Solution 1. The main idea is as follows. $N \geq 2^b$. Therefore, $b \leq \log_2 N < n$. Guess $b = 2$ to n and for each guess, binary search for the right a.

```
Input: An integer $N \geq 1$ using $n$ bits
Output: yes iff $N = a^b$ for some $a \geq 2, b \geq 2$
for $b = 2$ to $n$ do
    $l \leftarrow 2$;
    if $N = power(l, b)$ then
        return yes
    else if $N < power(l, b)$ then
        return no
    $u \leftarrow 2^{\lceil n/b \rceil}$;
    while 1 do
        // invariant: $l^b < N < u^b$
        if $u = l+1$ then
            break;
        end
        $mid \leftarrow \lceil 0.5(l+u) \rceil$;
        $Nguess \leftarrow power(mid, b)$;
        if $Nguess = N$ then
            return yes
        else if $Nguess > N$ then
            $u \leftarrow mid$;
        else
            $l \leftarrow mid$;
        end
    end
end
return no
```

Time complexity: outer for loop runs n times. The while loop runs for at most n times. Each call to *power* takes $O(n^{1.6} \log n)$ time using Karatsuba's method. Other operations take less time. The final complexity therefore is $O(n^4)$.

□

Solution 2. The greedy rule is 'schedule lowest $\frac{t_i}{w_i}$ job first'.

Let O be an optimal ordering among the problem that minimizes the total weighted completion time. Consider any two consecutive jobs i and j in O such that $\frac{t_i}{w_i} > \frac{t_j}{w_j}$. If for every consecutive jobs i, j in O, $\frac{t_i}{w_i} \leq \frac{t_j}{w_j}$, then there is nothing to show.

Swap i and j in O to get O'. The contribution of i and j in O is:

$$A = w_i(x + t_i) + w_j(x + t_i + t_j),$$

for some $x > 0$. The contribution of j and i in O' is:

$$B = w_j(x + t_j) + w_i(x + t_j + t_i).$$

Then, $A - B = w_j t_i - w_i t_j > 0$. This is a contradiction to O's optimality.

□

Solution 3. Let L_i be the number of neighbors of v_i within the set of nodes $\{v_1, \dots, v_{i-1}\}$. Then when v_i was being added as per the above greedy rule, it contributed to at least $\frac{1}{2}L_i$ many edges to the final cut. Therefore, the algorithm's output has a cut size of at least $\frac{1}{2}\sum_{i=1}^{n} L_i$.

Now, note that $\sum_{i=1}^{n} L_i = |E|$ since each edge is counted exactly once in both sides. Therefore, the approximation ratio is $\frac{1}{2}$. □

Solution 4. We will show by induction that $p_{i+1} = \frac{1}{i+1}$ works. For the base case, $i = 1$, and we keep x_1 with probability 1. Assume by induction hypothesis, $\Pr[s_i = x_j] = \frac{1}{i}$ for every $x_j \in \{x_1, \dots, x_i\}$.

For the induction step, we'll show that $\Pr[s_{i+1} = x_j] = \frac{1}{i+1}$ for every $x_j \in \{x_1, \dots, x_{i+1}\}$. For $x_j = x_{i+1}$, this follows trivially by construction and the choice $p_{i+1} = \frac{1}{i+1}$. For any $x_j \in \{x_1, \dots, x_i\}$, by construction

$$\begin{aligned}
\Pr[s_{i+1} = x_j] &= \Pr[s_i = x_j \text{ and } s_{i+1} = s_i] \\
&= \Pr[s_i = x_j].\Pr[s_{i+1} = s_i] && \text{(due to independence)} \\
&= \frac{1}{i}(1 - p_{i+1}) && \text{(using induction hypothesis)} \\
&= \frac{1}{i+1}
\end{aligned}$$

□

Solution 5a. We'll add one more attribute called 'count' to each node n of the AVL tree that will record the total number of nodes present in the subtree rooted at node n including node n itself.

When we start with a singleton node in the AVL tree, *count* will be initialized to 1. The search operation will remain as it is. During insertion, we'll travel a root to leaf path if the key does not exist and add the new key K as a child of some leaf. After adding this node, we'll backtrack along the K to root path and while backtracking, we'll increment the count of each node. Afterwards, we need to perform some rotation, which we'll come in the end.

For AVL trees, we know that delete always happens at a leaf or at the parent of a leaf (possibly after some copying). After deleting a key K, we again backtrack along the K to root path and decrement the count values.

Now we'll come to rotations. The following figure discusses the count updates for the LL rotation and skip the other three rotations. It can be seen that after completing each rotation, count updates can be performed in $O(1)$ additional time.

For range reporting, we start by searching for the given key l. We initialize a variable called 'output' to 0. In the resulting root to some leaf (say n) path, whenever we take a left child, we add the 'count' of the right child plus 1 to *output*. Finally if the search is successful, we add 1. Then we return *output* as our answer. □

Solution 5b. The main idea here is that when interval i departs, count how many *currently active* intervals started after i started. This counting is enabled by the data structure from part a.

```
Input: An array $A$ of $2n$ numbers from $\{1,\dots,n\}$; each number appears exactly twice
Output: count of pairs of intervals that intersect but don't contain one another
seen ← an all 0 array of size $n$;
start ← an all 0 array of size $n$;
active ← an empty rage-reporting data structure from part a;
count ← 0;
for $i = 1$ to $2n$ do
    if $seen[A[i]] = 0$ then
        // $A[i]$ is arriving
        $seen[A[i]] \leftarrow 1$;
        $start[A[i]] \leftarrow i$;
        active.insert(start[A[i]]);
    else
        // $A[i]$ is departing
        $count \leftarrow count + active.range(start[A[i]] + 1)$;
        $active.delete(start[A[i]])$;
    end
end
```

□

Solution 5c. Algorithm:

1. First tag each point by their chord index to get
$$\{(1,p_1),(1,q_1),\dots,(i,p_i),(i,q_i),\dots(n,p_n),(n,q_n)\}$$
2. next sort these $2n$ tuples according to the p_i or q_i values in the following order: anti-clockwise direction on the circle starting from $(0,1)$ (we may take any other ordering as well). Then, remove the p_i,q_i values so that only the indices are left. If we follow this for the above figure, we'll get the sequence $1,2,3,2,1,3,4,4$.
3. Then apply part b and report the count. In the above example, the count will be 2 since the two intervals $2,3$ and $1,3$ intersect without containment.

□

Solution 6. Algorithm:

1. Find strongly connected components and construct the DAG of components.
2. Topologically sort the DAG to get the ordering $w_1,\dots,w_n$ where each w_i is a component
3. G is semi-connected iff w_i has an edge to w_{i+1} for every i in the DAG.

Correctness: Note that inside a strongly connected component, any pair is reachable from one another. It remains to check across two SCCs w_i and w_j. If every w_i has an edge to w_{i+1} then the answer is clearly yes. Now suppose $w_1,\dots,w_n$ is the topological order such that some w_i don't have an edge to w_{i+1}. Then the only other way w_i can have a path to w_{i+1} is either to go back to some vertex w_j for some $j < i$ or to go forward to some vertex w_j for some $j > i$, and then take some (w_j, w_{i+1}) edge. Both the possibilities are prohibited by the topological ordering.

□

ation 7. Algorithm: Suppose, your friend got a shortest distance array d of size $|V|$. Make pass over each edge (u,v) using the given weighted adjacency list and check $d(v) \le d(u) + weight(u,v)$. If all checks pass return yes. If at least one check fails, return no.

Correctness: Firstly, it is clear that if for some edge (u,v) $d(v) > d(u) + weight(u,v)$, then we have witnessed a better path to v (via u) – thus the friend's claim was wrong. Showing the converse requires more work.

Suppose all checks pass. Let the friend's tree be T and s is the source node of T. For the sake of contradiction assume there exists at least one node for which the friend's shortest path is wrong. In that case, fix a correct shortest path tree S starting from s. For each vertex $v \in V$, compare the two s to v path costs given by S and T. At least one of these pairs must be differing. Among these differing pairs, pick one that has the smallest number of hops in S. Let these two paths be $s, w_1, \ldots, w_t = u$ in S and $s, v_1, \ldots, v_k = u$ in T for some $t \ge 1, k \ge 1$. Now by our assumption, T must have the correct shortest path cost to w_{t-1} since otherwise, we would have picked the pairs of path terminating at w_{t-1} instead of u.

At this point we have the following situation. s to w_{t-1} path in T is indeed shortest but s to w_t path is not. Moreover, there is a s to w_t shortest path that goes via w_{t-1} and then takes the edge (w_{t-1}, w_t). Therefore, the check $d(w_t) \le d(w_{t-1}) + weight(w_{t-1}, w_t)$ would not have passed. This is a contradiction to the assumption that all checks pass. □