1)

```
bool flag = false;
int count = 1;
if (k == 1) {
        flag = true;
}

for (int i = 1; i < n; ++i)
{
        if (arr[i] == arr[i-1]) {
                count += 1;
        }
        else {
                count = 1;
        }

        if (count >= k) {
                flag = true;
        }
}

return flag;
```

algorithm runs in O(n) time. assumes k and n are greater than 0.

Count keeps track of the number of the same consecutive numbers you have seen.
Flag tells you if there are k consecutive equal numbers.

Invariant:
Count tells you that from your current index i back to index i - count + 1 is the same number.
If flag is false, then there aren't k consecutive equal numbers before i. If flag is true, then there have been at least k consecutive equal numbers before i.

The base case is when the flag is false and the count is 1.
We start at the second number of the sequence so there should be a count of 1 already because that number is equal to itself.
The flag should be set as false in the base case because we haven't seen k consecutive equal values unless k is 1 in which case it should be true because there is already a count of 1 (that number is equal to itself).

We have shown the base case is true. Now let’s assume the algorithm is true up to i. We need to prove i + 1.

We will examine flag and count to make sure they hold to their defined roles for i + 1. There are four cases as we loop through i.

two options for count: either the next thing we are looking at is the same as the last number or it is different
if arr[i+1] == arr[i] (the next number is equal to the last) then the count should go up by 1, which it does as seen by the statement:

```
if (arr[i] == arr[i-1]) {count += 1;}
```

if arr[i+1] != arr[i] (the next number is not equal to the last) then the count should reset back to 1 as we have seen this new number, which is the same as itself.

```
This is correctly performed by this else condition: else {count = 1;}
```

two options for flag: it is false or true.
if the flag is true, it will stay true. We have shown that count keeps track of the number of equal consecutive numbers correctly so when is count >= k, flag should become true (representing there are in fact at least k consecutive equal values).
This only occurs when count >= k and has no opportunity to change (back to false) when this happens. Code:

```
if (count >= k) {flag = true;}
```

if we have seen at least k equal consecutive numbers in the last t steps then flag will be true on the i + 1 step flag will still be true.

if the flag is false, there hasn't been k equal consecutive numbers in the t steps so far. This means count < k. But count could increase to be equal to k on the i + 1 step or remain below k.
If count increases to be equal to k on the i + 1 step in this situation, the flag changes from false to true, which follows the second part of the invariant. This means there are k consecutive equal numbers.
If count increases on the i + 1 step, but is still below k in this situation, the flag stays false. This means there isn't yet k consecutive equal numbers (according to the invariant part 1 for count, which was proven above).
If count decreases back to 1, the flag stays false. This means there isn't yet k consecutive equal numbers.

Termination:
This algorithm essentially is a for loop that runs through every number in the sequence in order. The algorithm takes n time to run. It will terminate when it hits the last number. The boolean variable flag is returned representing if k consecutive equal numbers have been observed.

This invariant being true makes the answer correct because count correctly holds the number of local consecutive equal numbers at a given index. Flag correctly becomes true if there have been at least k consecutive equal numbers observed. Otherwise, flag is rightly false because count is less than k (there haven’t been k consecutive equal numbers).

2.1)
There must be an optimal solution. Let's say its O. If it has no overlaps, then we're done. This is an optimal solution with none of the intervals overlapping except possibly at the endpoints.
If it does have overlaps, then we can push everything over so nothing overlaps. Each interval will still be one hour. By shifting your interval to the end of the overlapping one, it covers at least as many points.
So this modified optimal solution with no overlapping intervals is still optimal because it covers at least as many points using the same number of intervals.

Pushing each interval over can be done rather simply:
Let's say each interval in the optimal solution has a start time S_i and end time F_i (sorted by start time).
We run through the intervals and if the start time of S_i is not 1 hr later than S_{i-1} then we change S_i to be equal F_{i-1} and F_i to equal F_{i-1} + 1 hr. We then re-sort based on start time and run this algorithm till there are no conflicts.
This algorithm will not create any more intervals.
It will cover at least the time intervals as the original optimal solution and therefore it will cover at least as many points.

To see this last point:
If (S_{i-1}, F_{i-1}) and (S_i, F_i) are overlapping intervals in the optimal solution then $S_{i-1} < S_i < F_{i-1}$ (again points are sorted by start time).
If we make S_i now equal F_{i-1} and F_i equal to F_{i-1} + 1 hr
(S_{i-1}, F_{i-1}) is still covering its original time interval.

S_i to F_{i-1} was repeated interval coverage so any point in that interval is covered by (S_{i-1}, F_{i-1}).
$(F_{i-1}, F_{i-1} + 1 \text{ hr})$ -- the new second interval covers from the end of the last interval past where (S_i, F_i) originally covered because $F_{i-1} + 1 \text{ hr} > F_i$.
Substitution: $F_{i-1} + 1 \text{ hr} > S_i + 1 \text{ hr}$, $F_{i-1} > S_i$, which is true as stated in the assumption.
These two modified intervals cover at least the times in the original intervals and therefore they don't miss any possible points.
This process was arbitrary and can be repeated for any overlapping intervals.
This shows there is no missed points from pushing the intervals over.

2.2)
Efficient Algorithm:
There is a sorted list of times.
This algorithm works by picking the first available time and creating an interval around it with the start time being that time and the end time being 1 hr later.
Then the algorithm removes any times from the original list that are within this 1 hr window as they are already captured.
Then it repeats, picking the first available time and creating an interval around that time.

```
R <- {t1, t2, ...., tn} sorted by time
A <- {}
while (R != {}) {
        i <- choose first available in R
        A <- A U {i, i + 1 hr}
        remove from R those times between i to i + 1 hr
}
return A
```

Definition:
first available time is the earliest time in the list that is not covered by a current interval. This algorithm always starts at the first available time (meaning any interval begins at a time in the list of times).

There exists an optimal solution where my solution is at least as good.
Principle idea: the end time of the optimal solution that follows this algorithm would have the latest possible end time of all optimal solutions at every possible step size.

Let's say an optimal solution with no overlapping intervals is O with intervals: $\{O_1 = \{s(1), f(1)\}, ..., O_n = \{s(n), f(n)\}\}$
Let's say this algorithm outputs solution A with intervals: $\{A_1 = \{s(1), f(1)\}, ..., A_m = \{s(m), f(m)\}\}$

Let's prove: For all indices $r<=m$, we have $f(O_r) <= f(A_r)$.
For r=1 the statement is true because the algorithm starts by selecting the request A1 with the latest possible start time (beginning the interval at the first available time) and therefore has the latest possible end time compared to any optimal solutions.
We will assume the statement is true for $f(O_{r-1}) <= f(A_{r-1})$ and prove it for r.

proof by contradiction:
assume $f(O_r) > f(A_r)$. This means $s(O_r) > S(A_r)$ because intervals are the same length.
This can't be true because A_r starts its interval at the the first available time so O_r is missing this time and therefore this assertion is false.
Therefore: $f(O_r) <= f(A_r)$

Therefore, the rth interval this greedy algorithm selects finishes at the latest possible time.

Prove: The greedy algorithm returns an optimal set A.
Assume: If A is not optimal, then an optimal set O must have less intervals, that is, $n < m$.
Applying the above conclusion with $r = n$: $f(O_n) <= f(A_n)$ which implies $s(O_n) <= s(A_n)$
Since $m > n$ there must be a $s(A_{n+1})$ which starts after $f(A_n)$ ends and after $f(O_n)$ end.
The greedy algorithm always starts an interval on an actual time so the greedy algorithm has identified an extra time that the optimal set did not return. Therefore, this is a contradiction.
Therefore, the assumption that A is not optimal is false and therefore A is optimal: m does equal n.

3)
Algorithm:
Sort the n jobs by length (shortest to longest).
Schedule these jobs in order. Shortest job is first and longest job is last. This minimizes s_i and thus minimizes sum of $s_i + t_i$ from i = 1 to n because all t_i are there regardless.

Let's say our output is jobs A_0 to A_n
With our output: $A_i < A_{i+1}$ (in terms of time)

Now let's say we have an optimal solution O.

The two solutions are the same until t_i. There is a reversal in the optimal solution with t_i and t_j switching places.
A = ..,t_i,....,t_j,...
O = ..,t_j,....,t_i,...
$t_i < t_j$ because of how our algorithm is set up (jobs are sorted by time). The time decreases if we swap these jobs. Will show below:

We are minimizing on just s_i (which as stated above is the same as minimizing on $s_i + t_i$ because all t_i are present):
We just need to focus on this in between section as the before and after are the same.
A = ..,|t_i,t_{i+1},..,t_{j-1},t_j|,...
O = ..,|t_j,t_{i+1},..,t_{j-1},t_i|,...

Let's say the starting time sum up to t_j and t_i is w.
$S(t_j) = w$
$S(t_{i+1}) = w + t_j$
...
$S(t_{j-1}) = w + t_j + t_{i+1}$
$S(t_i) = w + t_j + t_{j-1} + t_i$

If we switch them: so the lower value is earlier
$S(t_i) = w$
$S(t_{i+1}) = w + t_i$
...
$S(t_{j-1}) = w + t_i + t_{i+1}$
$S(t_j) = w + t_i + t_{i+1} + t_{j-1}$

$t_i < t_j$ so the switched version has a lower overall sum as seen by the two middle cases.
$S(t_{i+1})$ and $S(t_{j-1})$ both have lower totals and the two ends are the same so the overall sum of start times is decreased by switching them.

You get better by continuously applying this (a switch between a higher value at a lower index with a lower value at a higher index) until no more switches are possible.

This best case with no more switches possible is the sorted list the algorithm outputs.

b)
Algorithm:
sort by smallest t_i/w_i to largest. If t_i/w_i and t_j/w_j are equal, then order the one with the higher weight first. This is the order of the jobs.

This minimizes $w_i(s_i + t_i)$ as it balances the high effect of w_i on the t_i.
Basic idea: If w_i is high these numbers should appear sooner and if t_i is low these numbers should appear sooner (t_i/w_i does this).