Answer all questions. Give informal (at least) proofs for all answers. Grading will be on completeness and logical correctness, as well as correctness. For the data structure question, the efficiency of your solution will also be taken into account.

**Analyzing loops - 10 pts** Consider the following iterative algorithm, that uses an \(\Theta(I)\) time procedure \(\text{proc}(I)\), which does not change \(I\).

Algorithm(n: positive integer);
1. begin;
2. \(T \leftarrow 1;\)
3. While \(T^2 \leq n\) do:
4. begin;
5. \(\text{proc}(T^2);\)
6. \(T++;\)
7. end;
8. end;

Give a time analysis, up to \(\Theta\), for this algorithm.

After the \(i\)’th iteration, the value of \(T\) is \(i\), and the loop ends when \(T \geq \sqrt{n}\). Thus, the \(i\)’th iteration, the procedure will take time \(\Theta(i^2)\), since \(i^2\) is the input; since \(i \leq \sqrt{n}\) this is always \(O(n)\). Thus the total time is \(O(\sqrt{nn}) = O(n^{3/2})\) since the number of iterations is \(\sqrt{n}\) and each iteration takes \(O(n)\) time. This is also tight, since iterations \(i = \sqrt{n}/2,...i = \sqrt{n}\) all take time at least \(i^2 \geq cn/4 \in \Omega(n)\), so we have \(\Omega(\sqrt{n})\) iterations taking \(\Omega(n)\) time.

Alternately, we can characterize the time precisely as \(\Theta(\sum_{i=1}^{\sqrt{n}} i^2) = \Theta(\sqrt{n^3}/6 - \sqrt{n^2}/3 + \sqrt{n}/2) = \Theta(\sqrt{n^3}) = \Theta(n^{3/2}).\)

**Proof of correctness - 10 pts** Below is an algorithm, that, given a sorted array \(A[1..n]\) of integers, and an integer \(T\), decides whether there is a pair \(1 \leq K < L \leq n\) with \(T = A[K] + A[L]\). After the algorithm, there is a proof of correctness, with some parts missing. Fill in the missing sections in the proof to get a complete proof of correctness.

IsSum[A[1..n]: sorted array of integers, T: integer];
1. \(\text{Found} \leftarrow \text{False};\)
2. \(I \leftarrow 1;\)
3. \(J \leftarrow n;\)
4. While \(I < J\) and NOT \(\text{Found}\) do:
5. begin; while
6. \(\text{IF } A[I] + A[J] = T \text{ THEN } \text{Found} \leftarrow \text{True};\)
7. \(\text{IF } A[I] + A[J] > T \text{ THEN } J- \text{ ELSE } I++;\)
8. end; while
9. Return \(\text{Found}.\)

Proof of correctness: (with gaps filled in) Let \(A[1..n]\) be a sorted array, i.e., \(A[1] \leq A[2]... \leq A[n]\) Let \(T\) be any integer. We need to show that if \(T = A[K] + A[L]\) for some \(1 \leq K < L \leq n\) then \(\text{IsSum}\) returns \(\text{True}\); and conversely, that if \(\text{IsSum}\) returns \(\text{True}\), then for some \(1 \leq K < L \leq n\), \(T = A[K] + A[L]\).

To see the first direction, we use the loop invariant method. Assume there are \(1 \leq K < L \leq n\) with \(T = A[K] + A[L]\). Let \(I_t\) and \(J_t\) be the values of \(I, J\) after \(t\) iterations of the while loop. The intuition is that \(K, L\) never leave the range \(I_t, J_t\) to be searched by the algorithm, i.e, that \(I_t \leq K < L \leq J_t\).
We prove this by induction. First we show that it is true at $t = 0$. At initialization, $I_0 = 1$ and $J_0 = n$. Thus, $I_0 \leq K < L \leq J_0$. So the base case is true.

For the induction step, assume the statement is true after some number of loops $t$, i.e. that $I_t \leq K < L \leq J_t$. We want to show that after the next iteration, $I_{t+1} \leq K < L \leq J_{t+1}$.

There are three cases: if $A[I_t] + A[J_t] = T$, the algorithm halts without changing $I$ or $J$, and there is nothing to prove.

If $A[I_t] + B[J_t] > T$, then in the next iteration $I_{t+1} = I_t$ and $J_{t+1} = J_t - 1$. Thus, the only way we could not have what we want to show, $I_{t+1} \leq K < L \leq J_{t+1}$, is if $L = J_t$. Now if $L = J_t$ then $T < A[I_t] + A[J_t] = A[I_t] + A[L] \leq A[K] + A[L] = T$. This is a contradiction, so we have shown by contradiction that in this case $I_{t+1} \leq K < L \leq J_{t+1}$ which is what we wanted to prove.

The final case $A[I_t] + B[J_t] < T$ is very similar. In the next iteration $I_{t+1} = I_t + 1$ and $J_{t+1} = J_t$. Thus, the only way we could not have what we want to show, $I_{t+1} \leq K < L \leq J_{t+1}$ is if $K = I_t$. Now if $K = I_t$, then $T > A[I_t] + A[J_t] = A[K] + A[J_t] \geq A[K] + A[L] = T$. This is a contradiction so we have shown by contradiction that in this case $I_{t+1} \leq K < L \leq J_{t+1}$, which is what we wanted to prove. Thus, the loop invariant still holds in all three cases.

The first direction follows from the loop invariant, since at the end of the loop, either we return True or $I_t = J_t$. If $I_t = J_t$, by the loop invariant, we must have $I_t \leq K < L \leq J_t = I_t$, which is a contradiction. Thus, if $A[K] + A[L] = T$, the algorithm returns True.

To see the converse direction, if IsSum returns True, then Found must be set to True, which can only happen in the THEN clause of line 6. Because of the IF clause of that line, we must have $A[I_t] + A[J_t] = T$ which is what we were proving, since we can pick $K = I_t$ and $L = J_t$.

Using data structures and pre-processing: Say that a graph is $d$-dense if every node has at least $d$ edges adjacent to it. The following strategy determines whether graph $G$ has a non-empty $d$-dense sub-graph $G'$.

DenseSubgraph(G: undirected graph, d: positive integer)

1. While there is a node $x$ of degree $< d$ do:
2. \hspace{1cm} $G \leftarrow G - \{x\}$.
3. If $G$ is empty return True, else return False.

Give an efficient implementation of the above strategy when $G$ is given in adjacency list format. Specify pre-processing and data structures used.

We can use the method of the IsForest example from class. We need to keep track of a set of nodes of degree $< d$ and the degrees of all nodes. Since we only need to find any element of this set, not the least degree element, we can use a stack or queue, rather than a heap.

The algorithm would then be as follows: DenseSubgraph(G: undirected graph, d: positive integer)

1. Initialize an array Degree indexed by nodes, and an empty stack $S$ of nodes.
2. For each node $x$, go through $N(x)$ and increment Degree[$x$] for every neighbor of $x$. If at the end, Degree[$x$] $< d$, Push($x$) on $S$.
3. $t \leftarrow 0$.
4. While $S \neq \emptyset$ do:
5. \hspace{1cm} $x \leftarrow Pop.S$
6. \hspace{1cm} For each $y \in N(x)$ do: Degree[$y$]--; IF Degree[$y$] = $d - 1$ THEN PushS($y$).
7. \hspace{1cm} $t + +$
8. IF $t < n$ return True ELSE return False
The second line takes time $O(n + m)$, since we run through an array of size $n$ and do two increments for each edge in the graph, one for each endpoint.

Since we only push a node $y$ on $S$ when $\text{Degree}[y]$ has either started at less than $d$, or has just become less than $d$, we only push each node once. Thus, we only pop each node at most once. In the iteration of line 4 when we pop $x$, the total work in the loop is $O(1 + \text{deg}(x))$, since we do 1 Pop and at most $\text{deg}(x)$ decrements and $\text{Push}$ operations. So the total work for all iterations is $c(\sum_x 1 + \sum_x \text{deg}(x)) = c(\sum_x 1 + \sum_x \text{deg}(x)) \leq c(n + 2m) = O(n + m)$ since the sum of degrees of all vertices in a graph is $2m$. Thus, the total time is $O(n + m)$.


2. IF $n = 0$ return False;
3. IF $n = 1$
   ELSE return False;
4. $k \leftarrow \left\lceil n/2 \right\rceil$
5. $k' \leftarrow \left\lceil (n + 1)/2 \right\rceil$
7. IF $A[k] + B[k] > T$ THEN return $\text{TargetSum}(A[1..k], B[1..k], T) \text{ OR TargetSum}(A[1..k], B[k'+n], T)$ OR $\text{TargetSum}(A[k'+n], B[1..k], T)$;
8. ELSE return $\text{TargetSum}(A[k'+n], B[1..k], T) \text{ OR TargetSum}(A[1..k], B[k'+n], T) \text{ OR TargetSum}(A[k'+n], B[k'+n], T)$;

Here, $\text{OR}$ is the Boolean OR operation on the values of the returned calls.

Example: say $n = 4$, $A[1..4] = 3, 8, 13, 21$; $B[1..4] = 1, 2, 3, 5$; $T = 13$. Then the algorithm would first compare $A[2] + B[2] = 10$ to $T$. Since it is smaller, it would recursively check $\text{TargetSum}(A[1..2], B[3..4], 13)$, $\text{TargetSum}(A[3..4], B[1..2], 13)$ and $\text{TargetSum}(A[3..4], B[3..4], 13)$. (It could ignore the case $I, J \leq 2$, since all such sums are less than $T$). After computing recursively, which I won’t show here, the sub-procedures would return $\text{True}$, $\text{False}$ and $\text{False}$. Thus, the main procedure would return $\text{True}$.

The algorithm either makes 3 recursive calls to arrays of roughly size $n/2$ in line 6, or makes a different 3 recursive calls to arrays of roughly size $n/2$ in line 7. Either way, 3 recursive calls are made. The rest of the algorithm is constant time.

Thus, $T(n) = 3T(n/2) + \Theta(1)$, so we can use theorem B.5 with $a = 3$, $b = 2$, $k = 0$. Since $3 > 2^0 = 1$, we are in the “bottom-heavy” case, and so $T(n) \in \Theta(n^{\log_2 a}) = \Theta(n^{\log_3 3})$, or approximately $\Theta(n^{1.6\ldots})$. This is better than $\Theta(n^2)$ for exhaustive search, but might still be possible to improve.