Homework 2 – Solutions

Solutions to Assigned Problems

Following are the problems to be handed in, 25 points each.

1. (**Divide and Conquer**, 2-page limit – your solutions should fit on two sides of 1 page).

   A Walsh-Hadamard matrix H_n is an $2^n \times 2^n$ matrix with each entry being -1 or $+1$ and $n \in \mathbb{Z}^+$, such that the (i, j)-th entry of $H_n[i, j] = \frac{1}{\sqrt{2^n}}(-1)^{i \circ j}$. Assume that the rows and columns of H_n are counted from zero, i.e., the left-topmost entry of H_n is $H_n[0, 0]$. Here $i \circ j$ is the bitwise dot-product of the binary representation of i and j represented with n bits. For example, if $i = 7$ and $j = 5$, then $\mathbf{i \circ j = (1, 1, 1) \cdot (1, 0, 1) = 2}$. Following are couple of examples of Walsh-Hadamard matrices H_2 and H_3.

$$H_2 = \frac{1}{\sqrt{2^2}}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \qquad H_3 = \frac{1}{\sqrt{2^3}}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}$$

   - Starting from the fact that $H_n[i, j] = \frac{1}{\sqrt{2^n}}(-1)^{i \circ j}$, show that

$$H_n = \frac{1}{\sqrt{2}}\begin{bmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{bmatrix}$$

   - Show that the Euclidean norm of every column and every row is 1. (You are allowed to search Wikipedia for the definition of Euclidean norm.)
   - Using the above properties, write and prove an induction claim that shows that the columns of H_n form an orthonormal basis, i.e., the dot-product of any two columns of H_n equals to zero, and every column of H_n has Euclidean norm of one. *Hint: You will need one of the properties of orthonormal matrices to show this.*
   - Consider a vector $v \in \mathbb{R}^{2^n}$, i.e., a vector with 2^n entries with real numbers. Design an algorithm to compute $H_n \cdot v$ in time $O(n \log n)$. Prove the runtime bound for your algorithm.

- We start this proof by fixing an arbitrary value for $n \geq 1$ and we assume that H_{n-1} was built using the bit-wise dot product element representation $H_{n-1}[i,j] = \frac{1}{\sqrt{2^{n-1}}}(-1)^{i \circ j}$, where $i, j \in [0, 2^{n-1} - 1]$. When we move to the $H_n \in \mathbb{R}^{2^n \times 2^n}$. The new column and row indices, i' and j', need one more bit in order to be represented, effectively moving from a $n-1$-bit representation for i, j to an n-bit representation for i', j'. We can call these two new bits $b_{i'}, b_{j'}$ and we can then rewrite i', j' as

  $$i' = (b_{i'}, \underbrace{\ldots, \ldots, \ldots}_{n-1 \text{ values}}) \qquad j' = (b_{j'}, \underbrace{\ldots, \ldots, \ldots}_{n-1 \text{ values}})$$

  Since the elements of H_n are determined by a bit-wise dot product of i' and j', part of this product will be identical to the one for H_{n-1}. In other words, the bit-wise dot product for the last $n-1$ values in the representation of i' and j' is the same as $i \circ j$. The difference comes from the four possible combinations of $b_{i'}, b_{j'}$

  - $b_{i'} = 0, b_{j'} = 0$ This is for the upper left quadrant, and the bitwise dot product in this case is not changed, i.e. $i' \circ j' = i \circ j + 0 \times 0 = i \circ j$. In other words, the upper left quadrant is $\frac{1}{\sqrt{2}} H_{n-1}$(we still have the constant multiplier for each matrix element).
  - $b_{i'} = 1, b_{j'} = 0 \rightarrow i' \circ j' = i \circ j + 0 \times 1 = i \circ j. \rightarrow \frac{1}{\sqrt{2}} H_{n-1}$.
  - $b_{i'} = 0, b_{j'} = 1 \rightarrow i' \circ j' = i \circ j + 1 \times 0 = i \circ j. \rightarrow \frac{1}{\sqrt{2}} H_{n-1}$.
  - $b_{i'} = 1, b_{j'} = 1 \rightarrow i' \circ j' = i \circ j + 1 \times 1 = i \circ j + 1. \rightarrow -\frac{1}{\sqrt{2}} H_{n-1}$.

  Putting these together, we see that

  $$H_n = \frac{1}{\sqrt{2}} \begin{bmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{bmatrix}$$

- Proving that the columns are normalized is very straightforward. We fix a column k and do the summation

  $$\sqrt{\sum_i H[i,j]^2} = \sqrt{\sum_{i=0}^{2^n - 1} \frac{1}{2^n}} = \sqrt{1} = 1$$

- We note that if a square matrix M has the property that

  $$MM^T = M^T M = \mathbb{I}$$

  that matrix is orthonormal. We show that all Hadamard matrices are orthonormal. The proof starts by noting that $H_n = H_n^T$(this follows from the definition), so all we need to prove is that $H_n H_n = \mathbb{I}$.

  Claim Every Hadamard matrix H_n of has the property that $H_n H_n = \mathbb{I}$.

  Base case The base case for is that of $n = 1$. This follows trivially since

  $$H_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

  and through simple multiplication one can see that

  $$H_1 H_1^T = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = \mathbb{I}$$

Induction hypothesis We assume that for a given k, H_k satisfies $H_k H_k = \mathbb{I}$

Induction step We already know that H_{k+1} can be obtained as

$$H_{k+1} = \frac{1}{\sqrt{2}} \begin{bmatrix} H_k & H_k \\ H_k & -H_k \end{bmatrix}$$

Therefore, we can see that

$$\begin{aligned} H_{k+1}H_{k+1} &= \frac{1}{2} \begin{bmatrix} H_k & H_k \\ H_k & -H_k \end{bmatrix} \begin{bmatrix} H_k & H_k \\ H_k & -H_k \end{bmatrix} \\ &= \frac{1}{2} \begin{bmatrix} H_kH_k + H_kH_k & H_kH_k - H_kH_k \\ H_kH_k - H_kH_k & H_kH_k + H_kH_k \end{bmatrix} = \mathbb{I} \end{aligned}$$

- Let v be an arbitrary column vector of size $n = 2^k$. Also let v_u and v_l be two column vectors of size $n/2 = 2^{k-1}$ representing the upper and lower halves of v i.e. $v = \begin{bmatrix} v_u \\ v_l \end{bmatrix}$. Therefore, we have:

  $$H_k \cdot v = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} \begin{bmatrix} v_u \\ v_l \end{bmatrix} = \begin{bmatrix} H_{k-1} \cdot v_u + H_{k-1} \cdot v_l \\ H_{k-1} \cdot v_u - H_{k-1} \cdot v_l \end{bmatrix}$$

  Thus in order to find $H_k \cdot v$ it suffices to find $H_{k-1} \cdot v_u$ and $H_{k-1} \cdot v_l$ and then compute $H_{k-1} \cdot v_u + H_{k-1} \cdot v_l$ and $H_{k-1} \cdot v_u - H_{k-1} \cdot v_l$. See HadamardMult$(k, v)$.

Algorithm 1: HadamardMult(k, v)

```
1 if $k = 0$ then
2     Return $v$ – { $v$ is a scalar in this case } ;
3 $v_l \leftarrow$ Lower$(v)$;
4 $v_u \leftarrow$ Upper$(v)$;
5 $m_l \leftarrow$ HadamardMult$(k-1, v_l)$;
6 $m_u \leftarrow$ HadamardMult$(k-1, v_u)$;
7 $m \leftarrow \begin{bmatrix} m_u + m_l \\ m_u - m_l \end{bmatrix}$ – { Takes $O(n)$ time for elementwise vector addition and subtraction};
8 Return $m$ ;
```

If $T(n)$ is the cost for finding $H_k \cdot v$, then we can find each $H_{k-1} \cdot v_u$ and $H_{k-1} \cdot v_l$ in $T(n/2)$. In addition, $H_{k-1} \cdot v_u + H_{k-1} \cdot v_l$ and $H_{k-1} \cdot v_u - H_{k-1} \cdot v_l$ can be found in linear time since we need to do elementwise vector addition and subtraction. Therefore:

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

Observe that $f(n) = O(n)$ and $1 = \log_2 2$. Thus using Master Theorem, we obtain $T(n) = \Theta(n \log n)$.

2. (**Divide and Conquer**, 2-page limit – your solutions should fit on two sides of 1 page).

   You and your sister are travelling on a bus when you recall the "Hot and cold" game you used to play as kids. To kill time, you decide to play a version of it with numbers. One of you thinks of a number between 1 and n, and the other tries to guess the number. If you are the one guessing, every time you make a guess your sister tells you if you are "warmer", which is closer to the number in her head, or "colder", which is further away from the number in her head. Using this information you have to come up with an algorithm which helps you guess the number quickly. You can use a command called Guess(x), where x is your guess, which returns "warmer", "colder" or "you guessed it!". You are required to give an English explanation for your algorithm. Also, prove the correctness of your algorithm and give an analysis of the space and time complexity. An ideal solution will propose an algorithm that takes $\log_2(n) + O(1)$ guesses in the worst case scenario. Recalling how binary search works might be helpful.

Algorithm:

```
Algorithm 2: Hot-or-Cold
Input: Integer $n$ and access to function "guess"
$answer \leftarrow Guess(\lfloor n/3 \rfloor)$;
if $answer = correct$ then
    Return $\lfloor n/3 \rfloor$;
else
    $\ell \leftarrow 1$, $u \leftarrow n$; $prev \leftarrow 1$;
    while $u - \ell > 0$ do
        $mid \leftarrow \lfloor \frac{\ell+u}{2} \rfloor + \frac{1}{2}$;
        $new \leftarrow 2mid - prev$;
        $answer \leftarrow Guess(new)$;
        if $answer = correct$ then
            Return $new$;
        if $(answer = warmer\ and\ prev < mid)\ or\ (answer = colder\ and\ prev > mid)$ then
            $\ell \leftarrow mid + \frac{1}{2}$;
        else
            $u \leftarrow mid - \frac{1}{2}$;
        $prev \leftarrow new$;
    Return $u$;
```

As with binary search, the idea is to maintain an interval $[\ell, u]$ that contains the number.
No matter what my previous guess was, I can always find a new guess which will allow me to halve the length of the interval.
Let l to u be your current interval and $prev$ be your last guess; using this you calculate a new interval for your next guess.
You first calculate $mid = (\ell + u)/2$ of your current interval and use that to decide the new guess, $new = 2mid - prev$.
We can verify that this indeed cuts the interval in two, since the distance between mid and $prev$ is the same as the distance between mid and new.
Specifically, we have $mid - prev = new - mid$.

Now, our new interval will be contained in either $[\ell, mid]$ or $[mid, u]$ and we repeat the steps above with the new interval and new $prev$ value.

The case where $mid = prev$, since $next = prev$ the query won't give us any new information.
Also as we have only two outputs, *warmer* and *colder*, besides *correct* answer, we need to handle the case when the number is exactly half-way between $prev$ and new.

In the algorithm, we get around this by forcing mid to not be an integer and hence, at each iteration assigning $mid \leftarrow \lfloor \frac{\ell+u}{2} \rfloor + \frac{1}{2}$.

This ensures that we make progress on every iteration.

Analysis We have to show that on each iteration of the algorithm the interval ℓ, u contains the number, and that this interval shrinks sufficiently on each iteration. The following useful invariant simplifies our analysis:

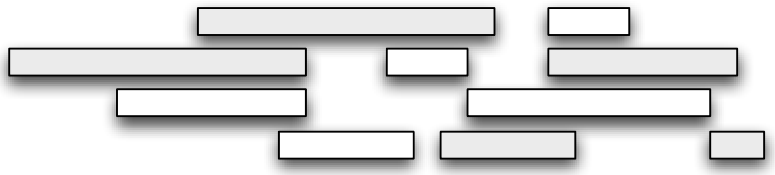
Loop invariant: *At the beginning of each iteration of the while loop, we have: (a) the number is contained in* $\{\ell, ..., u\}$ *and (b) the number* $prev$ *is an integer.*

Proof: It's easy to check that this holds the first time through. Let's prove by induction that it holds on every subsequent step. Let med_- and med_+ be the integers immediately below and above mid. Since $u > \ell$, these integers are always in the interval $\{\ell, ..., u\}$. If $prev < mid$, then the integers $\{\ell, ..., mid_-\}$ are closer to $prev$ than new, and the integers $\{mid_+, ..., u\}$ are farther away. If $prev > mid$, then the integers $\{\ell, ..., mid_-\}$ are farther from $prev$ than new, and the integers $\{mid_+, ..., u\}$ are closer. Either way, our algorithm correctly reassigns the bounds of the interval so that it still contains the number. Moreover, new is an integer since $2mid$ and $prev$ are integers; hence, $prev$ will again be an integer on the next pass through the loop.

Time analysis: At each iteration, we reduce the length of the interval fro $u - \ell$ to (at most) $\lceil (u - \ell)/2 \rceil$. Thus, the algorithm terminates after at most $\log_2(n) + 2$ iterations, and the total number of guess is at most $\log_2(n) + 3$.

Correctness: The algorithm is correct since the number is always contained in $\{\ell, ..., u\}$. The algorithm terminates when either the niece tells us we've guessed correctly or the interval has been reduced to a single integer (in which case we're certain of the answer). Either way, we guess the number.

3. (**Greedy Algorithm**, 2-page limit – your solutions should fit on two sides of 1 page). The Menlo Park Surgical Hospital admitted a patient, Mr. Banks, last night who was in a car accident and is still in critical condition and needs monitoring at all time. At any given time only one nurse needs to be on call for the patient though. For this we have availability slots of all the nurses, which is a time from which they are available, a_i, to the time they have other engagements, b_i. You need to devise an algorithm that makes sure you can cover Mr. Banks' entire stay while having minimum number of nurses be on call for him. A nurse leaving at the same time as another arrives is acceptable. Following is an example of how the availability slots for the nurses will look like. The darker bars correspond to a set of 5 nurses who can cover the entire duration. (Notice, though, that 4 nurses would have sufficed.)



Your efficient **greedy** algorithm should take input a list of pairs of times (a_i, b_i) for $i = 1$ to n

(a) Consider the greedy algorithm that selects nurses by repeatedly choosing the nurse who will be there for the longest time among the periods not covered by previously selected nurses. Give an example showing that this algorithm does **not** always find the smallest set of nurses.

(b) Present an algorithms that outputs a smallest subset of nurses that can cover the entire duration of Mr. Banks' stay at the hospital or say that no such subset exists.

(c) Prove that your algorithm is correct.

(d) State its running time.

- Suppose the duration of Mr. Banks' stay is from times 0 to 1, and you have three possible nurses, with availability

$$\left[0, \frac{1}{2}\right], \left[\frac{1}{2}, 1\right], \left[\frac{1}{10}, \frac{9}{10},\right]$$

  The greedy algorithm will use all three nurses (since it will select the third nurse first). However, the optimal solution uses on the first two nurses.

- In this case, the greedy criterion is that we greedily choose the nurse who is already available at the time the previously on call nurse has to leave and is also the last nurse to depart out of this list. If this isn't immediately clear, take a second and let the criterion sink in before moving on.

```
Sort nurses in the ascending order of availability start time (so $a_1 \leq a_2 \leq \cdots \leq a_n$) ;
$S \leftarrow \emptyset$;                                   /* Set of selected nurses */
$t \leftarrow$ start_of_stay;                  /* time that last selected nurse will leave */
$\theta \leftarrow 0$;
$i \leftarrow 1$ ;                                 /* nurse currently under consideration */
while $t <$ end_of_stay do
    /* max stores the availability start of the previously added nurse */
    $max \leftarrow t$;
    /* while loop looks for next greedy choice */
    while ($a_i \leq t$ and $i \leq n$) do
        /* Update $\theta$ if $i$ can stay later than $\theta$ */
        if ($b_i > max$) then
            $\theta \leftarrow i$ ;
            $max \leftarrow b_i$ ;
        $i \leftarrow i + 1$ ;
    if $max > t$ then
        Add $\theta$ to $S$;
    else
        /* If max = $t$, then no nurse was found who could stay later than t. */
        Return "Error: no valid solution exists";
Return $S$;
```

- There are two statements to prove: first, that the algorithm always returns a valid solution (if one exists) and second, that it finds the optimal solution.
  - To argue **feasibility**, note that max is always set to the time at which the nurse is supposed to leave for the last nurse added to the set (or the start of the stay of the patient, in the first iteration). The condition on the inner while loop ensures that we only consider nurses i such that $a_i \leq t$. Thus, we only add a nurse θ when all times between the start of stay and b_θ have been covered. Moreover, we consider *all* nurses who are available before time t since A is sorted in increasing order. Thus, if any nurse can cover a given time slot, we will find one.
  - For **optimality**, we use a "greedy stays ahead" argument. Let's call our strategy $S = s_1, \cdots, s_k$ and consider any other strategy $T = t_1, \cdots, t_k$.

    Claim 1. *For every $k \geq 1$, the first k nurses in S (sorted by start time of availability) cover the largest interval that can be covered by k nurses.*

To see why, consider any other valid strategy T that differs with S somewhere in the first k nurses. Sort both S and T in order of ascending start time and let $j \leq k$ be the first nurse in which S and T differ. Both s_j and t_j must start before $b_{s_{j-1}} = b_{t_{j-1}}$ (otherwise there's a hole in the schedule). Since our algorithm selects the nurse who can stay latest at each stage, we also know that $b_{s_j} \geq b_{t_j}$. So we can replace t_j with s_j in T without affecting the amount covered by the first k nurses of T. However, this contradicts the minimality of j.

- $\Theta(n \log n)$, dominated by sorting. So why the main loop takes $O(n)$ time, note that only a constant number of operations are done for each increment of the variable i, which increases from 1 to at most $n + 1$.

4. (**Greedy Algorithm**, 2-page limit – your solutions should fit on two sides of 1 page). Picture, if you will, a long river with some towns scattered sparely along it. You are in charge of building a series of small hydro-electric power plants for these towns to give them some renewable electricity; however, the power plants can't power a town further than 20 miles away, due to their particular design - they can, however, give power to as many towns as they can reach. You want to build them along the river such that every town is within 20 miles of at least one of the plants.

   Give an efficient algorithm that achieves this goal using the minimum number of power plants. Prove using *greedy stays ahead strategy.*

Algorithm:

Starting at the source of the river and going downstream, place the first power station 20 miles downriver from the first town along the way. Then, find the next town that isn't covered by that power station, and place the next power station 20 miles downriver from it. Repeat this process until all towns are covered.

Correctness. First observe that all towns are covered because we go through them one by one, and when we find a town that is not covered by the previous power plant, we add a new power plant that covers that town. It remains to prove that the algorithm outputs an optimal solution. Let $B_1, B_2, \cdots, B_t$ be the power plant locations produced by the algorithm, and $O_1, O_2, \cdots, O_m$ be an optimal set of power plant locations, sorted in increasing order.

Claim 2. *$O_i \leq B_i$ for all $0 \leq i \leq m$.*

Proof. (By induction on i.)

Base case: $i = 1$. Since the first town should be covered, $O_1 \leq B_1$.

Induction step: Assume the claim holds for $i = k$. Suppose, for contradiction, that $O_{k+1} > B_{k+1}$. Since our algorithm placed a power plant at B_{k+1}, there must be a town at $x = B_{k+1} - 20$.

$$\begin{aligned} O_{k+1} - x &> B_{k+1} - x = 20, && \text{since } O_{k+1} > B_{k+1}. \\ x - O_k &\geq x - B_k > 20, && \text{since } O_k \leq B_k \text{, and } x \text{ could not be covered by } B_k. \end{aligned}$$

Therefore, x can be covered by neither O_k nor O_{k+1}. Contradiction. Thus, $O_i \leq B_i$ for all i. □

Claim 3. *Our algorithm outputs the optimal number of power plants: $t \leq m$.*

Proof. Suppose, for contradiction, $t > m$. Thus, there is some town at location x, where $x - B_m > 20$. By the claim above, we get $O_m \leq B_m$. So $x - O_m \geq x - B_m > 20$, which implies that some town is not covered by the optimal placement. Contradiction. □

Time and space complexity. This algorithm runs in time $O(n)$ and uses space $O(n)$.

5. [*] (**Optional, no collaboration**, 2-page limit – your solutions should fit on two sides of 1 page).

You and your roommate are in a war over the thermostat - they like it cold, and you like it warmer. The temperatures range from 1 to n degrees. Your roommate is willing to leave the thermostat at some (unknown to you) temperature t or any lower temperature. You sit down with your roommate and agree to negotiate in the following way: In each round, you can name any temperture s between 1 and n. If $s > t$, they will say "too warm". Otherwise, they'll agree to s. Your goal is to ensure that the apartment is at the maximum acceptable temperture t.

One way to ensure that you will have the temperture at s is to name all integers, starting from n and going down by 1 in each round, until they agree to the temperture t. But, if you follow this strategy, you might have to go through n rounds (irratating everyone involved).

If you are allowed to change your mind (that is, ask for a higher temperature) after your roommate accepted your offer, you might want to try binary search as your strategy. However, it would also be annoying to change your mind so many times.

(a) Suppose you are allowed to change your mind exactly once. Describe a strategy for ensuring a fair temperture, t, that uses $o(n)$ rounds of negotiation (as few as you can).

(b) Now suppose you are allowed to change your mind k times, where $k > 1$. Describe a strategy for ensuring a fair temperture, s, with as few negations as you can. Let $f_k(n)$ denote the number of rounds you use, as a function of n. (The answer from part (a) is your $f_1(n)$.) For each k, you should be able to get an asymptotically better solution than for $k-1$: that is, make sure that $f_k(n) = o(f_{k-1}(n))$.

(a) Suppose for simplicity that $\sqrt{n}$ is an integer. You can start from n and go down by $\sqrt{n}$ in each subsequent round of negotiations, until your roommate accepts your proposed temperature $g = t\sqrt{n}$ for some integer t. (If all offers are rejected, set t to 0). Now, you know that s is between $t\sqrt{n}$ and $(t+1)\sqrt{n} - 1$. So, you can ask to renegotiate once and figure out s by requesting consecutive integers from $(t+1)\sqrt{n} - 1$ down until your offer is accepted.

That way, you need at most $\sqrt{n}$ rounds before you change your mind and at most $\sqrt{n}$ rounds after, for a total of $\Theta(\sqrt{n}) = o(n)$ rounds.

If $\sqrt{n}$ is not an integer, you can go down in steps of $\lfloor \sqrt{n} \rfloor$, which still gives a bound of at most $3\sqrt{n}$ on the number of rounds: at most $2\sqrt{n}$ before and at most $\sqrt{n}$ after you change your mind.

(b) We will show by induction that $f_k(n) \le 3kn^{\frac{1}{k+1}}$. You can start going down from n in steps of $\lfloor n^{\frac{k}{k+1}} \rfloor$. This way, you will go through at most $2n/n^{\frac{k}{k+1}} = 2n^{\frac{1}{k+1}}$ rounds of negotiation before your roommate accepts (or you reach 0). This allows you to narrow down the interval containing s to length at most $n^{\frac{k}{k+1}}$.

Then you can tell your roommate you changed your mind and apply the strategy for $k-1$ mind changes recursively. By induction, it uses at most $3(k-1)(n^{\frac{k}{k+1}})^{\frac{1}{k}} = 3(k-1)n^{\frac{1}{k+1}}$ rounds. Adding in at most $2n^{\frac{1}{k+1}}$ rounds made before the first change of mind, we get a bound of $3kn^{\frac{1}{k+1}}$, completing the inductive step.