CSE 417 HW3

Zachary McNulty (zjmcnulty, 1636402)

January 2019

1 Problem 1

Q: Prof. Flashbulb proposed the following greedy algorithm for solving the interval partitioning problem (aka interval coloring, aka scheduling all intervals; section 4.1, p 122): apply the greedy algorithm for interval scheduling from the earlier part of section 4.1 (the algorithm on page 118); give all of those intervals one color, remove them from consideration, and recursively apply the same method to the resulting reduced set of intervals (using new colors). Show that this method does not give an optimal solution to the problem by giving a counterexample. (Keep it simple; 4 to 6 intervals should suffice.) Explain why your counter example is a counter example.

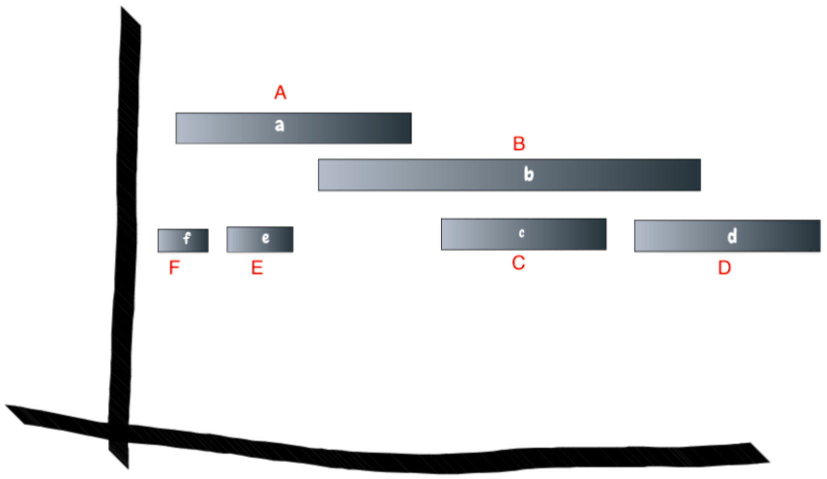


Figure 1: Counterexample

Here, we see that the algorithm will generate the following interval partition:

- The first round of interval scheduling will clearly generate the schedule F, E, C, D as this is clearly the optimal single schedule.
- Place these intervals in one part and repeat the algorithm without these intervals.
- The second and third rounds of interval scheduling will yield A and B in an arbitrary order, placing them both in their own parts, as the two overlap.
- This yields a partition with 3 parts: $\{F, E, C, D\}\{A\}\{B\}$.

However, the partition $\{A, C, D\}\{F, E, B\}$ requires only two parts, and thus the interval partition generated by the algorithm is not optimal. Therefore, the algorithm does not guarantee an optimal solution always, as shown by this counterexample: we could have generated a partition with only 2 parts, but the algorithm generated one with three parts

2 Problem 2

Suppose you are designing a new Computer Science building. It has one very long corridor, along which all faculty offices are located, say at positions $x_1 < x_2 < \ldots < x_n$ (arbitrary real numbers, for simplicity). Of course, faculty will die if their offices are not located within 100.0 feet of a combined espresso cart/WiFi hotspot.

Q: Briefly describe a simple greedy algorithm which, given the xi , will find cart/hotspot locations $g_1 < g_2 < \ldots < g_m$ satisfying the above constraint while minimizing the total number m of carts/hotspots. Prove your algorithm is correct.

Algorithm

We will choose where to place the carts/hotspots starting from the beginning of the hall, placing them as far as possible from the nearest office down the hall, x_i with the lowest index, still not within 100 ft of one of these stands. For example:

Choose g_1 to be the spot $100ft$ down the hall from the first office (i.e. $x_1 + 100$). Now, suppose that offices $x_1, x_2, \ldots, x_k$ are within 100 ft of g_1. Then, choose $g_2 = x_{k+1} + 100$. Now suppose offices $x_1, x_2, ..., x_k, ...x_n$ are within 100ft of either g_1 or g_2. Choose $g_3 = x_{n+1} + 100$. Repeat this until all offices are within 100 feet of a hotspot/cart.

Proof of correctness by Induction

Proof: Let $g_1 < g_2 < \ldots < g_m$ be the locations chosen by our algorithm and $j_1 < j_2 < \ldots < j_k$ be another valid choice of locations for the hotspots/carts.

Base case: Note that if $j_1 > g_1 = x_1 + 100$, then the distance from j_1 to x_1 is over 100ft, and thus solution j would not be a valid set of locations, a contradiction. Thus, $j_1 \leq g_1$.

Induction: Suppose $j_i \leq g_i$. Suppose the hotspots/carts $g_1, ..., g_i$ cover all the offices up until office x_j (no offices in between will be missed as the algorithm is constructed to cover all offices). Since g_i cannot cover office x_{j+1}, j_i certainly cannot cover it either as $j_i \leq g_i$. This implies that the next hotspot/cart, j_{i+1}, must cover office x_{j+1}. As the algorithm chooses $g_{i+1} = x_{j+1} + 100$, $j_{i+1} \leq g_{i+1}$ otherwise j_{i+1} would be too far from office x_{j+1} to cover it.

Suppose our greedy approach requires more hotspots/carts, i.e. $m > k$. As shown above, $j_k \leq g_k$. Since $m > k$, g_{k+1} exists, which means there is an office x_j such that $x_j > g_k + 100$. But this implies that the solution j does not cover all offices. Thus, $m \leq k$ for any solutions j, and our algorithm produces the optimal solution. **Q.E.D.**

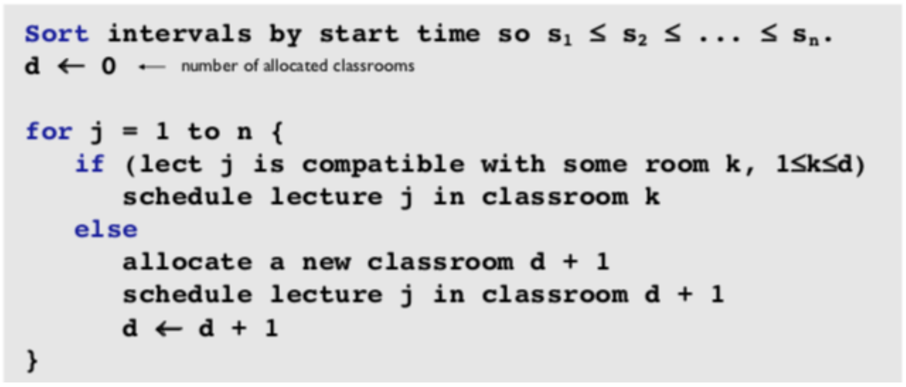
Proof of correctness by Exchange Argument

Consider an arbitrary solution placing hotspots/carts at locations $y_1, y_2, ..., y_m$ for office locations $x_1, ..., x_n$. Suppose $y_1 < x_1 + 100$. Then, as no offices precede x_1, we can move y_1 to $x_1 + 100$ without losing service to any of the offices y_1 previously covered. In fact, now y_1 covers a farther distance past the beginning of the hall than before. Therefore, y_1 may now cover more offices than before, and certainly cannot cover less. If it covers an additional office(s) x_j, then y_2 no longer needs to cover x_j and can move farther down the corridor. In general, suppose x_k is the earliest office that y_i covers. Then, $y_i \leq x_k + 100$. Moving $y_i = x_k + 100$ then only expands the reach of y_i down the hallway, and cannot increase the number of hotspots required. Repeating this for every hotspot, we can generate a placement that resembles our greedy algorithm. Thus, even an optimal arrangement can be shifted into our greedy arrangement without increasing the number of hotspots, implying our greedy arrangement is optimal.

3 Problem 3

A quick look at the algorithm on page 124 for the interval partitioning problem suggests an $O(n^2)$ time bound: an outer loop for $j = 1, ..., n$, and an implicit inner loop "for each interval I_i" that also seems to need order n time.

Qa: Give a slightly more detailed description of an implementation of the following algorithm that will take only $O(dn)$ time.



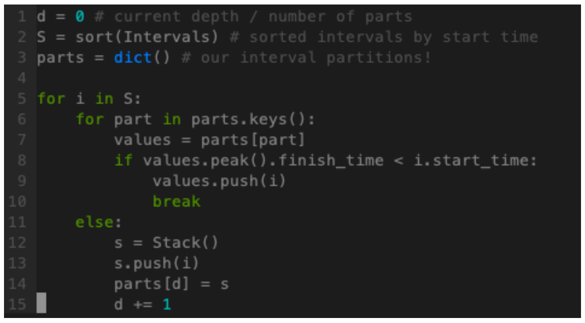
```
Sort intervals by start time so s₁ ≤ s₂ ≤ ... ≤ sₙ.
d ← 0  ← number of allocated classrooms

for j = 1 to n {
   if (lect j is compatible with some room k, 1≤k≤d)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

Figure 2: Lecture 5 slide 29 (pg 6)

Algorithm

Construct a dictionary/map which stores the part number as a key and a stack of intervals as the value. Generate a global variable which keeps track of the current depth. Sort the intervals by start time ascending. For each interval I_i, loop over the keys of the dictionary (the part numbers) and check the finish time of the interval I_s at the top of each stack. If the finish time of I_s is before the start of I_i, added I_i to that stack (i.e. I_i is compatible with the part I_s is in). Else, if I_i is not compatible with any current parts, add it to its own part. Repeat this process until all intervals are placed in parts.



```
d = 0 # current depth / number of parts
S = sort(Intervals) # sorted intervals by start time
parts = dict() # our interval partitions!

for i in S:
    for part in parts.keys():
        values = parts[part]
        if values.peek().finish_time < i.start_time:
            values.push(i)
            break
    else:
        s = Stack()
        s.push(i)
        parts[d] = s
        d += 1
```

Complexity

Ignoring the $O(nlogn)$ time required to sort the intervals based on start time, this algorithm is $O(dn)$. The outer loop runs n times, once for each interval, and for each of those executions we loop over each of the current parts (up to d parts). Within this inner loop, all the operations are O(1): peeking at the top of the stack, pushing on to the top of the stack, accessing key-value pairs from the dictionary, and doing simple logical comparisons. Thus, overall the algorithm runs in $O(dn)$ time.

Qb : If d is small, the above method typically will be much faster than the simple $O(n^2)$ analysis suggests. Show, however, that it is not faster in the worst case.

Note that $dn \in O(n^2)$ if $d \in O(n)$. Consider the graph G representing the interval partitioning problem : a vertex for each interval and vertices i, j are connected by an edge **iff** intervals I_i, I_j conflict/overlap in time. If our system has at least depth d, this implies there exists at least d intervals that all overlap with each other, implying that G has a complete subgraph on d vertices K_d. In a graph on n vertices, there are $\binom{n}{2}$ possible edges and $\binom{d}{2}$ of these are part of K_d, leaving

$$\binom{n}{2} - \binom{d}{2} = \frac{n(n-1) - d(d-1)}{2}$$

edges that are not. This yields

we have two choices for each of these edges: include or exclude it from the graph $\longrightarrow$ $2^{\frac{n(n-1)-d(d-1)}{2}}$

possible graphs with this **specific** K_d subgraph. If $d = \frac{n}{2}$, then $d \in O(n)$ and we see there are at least:

$$2^{\frac{n(n-1)-n/2(n/2-1)}{2}} = 2^{\frac{4n(n-1)-n(n-2)}{8}} = 2^{\frac{3n^2-2n}{8}}$$

graphs of depth $d = \frac{n}{2}$ or more, all of which thus then run $O(n^2)$ with this algorithm. In actuality, there are many more as we required there be a specific K_d present. Regardless, as $n \to \infty$, we can see there are infinitely many such graphs. Thus, there are infinitely many such cases where this algorithm runs in $O(n^2)$ worst case time. **Q.E.D.**

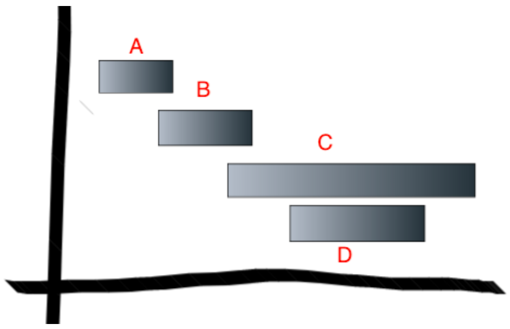
Qc: Give a more clever implementation of the algorithm whose running time is $O(nlogn)$, (even if depth is $\Omega(logn)$) and justify this time bound.

Instead of storing the parts (stacks) in a dictionary individually, store them all together in a Priority Queue (sorted based on the finish time of their top elements, with earlier finish times coming first). For each interval I_i, peek at the element at the front of the Priority Queue. If the element last added to this part (top of the stack) has a finish time earlier than the start time of I_i, remove the part from the queue, push I_i onto the part's stack, and add the modified stack back to the queue. Else, since the queue is sorted in ascending order by finish time, I_i's start time cannot precede any of the finish times of the other parts as well, so it must go into its own part. Create a new stack, add I_i to it, and put it into the priority queue. If this priority queue is implemented using Min Heaps, adding/removing elements is $O(logn)$ and the other operations like peeking are $O(1)$. Since we perform an add/remove for each of the n intervals, overall this yields $O(nlogn)$.

4 Problem 4

The interval partition algorithm sorts intervals by start time. At first glance, it seems possible that sorting by end time would work just as well. Indeed, most of the statements made in the correctness proof (approximately slide 30 in the greedy slides) remain correct when applied to this modified algorithm (after changing "start time" to "end time" and some other obvious things in a few places).

Qa: Give a counterexample to the correctness of the modified algorithm and explain why it is a counterexample



According to the end-time algorithm, the intervals will be placed in the order $ABDC$. A will be placed in its own component, and as B overlaps A so will B. D can be placed in parts with either A or B, it depends on how the parts are iterated over and is thus an implementation detail (not part of the algorithm). Suppose it is placed with A. Then, as C overlaps both parts, it must be put in its own part. Overall the partition is $\{A, D\}, \{B\}, \{C\}$. However, the partition $\{A, C\}, \{B, D\}$ contains fewer parts, and thus the algorithm generates a sub-optimal solution. While it is possible that the algorithm finds an optimal solution with these intervals, this relies on specifics of the implementation of the algorithm (i.e. the order parts are traversed in) which is not an aspect of the algorithm. Thus, the algorithm itself is not valid.

Qb: Explain where the proof goes wrong and why

"Proof":

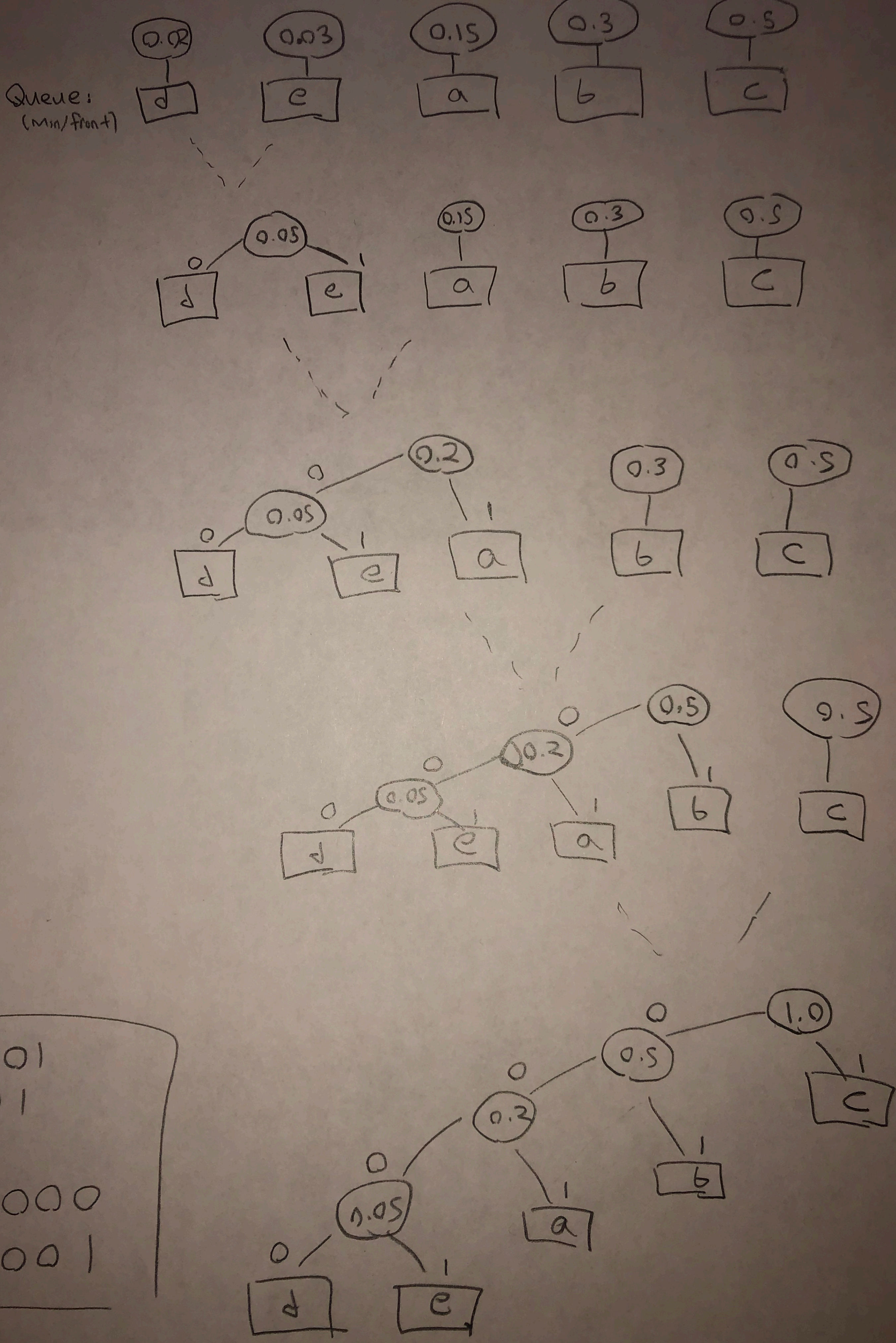
1. Let d = number of rooms the greedy algorithm allocates.
2. Classroom d is opened because we needed to schedule a job, say j, that is incompatible with all $d - 1$ previously used classrooms.
3. Since we sorted by end time, all these incompatibilities are caused by lectures that end no earlier than e_j.

4. Thus, d lectures overlap at time $e_j - \epsilon$, i.e. $depth \geq d$

5. all schedules use $\geq$ depth rooms, so $d = depth$ and greedy is optimal

The proof breaks down at step 4. We know that there $d-1$ jobs that overlap with job j and that they all end before time e_j, but that does not imply each one of these overlap pairwise. The problem is the restriction on the end-time does not force the overlap to occur in any specific region of job j. For example, suppose you claim there are $d-1$ jobs overlapping at point k somewhere in the time interval for job j I_j. For any such k, I could replace one of the $d-1$ jobs with a job on the interval $[start_j, k-\epsilon]$ and it would no longer be true that $d-1$ jobs overlap at time k. Thus, the algorithm would have failed to produce an optimal solution on this modified set of jobs. Therefore, the algorithm cannot be valid.

5 (a)



a: 001

b: 01

c: 1

d: 0000

e: 0001

5 Problem 5

Q5b: What will be the total length in bits of the compressed text built from a sequence of 100 letters having the above frequencies using the Huffman code you calculated?

The total number of bits in the compressed text will be:

$$\begin{aligned} total &= 15(bits_a) + 30(bits_b) + 50(bits_c) + 2(bits_d) + 3(bits_e) \\ &= 15(3) + 30(2) + 50(1) + 2(4) + 3(4) \\ &= 175 \end{aligned}$$

Q5c: Suppose I tell you that the frequencies of a and b are .15, .30, resp, as above, but I don't tell you the other frequencies. I can quickly tell (e.g., without running the full Huffman algorithm) that the Huffman code for this data is not the one shown in figure 4.16 (a) (pg 168). Explain how.

Proof: b is lower in the Huffman tree than a is, so that implies b came before a in the priority queue. This is because we build the Huffman tree from the bottom up, selecting the next lowest subtree to connect to the tree we are building. For example, since we connected the b, c subtree to the e, d subtree rather than to a in figure 4.16a, this implies $freq_a \geq freq_b + freq_c$ as the ed and bc subtrees must have the lowest current values. This implies $freq_b \leq freq_a$, which is clearly not the case in the frequencies we have. Thus, our data must not be the same as that used to make figure 4.16a. **Q.E.D.**

6 Problem 6

Q: You have a group of contestants for a triathlon. Their running, swimming, and biking times are known. Only one person can swim at a time, but anyone can run/bike at the same time. Develop, prove, and analyze an efficient algorithm to schedule the athletes to run the athletes in that minimizes the total time of the competition. (NOTE: athletes MUST swim first, then bike, then run).

Algorithm
Order the racers in descending order based on the sum of their running and biking times.

Proof Of Correctness
Firstly, note that obviously adding idle times between the starting of each athlete can certainly not improve the overall finish time, so assume all athletes start one after the other.

Consider an arbitrary racing schedule. Suppose the schedule is not monotonically non-increasing in total run/bike times. Then their are two athletes a_i and a_j where a_j follows a_i in the schedule yet the running/biking time of a_j is greater than that of a_i. We will show this implies there exists a consecutive pair of athletes that has this same property. Consider a_{i+1}. If the running/biking times of a_{i+1} is above that of a_i, we have found the pair we are looking for. Else, a_{i+1} is also shares the aforementioned property with a_j. We can repeat this search process until we find the pair we are looking for, or it terminates at a_{j-1}, in which case our pair is a_{j-1}, a_j and we are still done.

Consider an arbitrary racing schedule. Suppose athlete a_i is a slower biker/runner (combined) than the athlete immediately preceding him in the schedule, a_j. Let s_i, s_j be the swimming times, b_i, b_j the race start times, r_i, r_j the run/bike (combined) times, and f_i, f_j the finish times of the athletes respectively. As a_i follows a_j, $b_i = b_j + s_j$. As $r_i \geq r_j$ by assumption, it follows that:

$$f_i = b_i + s_i + r_i = b_j + s_j + s_i + r_i \geq b_j + s_j + s_i + r_j \geq f_j + s_i$$

.

Swapping the spots of athletes i and j clearly lowers f_i as a_i gets to start sooner, and it increases f_j by s_i as now a_j has to wait for a_i to swim first. However, we already see that $f_i \geq f_j + s_i$, so this cannot increase the maximum finish time between the two runners. As we showed earlier, as long as the run/bike times are not monotonically non-increasing, we can always find a pair of consecutive athletes a_i and a_j such that a_i is a slower biker/runner than a_j where a_i follows a_j in the race, and that swapping the two cannot increase their finishing times. Furthermore, since their summed swimming time remains the same, swapping the two will not affect the finishing times of any runners after them as well. Thus, any schedule, even optimal ones, can be converted to our greedy schedule without increasing the max finishing time, and thus our greedy algorithm must be optimal.

Complexity
Sorting the athletes by their combined running/biking time is a simple sort. It could be done in $O(nlogn)$ using quicksort or merge sort.