- 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: Here is the pseudocode for a divide and conquer algorithm which is based on the idea of binary search:

```
FindIndex($A, n$)
  - return(FindBetweenIndices($A, 1, n$))
FindBetweenIndices($A, i, j$)
  - if ($j < i$)return("no")
  - $mid \leftarrow \lfloor \frac{i+j}{2} \rfloor$
  - if ($A[mid] == mid$)return("yes")
  - elseif ($A[mid] < mid$)return(FindBetweenIndices($A, mid + 1, j$))
  - else return(FindBetweenIndices($A, i, mid - 1$))
```

Correctness: We prove correctness using induction on the array size. The base case holds since for arrays containing no elements the algorithm correctly returns "no". The induction hypothesis says that the algorithm returns the correct answer for any array of size $0, 1, ..., n-1$. In the induction step, we need to show that the algorithm is correct for any array of size n. The algorithm checks if $A[\lceil n/2 \rceil] = \lceil n/2 \rceil$. If so, it correctly returns "yes". Otherwise, consider the following two cases:

- if $A[\lceil n/2 \rceil] > \lceil n/2 \rceil$, then any index j such that $A[j] = j$ cannot satisfy $j \geq \lceil n/2 \rceil$. This is because $A[\lceil n/2 \rceil + t] \geq A[\lceil n/2 \rceil] + t > \lceil n/2 \rceil + t$ for any positive t because the elements of the array A are distinct. So, the answer is the same as the answer for the first half of the array, which by the induction assumption, the algorithm correctly returns.
- if $A[\lceil n/2 \rceil] < \lceil n/2 \rceil$, then any index j such that $A[j] = j$ cannot satisfy $j \leq \lceil n/2 \rceil$. This is because $A[\lceil n/2 \rceil - t] \geq A[\lceil n/2 \rceil] - t < \lceil n/2 \rceil - t$ for any positive t because the elements of the array A are distinct. So, the answer is the same as the answer for the second half of the array, which by the induction assumption, the algorithm correctly returns.

So, the algorithm returns the correct answer for any array of size n. This completes the inductive step and the proof of correctness of the algorithm.

Running time: The recurrence relation for the running time of the above recursive algorithm is $T(n) = T(n/2) + O(1)$ and $T(0) = T(1) = O(1)$. Using the Master theorem with $a = 1; b = 2; d = 0$, we obtain $T(n) = O(\log n)$.

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: A divide-and-conquer algorithm for this problem is:

```
MinPath($r$)
  - if ($r.lc \neq NIL$)
      - $leftweight$ :=MinPath($r.lc$)
  - else
      - $leftweight := 0$
  - if ($r.rc \neq NIL$)
      - $rightweight$ :=MinPath($r.rc$)
  - else
      - $rightweight := 0$
  - if($leftweight \neq 0$ and $rightweight \neq 0$)
      - return($\min(leftweight, rightweight) + Val(r)$)
  - else
      - return($\max(leftweight, rightweight) + Val(r)$)
```

Correctness Proof: We prove the correctness of this algorithm by strong induction on the size of the tree. If the tree is size 1, both the left child and right child are NIL So we set $leftweight$ and $rightweight$ to 0, and return $Val(r)$. Since the only path to a leaf is the path just containing r itself and no edges, that is the correct minimum path value.

Assume `MinPath` returns the minimum total weight in any tree of size $1 \leq k \leq n-1$. We'll show that this is also true for trees of size n. If the root r of a tree of size n has both a left sub-tree and a right sub-tree, the set of paths from r to a leaf are all the paths that go from r to the left child to a leaf, together with all the paths that go from r to the right child to a leaf. Both types of paths have total value the total values along the path from the child to the leaf + $Val(r)$. Thus, the minimum path length for r is $Val(r)$ + smaller of minimum path value for the two children. By the induction hypothesis, since each subtree is of size between 1 and $n-1$, `MinPath(`$r.lc$`)` returns the minimum path value for the left child of r and `MinPath(`$r.rc$`)` that for the right child of r, so this is also what `MinPath(`r`)` returns.

Similarly, if r only has one child, the minimum total value along a path for r is $Val(r)$ plus the minimum total value for that child. Then one of the terms $leftweight$, $rightweight$ is 0, and by the induction hypothesis, the other is this total value for the child. Since all the values are non-negative, the minimum path for the child is > 0, so we return that value + $Val(r)$, which is the correct value for the tree rooted at r.

Time analysis: Let L be the size of the left-subtree, R the size of the right-subtree. Then $n = L + R + 1$, since all vertices are either in the left-subtree, right sub-tree or are the root, and the total time of the algorithm is $T(n) = T(L) + T(R) + c$ for some c, and $T(1) = c'$. For $C = max(c, c')$, we'll show by strong induction that $T(n) \leq Cn$. This is true for $n = 1$ since $C \geq c'$. Assume it is true for all $1 \leq k \leq n-1$. Then $T(n) \leq T(L) + T(R) + c \leq CL + CR + c \leq C(L + R + 1) = Cn$. So by strong induction, this is true for all n, meaning $T(n) \in O(n)$.

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:

Algorithm Description: Given an input of $(x_1, y_1), \ldots, (x_n, y_n)$, if $n = 1$, return the single ordered pair (x_1, y_1), otherwise sort the ordered pairs by their x values. Use the y values as a secondary key to break ties in x values. Let $m = \lfloor n/2 \rfloor$ and split the input into $L = (x_1, y_1), \ldots, (x_m, y_m)$ and $U = (x_{m+1}, y_{m+1}), \ldots, (x_n, y_n)$. Then recursively find PL, PU, the pareto max subset of L, U, recursively. Then let yU be the maximum y value of U and let PLy be all the ordered pairs in PL that have a larger y value than yU. Then return $PLy \cup PU$.

Correctness: The base case works. Since all x values of L are lower than all x values in U, this means that there are no ordered pairs in L that dominate any ordered pair in U so all ordered pairs in the pareto max subset of U, PU must also be in the pareto max subset of the original input. Each ordered pair in PL has a lower x value than all ordered pairs in U so in order for an ordered pair in PL to be in the pareto max of the original set, it must have a higher y value than all ordered pairs of U. So, PLy is the set of all ordered pairs in PL that have a larger y value than all the ordered pairs in U.

Runtime: There is the cost of sorting. But this can be done as a preprocessing step. Then in the algorithm there are 2 recursive calls each of size $n/2$ and the non-recursive part of finding the max y value of U and finding all ordered pairs in PL that have a larger y value than the largest y value of U all can be done in $O(n)$. So this recursion has $a = 2, b = 2, d = 1$ and the algorithm will take $O(n \log(n))$.

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: The following algorithm outputs the n^{th} smallest element.

```
NthElement($S, T, n$)
  - If $(n = 1)$return$(\min(S[1], T[1]))$
  - $i \leftarrow \lfloor n/2 \rfloor$; $j \leftarrow \lceil n/2 \rceil$
  - If $(S[i] = T[j])$return$(T[j])$
  - If $(S[i] < T[j])$
    - $A \leftarrow S[i+1]...S[n]$; $B \leftarrow T[1]...T[j-1]$
    - return(NthElement($A, B, n - \lfloor n/2 \rfloor$))
  - Else
    - $A \leftarrow S[1]...S[i-1]$; $B \leftarrow T[j+1]...T[n]$
    - return(NthElement($A, B, n - \lfloor n/2 \rfloor$))
```

When $n = 1$, then the smallest element in arrays S and T is just $\min(S[1], T[1])$. Let $i = \lfloor n/2 \rfloor$ and $j = \lceil n/2 \rceil$. If $S[i] = T[j]$, then $T[j]$ is the n^{th} smallest number since there are $(i + j - 1) = n - 1$ numbers that are $\leq T[j]$ when S and T are combined. If $S[i] < T[j]$, then for any $1 \leq k \leq i$ $S[k]$ cannot be the n^{th} smallest number since the number of integers that are larger than $S[k]$ is at least $n - k + n - j + 1 \geq n - i + n - j + 1 = n + 1$. Similarly, for any $j \leq k \leq n$, $T[k]$ cannot be the n^{th} smallest number. So, when $S[i] < T[j]$, then the n^{th} smallest integer in S and T combined is the same as the $(n - \lfloor n/2 \rfloor)^{th}$ smallest integer in $S[i+1]....S[n]$ and $T[1]....T[j-1]$. Also, when $S[i] > T[j]$, then the n^{th} smallest integer in S and T combined is the same as the $(n - \lfloor n/2 \rfloor)^{th}$ smallest integer in $S[1]....S[i-1]$ and $T[j+1]....T[n]$.

Running time: The recurrence relation for the running time is given by $T(n) \leq T(\lceil n/2 \rceil) + O(1)$ and $T(1) = O(1)$ which solves to $T(n) = O(\log n)$.

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.*)

   (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:

(a) *Solving the problem in $O(n \log n)$ time.*

Algorithm:

```
function majority ($A[1 \dots n]$)
if $n = 1$:  return $A[1]$
let $A_L, A_R$ be the first and second halves of $A$
$M_L$ = majority($A_L$) and $M_R$ = majority($A_R$)
if $M_L$ is a majority element of $A$:
   return $M_L$
if $M_R$ is a majority element of $A$:
   return $M_R$
return ''no majority''
```

Running time: $T(n) = 2T(n/2) + O(n) = O(n \log n)$.

Correctness: Suppose we divide array A into two halves, A_L and A_R. Then:

$$\begin{aligned} A \text{ has a majority element } x &\iff x \text{ appears more than } n/2 \text{ times in } A \\ &\implies x \text{ appears more than } n/4 \text{ times in either } A_L \text{ or } A_R \text{ (or both)} \\ &\iff x \text{ is a majority element of either } A_L \text{ or } A_R \text{ (or both)} \end{aligned}$$

(b) **Algorithm:**

```
function majority ($A[1 \dots n]$)
$x$ = prune($A$)
if $x$ is a majority element of $A$:
   return $x$
else:
   return ''no majority''
```

```
function prune ($S[1 \dots n]$)
if $n = 1$:  return $S[1]$
if $n$ is odd:
   if $S[n]$ is a majority element of $S$:  return $S[n]$
   $n = n - 1$
$S' = [\ ]$ (empty list)
for $i = 1$ to $n/2$:
   if $S[2i-1] = S[2i]$:  add $S[2i]$ to $S'$
return prune($S'$)
```

Correctness: We'll show that each iteration of the **prune** procedure maintains the following invariant: if x is a majority element of S then it is also a majority element of S'. The rest then follows.

Suppose x is a majority element of S. In an iteration of **prune**, we break S into pairs. Suppose there are k pairs of Type One and l pairs of Type Two:

- Type One: the two elements are different. In this case, we discard both.
- Type Two: the elements are the same. In this case, we keep one of them.

Since x constitutes at most half of the elements in the Type One pairs, x must be a majority element in the Type Two pairs. At the end of the iteration, what remains are l elements, one from each Type Two pair. Therefore x is the majority of these elements.

Running time. In each iteration of **prune**, the number of elements in S is reduced to $l \leq |S|/2$, and a linear amount of work is done. Therefore, the total time taken is $T(n) \leq T(n/2) + O(n) = O(n)$.

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 $z \in N(x)$ do:
      - If HamPath($G - \{x\}, z, y$), return $True$.
  - return $False$
```

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

(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: We want to find a Hamiltonian path from x to y. The back-tracking algorithm branches on the first node visited after x. The choices are all the neighbors of x in the graph. We can't return to x because the path must be simple. If we can find a Hamiltonian path from some neighbor z of x to y in $G - \{x\}$, we can append x to the front of the path and get a Hamiltonian path from x to y. On the other hand, if we cannot for any neighbor z of x, then we cannot find a Hamiltonian path from x to y, since the second node along such a path must be such a z.

The time of the algorithm is proportional to the number of recursive calls that get made, since each time through the loop makes one recursive call and then does $O(1)$ work. The recursion tree for the program has depth n, since each time step we delete exactly one node from G. In general graphs, the time could be as much as $(n-1)!$ since the first node could have $n-1$ neighbors, each possibility for the second node $n-2$ neighbors, and so on. However, if no node in G has more than 4 neighbors, an obvious upper bound for the total time is $O(4^n)$, since no node in the recursion tree has more than 4 children, and the depth is n.

In fact, since after the first step, we only call y right after one of its neighbors x has been deleted from the tree, we can see that the total size of the tree is at most $4 \cdot 3^{n-1} = O(3^n)$.