow mid = \frac{k-1}{2}$
$\Rightarrow mid = \frac{k}{2} - \frac{1}{2}$

case 2: k **is even**

$\Rightarrow mid = \frac{k}{2}$

$\Rightarrow$ Now, our code executes on $(mid - low + 1)$ and $(high - mid)$ number of elements and,

$$(mid - low + 1) = \left[\frac{k}{2} - \frac{1}{2} - 0 + 1\right] \text{ or, } \left[\frac{k}{2} - 0 + 1\right]$$
$$= \left(\frac{k}{2} + \frac{1}{2}\right) \text{ or, } \left(\frac{k}{2} + 1\right) \text{ , both of which are less than } \left(\frac{k}{2} + 2\right) \text{ and thus lie in range } [1...k].$$

Similarly,

$$(high - mid) = \left[k - \left(\frac{k-1}{2}\right)\right] or, \left[k - \frac{k}{2}\right]$$
$$= \left(\frac{k}{2} + \frac{1}{2}\right) or, \left(\frac{k}{2}\right) \text{ , both of which are less than } \left(\frac{k}{2} + 1\right) \text{ and thus lie in range } [1...k].$$

$\because$ Our Algorithm returns correct output for both these cases (Induction Hypothesis).
$\therefore$ Value of $left$ and $right$ is calculated correctly by our algorithm.

Also, we proved that our combine step is correct as it uses the above-proven claims.
$\therefore$ $P(k+1)$ holds.

$\because [P(1) \wedge P(2) \wedge \ldots \wedge P(k)] \Rightarrow P(k+1)$
$\therefore$ P(n) is true, $\forall$ n ≥ 1
$\therefore$ Our algorithm is correct.

Q.E.D.

4. (15 points) Let S and T be sorted arrays each containing n elements. Design an algorithm to find the n^{th} smallest element out of the $2n$ elements in S and T. Discuss running time, and give proof of correctness.

Solution:

Input: Two sorted arrays $S[1..n]$ and $T[1..n]$ containing n elements each.

Output: An integer representing n^{th} smallest element in $S \cup T$

Algorithm:

Procedure solve(S, T, n)

- Initialize two variables $low = 0$ and $high = n$.
- while $(low \leq high)$:
  - Initialize a variable $mid1 = low + (high - low)/2$
  - Initialize a variable $mid2 = n - mid1$
  - Initialize two variables $l1$ and $l2$ to $-\infty$
  - Initialize two variables $r1$ and $r2$ to $+\infty$
  - if $(mid1 < n)$:
    - Set $r1 = S[mid1 + 1]$
  - if $(mid2 < n)$:
    - Set $r2 = T[mid2 + 1]$
  - if $(mid1 - 1 \geq 1)$:
    - Set $l1 = S[mid1]$
  - if $(mid2 - 1 \geq 1)$:
    - Set $l2 = T[mid2]$
  - if $(l1 \leq r2$ and $l2 \leq r1)$:
    - return max(l1, l2)
  - else if $(l1 > r2)$:
    - Set $high = mid1 - 1$
  - else
    - Set $low = mid1 + 1$
- return any bogus value (e.g. -1)

Note: In the above algorithm:

→ *= means assignment.*

→ *Indentation, along with hyphen (-), denotes the scope.*

Running Time Analysis:

1. Initialization of variables takes $O(1)$ time.
2. In each iteration of the while loop, we either search in the left half or, in the right half of the array, thereby performing $O(\log_2 n)$ iterations in the worst case.
3. Returning answer takes $O(1)$ time.

$$\Rightarrow \text{T.C.} = O\left(1 + \log_2 n + 1\right)$$
$$= O\left(c + \log n\right)$$

Proof Of Correctness:

Termination Check: In each iteration of the while loop, either the value of *low* is increased or, the value of *high* is decreased or, the answer is returned. Hence, the condition $low <= high$ eventually becomes $false$, and the program comes out of the while loop, thereby terminating.

Proof: The proof of correctness of our algorithm lies in the correctness of the following claim:

f**Claim:** There exists a unique number of elements that must be picked from the 1^{st} array (k) and the 2^{nd} array ($n-k$), so as to create left and right halves of a correct single-sorted merged array.

Let M denote the correct single-sorted merged array created from given arrays such that $M[n] = mid$. Let L denote the left half and R denote the right half of M such that $L = M[1...mid]$, $R = M[mid+1...n]$ and, $L \cup R = M$.
We know that, if we merge the given arrays, then the n^{th} smallest element will be equal to mid and, mid will be unique as M will be unique.
Also, the configuration of L must be created by picking k elements from the 1^{st} array and $(n-k)$ elements from the 2^{nd} array, $0 \leq k \leq n$.
Assume that there are multiple possible values of k that can be used to create L and R.
$\Rightarrow$ There are multiple possible configurations of L and R.
$\Rightarrow$ There are multiple possible configurations of M as $M = L \cup R$
But this is impossible as we know that M must be unique (contradiction).
$\Rightarrow$ Our assumption was wrong.
$\Rightarrow$ There is only a unique value of k possible.
$\therefore$ Our claim is correct.

Now our algorithm uses the above claim to discard either the left half or, the right half of the search space of k, i.e., $k \in [0...n]$ and, uses binary search to determine the exact value of k. So, the entire proof of correctness of our algo lies in the proof of correctness of the binary search algorithm, which is correct *(proved in question 1.)*
Also, we use a simple fact to decide if the current value of k is correct or not, i.e.,
Suppose L and R denote unique left and right half configurations of M, such that $L = S[1...k] \cup T[1...(n-k)]$, $R = S[(k+1)...n] \cup T[(n-k+1)...n]$, and $L \cup R = M = S \cup T$.
$\because$ We know that, S and T are sorted.
$\Rightarrow S[k] \leq S[k+1]$ and $T[n-k] \leq T[n-k+1]$, always.
Thus, the current value of k will be correct iff it leads to a valid configuration, i.e.,
$S[k] \leq T[n-k+1]$ and $T[n-k] \leq S[k+1]$, always.
$\Rightarrow$ By performing the above check, we can determine if the current value of k is correct or not.
$\therefore$ Our algorithm is correct.

Q.E.D.

5. An array $A[1...n]$ is said to have a majority element if more than half (i.e., $> n/2$) of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] \geq A[j]$?" (Think of the array elements as GIF files, say.) However you can answer questions of the form: "is $A[i] = A[j]$?" in constant time.

   (a) (15 points) Show how to solve this problem in $O(n \log n)$ time. Provide a runtime analysis and proof of correctness.
   (*Hint: Split the array A into two arrays $A1$ and $A2$ of half the size. Does knowing the majority elements of $A1$ and $A2$ help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.*)

Solution:

Input: An array $A[1..n]$ containing n elements, which need not be from some ordered domain (like integers).

Output: The majority element of the array, if it exists, else return some bogus character such as -1.

Algorithm:

Procedure solve1(A, n)

- Declare a variable called res.
- $res = solve2(A, n)$.
- if $(res! = -1)$
  - Output "res" as the majority element.
- else
  - Output "No majority element exists".

Procedure solve2(arr, n)

- Declare a variable called $candidate$.
- $candidate = solve3(arr, 1, n)$.
- Initialize a variable $cnt = 0$.
- for each ele in arr:
  - if $(ele == candidate)$
    - Increment the value of cnt by 1.
- if $(cnt > n/2)$
  return $candidate$.
- else
  return -1.

Procedure solve3$(arr, low, high)$

- if $(low == high)$
  return $arr[low]$.
- Initialize a variable $mid = low + (high - low)/2$.

- Declare two variables $lMajority$ and $rMajority$.
- $lMajority = solve3(arr, low, mid)$
- $rMajority = solve3(arr, mid + 1, high)$
- if $(lMajority == rMajority)$
  return $lMajority$.
- Initialize two variables $lcnt$ and $rcnt$ to 0.
- for $i = low, low + 1, low + 2, ..., high$:
  if $(arr[i] == lMajority)$
  Increment value of $lcnt$ by 1.
  else if $(arr[i] == rMajority)$
  Increment value of $rcnt$ by 1.
- if $(lcnt > rcnt)$
  - return $lMajority$.
- else
  - return $rMajority$.

Note: In the above algorithm:

→ *= means assignment.*

→ *== means equality comparison operator.*

→ *!= means inequality comparison operator.*

→ *Indentation, along with hyphen (-), denotes the scope.*

Running Time Analysis:

1. Excluding the time required for the $solve2$ procedure call, the $solve1$ procedure just performs some constant comparisons which takes $O(1)$ time.
2. Excluding the time required for the $solve3$ procedure call, the $solve2$ procedure just performs a single traversal of the array of size n, which takes $O(n)$ time.
3. Let $T(n)$ denote the time required for the $solve3$ procedure call. Hence,

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Here,
$a = 2$, $b = 2$ and $d = 1$, which corresponds to the steady-state case of Master's Theorem

$$\Rightarrow T(n) = O(n^d . \log_b n)$$
$$\Rightarrow T(n) = O(n . \log n)$$

Proof Of Correctness:

Termination Check: The algorithm keeps on dividing the given array into smaller halves until an array with a single element is achieved. Once this happens, the algorithm keeps on returning

the majority elements of smaller sub-arrays, until it finally returns the majority element of the original array, thereby eventually terminating.

Proof: The proof of correctness of our algorithm lies in the correctness of the following claim:

Claim: If the majority element exists in our original array, then it must also be the majority element of at least one of the smaller subarrays, created by dividing the original array into almost equal halves.

We know that, if the majority element exists in the original array A of size n, then it must be present $> \lfloor n/2 \rfloor$ times in A, or, at least $\lfloor n/2 \rfloor + 1$ times in A.
Suppose x be the majority element of A such that it is present $\lfloor n/2 \rfloor + 1$ times in A.
Let L and R denote the left and right halves created by the divide step in the *solve*3 procedure, such that x is present k times in L and $(\lfloor n/2 \rfloor + 1 - k)$ times in R, $1 \leq k \leq \lfloor n/2 \rfloor$.
Now, there exists three possibilities:

case 1: $k = \lfloor n/4 \rfloor - i$**,** $i \in [1...\lfloor n/4 \rfloor]$

$\Rightarrow$ x is present in the left half $(\lfloor n/4 \rfloor - i)$ times and, in the right half $(\lfloor n/2 \rfloor + 1 - \lfloor n/4 \rfloor + i)$ times.
$\Rightarrow$ x is present in the left half $(\lfloor n/4 \rfloor - i)$ times and, in the right half $(\lfloor n/4 \rfloor + 1 + i)$ times, $i \in [1...\lfloor n/4 \rfloor]$.
$\Rightarrow$ x is the majority element of the right half of the array ($> \lfloor n/4 \rfloor$ times).

case 2: $k = \lfloor n/4 \rfloor$

$\Rightarrow$ x is present in the left half $(\lfloor n/4 \rfloor)$ times and, in the right half $(\lfloor n/2 \rfloor + 1 - \lfloor n/4 \rfloor)$ times.
$\Rightarrow$ x is present in the left half $(\lfloor n/4 \rfloor)$ times and, in the right half $(\lfloor n/4 \rfloor + 1)$ times.
$\Rightarrow$ x is the majority element of the right half of the array ($> \lfloor n/4 \rfloor$ times).

case 3: $k = \lfloor n/4 \rfloor + i$**,** $i \in [1...\lfloor n/4 \rfloor]$

$\Rightarrow$ x is present in the left half $(\lfloor n/4 \rfloor + i)$ times and, in the right half $(\lfloor n/2 \rfloor + 1 - \lfloor n/4 \rfloor - i)$ times.
$\Rightarrow$ x is present in the left half $(\lfloor n/4 \rfloor + i)$ times and, in the right half $(\lfloor n/4 \rfloor + 1 - i)$ times, $i \in [1...\lfloor n/4 \rfloor]$.
$\Rightarrow$ x is the majority element of the left half of the array ($> \lfloor n/4 \rfloor$ times).

$\therefore$ Our claim is correct.

Now, our algorithm uses the above claim to find a suitable candidate for the majority element of the array and returns if the majority element exists or not, by simply counting the frequency of the candidate in the array/subarray.
$\therefore$ Our algorithm is correct.

Q.E.D.

(b) (15 points) Design a linear time algorithm. Provide a runtime analysis and proof of correctness. (*Hint: Here is another divide-and-conquer approach:*

- *Pair up the elements of A arbitrarily, to get $n/2$ pairs (say n is even)*
- *Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them*
- *Show that after this procedure there are at most $n/2$ elements left, and that if A has a majority element then it's a majority in the remaining set as well)*

Solution:

Input: An array $A[1..n]$ containing n elements, which need not be from some ordered domain (like integers).

Output: The majority element of the array, if it exists, else return some bogus character such as -1.

Algorithm:

Procedure solve1(A, n)

- Declare a dynamic data structure *(such as a vector in C++ or, an ArrayList in Java)* called v.
- for each *ele* in *arr*:
  - Add *ele* to v.
- Initialize two variables $cand1$ and $cand2$ to -1.
- if (n is odd):
  - Set $cand2 = v[n-1]$.
  - Declare a dynamic data structure *(such as a vector in C++)* called *tmp*.
  - Add all the elements of v to *tmp*, except the last element.
  - $cand1$ = Call $solve2(tmp)$.
- else:
  - $cand1$ = Call $solve2(v)$.
- if ($cand1 != -1$):
  - if (Call $solve3(v, n, cand1) == true$):
    - Output "$cand1$" as the majority element.
  - else
    - Output "No majority element exists".
- else if ($cand2 != -1$):
  - if (Call $solve3(v, n, cand2) == true$):
    - Output "$cand2$" as the majority element.
  - else
    - Output "No majority element exists".
- else:
  - Output "No majority element exists".

Procedure solve2(v)

- Initialize a variable equal to the size of the size of the vector v.

- if ($n == 0$)
  - return -1.
- if ($n == 1$)
  - return $v[0]$.
- Declare a dynamic data structure *(such as a vector in C++ or, an ArrayList in Java)* called *tmp*.
- for $i = 0, 2, 4, 6, ...$:
  if ($i == n - 1$ or $v[i] == v[i + 1]$)
  Add $v[i]$ to *tmp*.
- Return Call $solve2(tmp)$.

Procedure solve3($v, n, cand$)

- Initialize a variable $cnt = 0$.
- for each *ele* in v:
  - if ($ele == cand$):
    - Increment the value of *cnt*.
- if ($cnt > n/2$)
  - Return $true$
- else
  - Return $false$

Note: In the above algorithm:

→ *= means assignment.*

→ *== means equality comparison operator.*

→ *!= means inequality comparison operator.*

→ *Indentation, along with hyphen (-), denotes the scope.*

Running Time Analysis:

1. Excluding the time required for the $solve2$ procedure call and $solve3$ procedure call, The $solve1$ procedure just performs some constant traversals of the array/vector of size n, which takes $O(n)$ time.
2. Let $T(n)$ denote the time required for the $solve2$ call. Hence,

$$T(n) = T(n/2) + O(n)$$

Here,

$a = 1$, $b = 2$ and $d = 1$, which corresponds to the top-heavy case of Master's Theorem

$$\Rightarrow T(n) = O(n^d)$$
$$\Rightarrow T(n) = O(n)$$

3. The *solve*3 procedure just performs a single traversal of the array/vector of size n, which takes $O(n)$ time.

$$\Rightarrow \text{T.C.} = O(n + n + n)$$
$$\Rightarrow \text{T.C.} = O(n)$$

Proof Of Correctness:

Termination Check: The *solve*2 procedure keeps on dividing the original array into almost half the size of the original array, until a single element or no element remains, thereby eventually terminating.

Proof: The proof of correctness of our algorithm lies in the correctness of the following claims:

Claim: In an even-sized array, the majority element is bound to repeat in at least one pair, if we pair up elements arbitrarily.

Let x be the majority element in an array A of size n, i.e., x, will occur at least $[n/2+1]$ times in A.
Suppose x does not repeat in any of the $n/2$ pairs, i.e., x is present exactly once in each pair.
$\Rightarrow x$ is present $\underbrace{1+1+1+\ldots+1}_{n/2 \text{ times}} =$ n/2 times in A.
But, x was supposed to be present $[n/2+1]$ times in A. (contradiction).
$\Rightarrow$ Our assumption was wrong.
$\Rightarrow x$ repeats once in at least any one of the $n/2$ pairs.
$\therefore$ Our claim is correct.

Claim: In an odd-sized array, the majority element will either be the last element $A[n]$ or, it will repeat in in at least one pair, if we pair up elements of the array $A[1...(n-1)]$ arbitrarily.

Let x be the majority element in an array A of size n, i.e., x, will occur at least $[n/2+1]$ times in A.

case 1: $A[n] \neq x$

$\Rightarrow$ Our algorithm recurses on $A[1...n-1]$ *(even-sized array)* and will return the correct majority element as it has been proved in the above claim.

case 2: $A[n] = x$

$\because$ Majority element, if exists, will always be unique (only one element can repeat more than $n/2$ times in an array of size n).
$\Rightarrow$ In worst case, x will be present in each arbitrary pair in $A[1...n-1]$ exactly once.
$\Rightarrow$ The call on $A[1...n-1]$ will either return no candidate *(in the worst case)* or, will return x *(in other cases)*.
Also, our algorithm considers $A[n] = x$ as candidate for majority element as well.
$\therefore$ Either way, the algorithm will return the correct result as it considers the right candidate for majority element checking.

Now our algorithm uses the above claims to find candidates for the given array, using divide and conquer paradigm and uses a simple frequency check for majority element checking.
$\therefore$ Our algorithm is correct.

Q.E.D.

6. Consider the following algorithm for deciding whether an undirected graph has a Hamiltonian Path from x to y, i.e., a simple path in the graph from x to y going through all the nodes in G exactly once. ($N(x)$ is the set of neighbors of x, i.e. nodes directly connected to x in G).

```
HamPath(graph G, node x, node y)
  - If x = y is the only node in G return True.
  - If no node in G is connected to x, return False.
  - For each z ∈ N(x) do:
      - If HamPath(G − {x}, z, y), return True.
  - return False
```

(a) (7 points) Give proof of correctness of the above backtracking algorithm.

Solution:

Proof Of Correctness:

Termination Check:

Proof: Let $P(n)$ denote that our algorithm is correct on a graph of n vertices.

For $n = 1$: $P(1)$ holds trivially as our algorithm returns *true*, which is correct.

Suppose that for some $k > 1$, $P(k)$ holds, $\forall\ k,\ 1 \leq k < n$, i.e., our algorithm is correct.

Now,

For $n = k + 1$: Let $x \in V$ be any node in graph $G = (V, E)$ with set of neighbours as $N(x)$.
Let z_i denote i^{th} element of set $N(x)$, $1 \leq i \leq |N(x)|$, such that $\{x, z_i\} \in E$.
Let the Hamiltonian path be abbreviated as HP.
Now, For any node x, we can break $P(k+1)$ as:

$$\begin{aligned} P(k+1) &= \text{ HP from } x \text{ to } z_i + \text{ HP from } z_i \text{ to } y \text{ in remaining graph excluding } x \\ \Rightarrow P(k+1) &= \text{ HP from } x \text{ to } z_i + \text{ HP in a Graph of } k \text{ vertices} \\ \Rightarrow P(k+1) &= \text{ HP from } x \text{ to } z_i + P(k) \end{aligned}$$

We know that $P(k)$ gives the correct result. (By Induction Hypothesis)
Hence, $P(k+1)$ is correct as $\{x, z_i\}$ is an edge (Hamiltonian Path from x to z_i, $\forall\ z_i \in N(x)$).
$\therefore$ Our algorithm is correct.

Q.E.D.

(b) (8 points) If every node of the graph G has degree (number of neighbors) at most 4, how long will this algorithm take at most? *(Hint: you can get a tighter bound than the most obvious one.)*

Solution: In the worst case, consider a 4-regular graph $G = (V, E)$, in which a Hamiltonian Path from x to y exists, $x, y \in V$.
Now, when `HamPath` function is called on x, it will lead to four recursive calls on all four neighbors of x [elements of $N(x)$].
But, when `HamPath` function is called on $z_i \in N(x)$, then it will be called on a graph $G = (V - x, E - \{x, z_i\})$.
Therefore, z_i will only have 3 neighbors to call `HamPath` function to on.
This process will repeat and `HamPath` function will be called thrice on most of the nodes of graph G, even in the worst case.
Let $T(n)$ denote the time taken by our algorithm if every node of the graph G has a degree at most 4.
Hence, in the worst case:

$$T(n) \approx \begin{cases} 3T(n-1)+1 & , n > 1 \\ 1 & , \text{otherwise} \end{cases}$$

$$\Rightarrow T(n) = 3T(n-1) + 1$$
$$\Rightarrow T(n) = 3[3T(n-2)+1] + 3^0$$
$$\Rightarrow T(n) = 3^2T(n-2) + 3^1 + 3^0$$
$$\Rightarrow T(n) = 3^2[3T(n-3)+1] + 3^1 + 3^0$$
$$\Rightarrow T(n) = 3^3T(n-3) + 3^2 + 3^1 + 3^0$$
$$\vdots$$
$$\Rightarrow T(n) = 3^kT(n-k) + 3^{k-1} + ... + 3^2 + 3^1 + 3^0$$

Now,

$$n - k = 1$$
$$\Rightarrow k = n - 1$$

Thus,

$$\Rightarrow T(n) = 3^{n-1}T(1) + 3^{n-2} + ... + 3^2 + 3^1 + 3^0$$
$$\Rightarrow T(n) = 3^{n-1}.1 + 3^{n-2} + ... + 3^2 + 3^1 + 3^0$$
$$\Rightarrow T(n) = 3^0 + 3^1 + 3^2 + ... + 3^{n-1}$$
$$\Rightarrow T(n) = \frac{1(3^n - 1)}{3 - 1}$$
$$\Rightarrow T(n) = \frac{(3^n - 1)}{2}$$
$$\Rightarrow T(n) = O(3^n)$$