Compromise

- Keep a 2-D array. For each column, output the majority value.

Marbles on the Tree

- Traversing: Imagine a “BFS” from the leaf nodes to the root node by iteratively removing leaf nodes.
- Keep a queue of leaf nodes. When we pop a node from the queue, we also remove it from the tree (after some updating). When a node becomes a leaf node, we push it to the end of the queue.
- For each leaf node v popped from the queue, we check its number of marbles m_v. We move $m_v - 1$ marbles from v to its parent and remove v from the tree. If the parent becomes a leaf node, we push it to the queue.
- We stop when the queue is empty and return the total number of moves done.

Marbles on the Tree

Intuitions

- The total number of movements equals the total flow in and out of a node, for all nodes.
- For each leaf node, the only movement that will happen is one that go to or from its parent and there is only one such movement.
- When a node becomes a leaf node, all the movements with its children are already arranged so we only need to deal with the movements that it will have with its parent.

Enemy Division

- We only need 2 groups in any case.
- First, divide the soldiers arbitrarily into 2 groups.
- Each iteration, find a soldier that has more than 1 enemy in the same group and move that soldier to the other group.
- Let's say we move the soldier from group A to group B. In A, the soldier has at least 2 enemies and in B the soldier has at most 1 enemy (because each soldier has no more than 3 enemies). When we do this, we reduce the number of enemy pairs in both groups by at least one each iteration. Because the number of enemy pairs m is O(n), we only need O(n) iterations until we reach the desired state. Therefore, this greedy strategy always works and is optimal because it only needs 2 groups.

HW2 - Q1

- A. Yes, it is a legal prefix coding. This is because, given a code string, we can without any ambiguity separate recover the letters as 0 serves as the separator. Whenever we see a 0 in the input string, we break and recover a letter.

- B. No, this is not an optimal strategy.

Example: Assume we have 4 letters a, b, c, and d each with frequency 10. Genius will assign, 0 to a, 10 to b, 110 to c, and 111 to d. Here the average code length will be 2.25. Consider the following legal code, a -> 11, b-> 00, c -> 01, and d -> 10. It is easy to verify that this is a legal code. Here, the average code length is 2.

HW2 - Q2

- A. Yes, it is a legal prefix coding. The division only stops when there are either 2 or 1 symbol. In either case, the symbols are assigned to leaf nodes. Therefore, the prefix coding is legal.
- B. No, this is not an optimal strategy.
- Example: Consider 4 symbols a:10, b:10, c:1, d:1. With the encoding scheme above, we get a = 00, b = 01, c= 10, d= 11. Hence, the average code length is 2. Consider the encoding: a = 0, b=10, c=110, d=1110. Here the average code length is = 1.68.

HW2 - Q3

- A. $T_1(n) = T_1\left(\frac{n}{2}\right) + 1 : O(\log n)$
- B. $T_2(n) = T_2\left(\frac{n}{4}\right) + \log\log n : O(\log n \,.\log\log n)$
- C. $T_3(n) = 2T_3\left(\frac{n}{2}\right) + n : O(n \log n)$
- D. $T_4(n) = 3T_4\left(\frac{n}{4}\right) + n : O(n)$

HW2 – Q4

- Sort the jobs in order of latest to earliest finishing time. Set time t to be the latest finishing time out of every job in the list. Initialize an empty max heap which ranks jobs based on their reward.

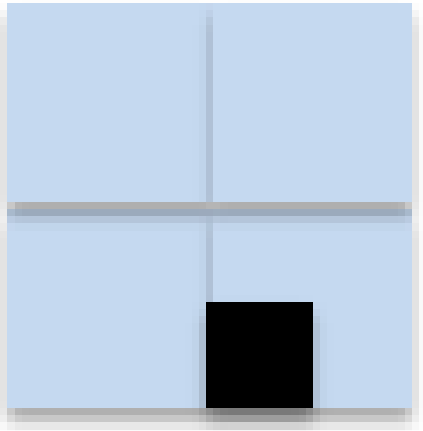
- Iterate through each job with ending time equal to t, removing each job from the list and adding it to the heap. Pop the top of the heap and add that job to the schedule at time t-1. If the heap is empty, set t to the max finishing time of jobs remaining in the list, otherwise decrement t by 1.

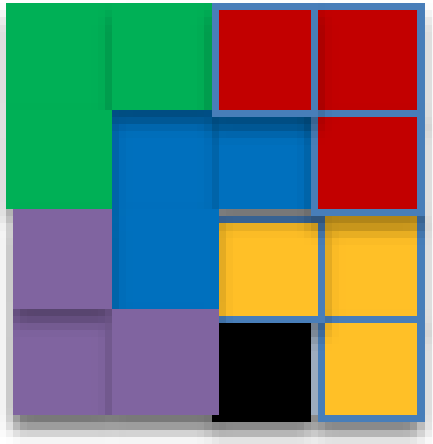
- Repeat steps 2 and 3 until t equals 0 or there are no more jobs to schedule.

Justification for O(nlogn) time:

Step 1 is a sort, which takes O(nlogn).
Step 2 is O(nlogn) because we iterate over each job and add it to the heap exactly once.
Step 3 takes t*O(logn) time, so it is less than or equal to O(nlogn).

HW2 - Q5





The right image contains a hint.

What if we replace the blue squares with black (blocked) squares. Do you see the sub-problems?

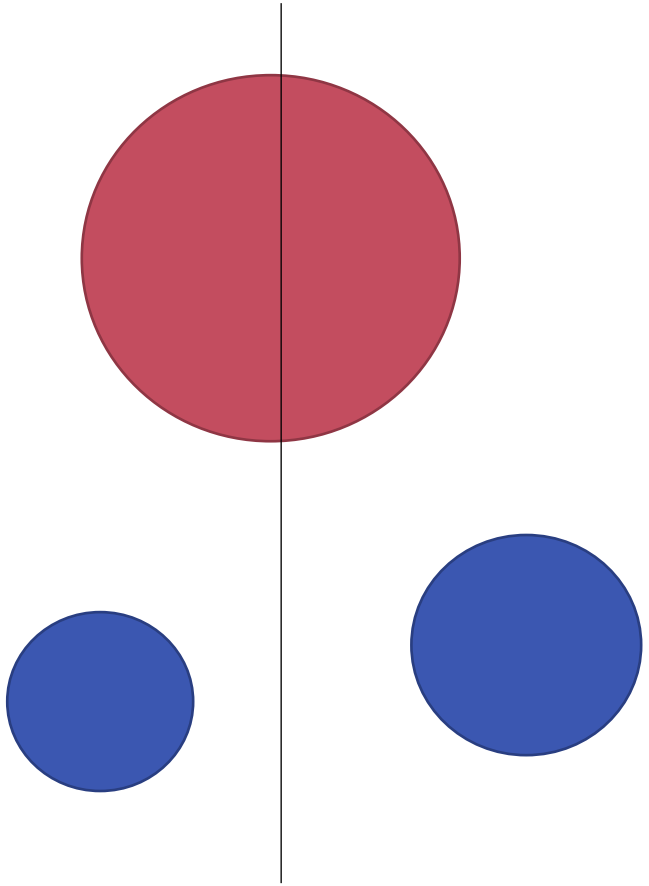
HW2 - Q5

- Trivial case: n = 2^1. One corner is blocked so we simply fill the remaining squares with 1 L-shaped tile.
- The solution for case n = 2^{k+1} can be built using the solution for the case n = 2^k.
- [Divide]: Divide the big square into 4 quadrants. The quadrant containing the blocked square can be tiled using the solution for the case n = 2^k. Place a L-shaped tile in the position like that of the blue squares. Now the remaining part of each quadrant can be tiled using the solution for the case n = 2^k.
- [Conquer]: Trivial.
- T(n) = 4*T(n/2) + O(1) => O(n^2)

HW2 – Q6: O(nlogn) Algorithm

- Arbitrary order the vertices in the graph.
- [Divide]: Recursively divide the graph into two equal vertex graphs and solve the problems on the left and the right induced subgraph. For each subgraph, we want to check if the subgraph has a clique of size more than half of the total vertices in it. If so, we return an arbitrary vertex of the clique. Else, return None.
- [Conquer]: Let l and r be the return values from the left half and right half subgraph.
  - If l = r = None, return None.
  - Else, let l = v1, r = v2. Count the number of neighbors of v1 in both subgraphs. Do the same for v2. We want to know if either v1 or v2 has more than half the number of nodes in both subgraphs as neighbors. If either of them does, we return that vertex. If both do, v1 and v2 must belong to the same clique so we arbitrarily return v1 or v2. Else, we return None.
- $T(n) = 2T(n/2) + O(n) \Rightarrow O(n\log n)$.

HW2 – Q6: O(nlogn) Algorithm



Red circle is the dominant clique.

If we split the graph in half, then at least of half of the dominant clique is the dominant clique in its subgraph. If both half of the red circle are not dominant cliques, then the red circle cannot be a dominant clique in the first place.

HW2 – Q6: O(n) Algorithm

- Initially we start with all vertices in the ‘live’ vertex set.
- Arbitrarily pair up the vertices in the ‘live’ set.
- For each pair, we query if there exists an edge between them.
  If there is an edge, we arbitrarily remove one of the vertices from the set.
  Otherwise, we remove both the vertices.
- If there exists an edge between two vertices in the final iteration (before the ‘live’ set contains at most one vertex), we return YES, otherwise return NO.
- $T(n) = T(n/2) + O(n) \Rightarrow O(n)$. We throw away at least half the vertices each step.

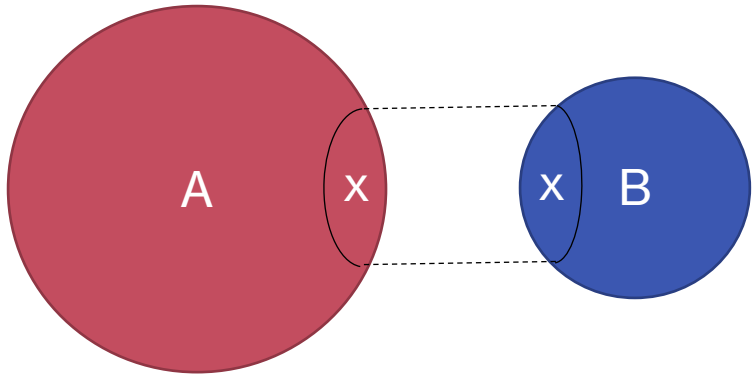
- Correctness: If there exists a dominant clique in the graph, then after each iteration, the dominant clique become even more dominant (shown in the next slide). Hence, at the end, we will remain with a pair of vertices both belonging to the dominant clique. Otherwise, we remain with a pair of independent vertices in the last iteration.

HW2 – Q6: O(n) Algorithm

Why does the dominant clique become more dominant?

For simplicity, assume there are 2 cliques: a dominant clique A of size a and a less dominant clique B of size b.

Let's consider the ratio of the size of A and the size of B after each iteration.



Ratio before the iteration:

$$\frac{a}{b} > 1$$

Ratio after the iteration:

$$\frac{(a-x)/2}{(b-x)/2} = \frac{a-x}{b-x} > \frac{a}{b}$$

where x is the number of pairs of which one node belongs to A and the other one belongs to B.

ratio is even larger because of the pairs formelf there are more than 2 cliques, the latter d by different nodes from the non-dominant cliques.

HW2 – Q7

- Sort the points according to their x coordinates.
- **[Divide]** Recursively divide the problem into two equal sized subparts containing left and right half of the points. Solve the problem on each subpart independently.
- **[Conquer]** Let d_left, d_right be the optimal perimeter from the two subparts. Let d = min (d_left, d_right).
  Let mid be the vertical line dividing left and right subparts.
  Let S be the set of all points within distance d/2 from mid, i.e., we consider all points in a d length strip centered at mid.
- Sort points in S according to their y coordinates.
- Let S' be the set of first 64 points from S. Compute the perimeter of the triangle formed by each triplet in S' and keep track of the minimum perimeter triangle.
- In each iteration, update S' by removing the point with least y coordinate and adding next point from S, and repeat 5.
- d_mid = optimal perimeter found in strip S. Return min(d, d_mid).

HW2 – Q7

Running time analysis:

- Presort the points according to x coordinate and y coordinate. This step takes O(n log n) time.
- Given the presorted points, merge step runs in O(n) time: For each set of 64 points, we spend a constant amount of time. There can be at most n such sets. Hence, the overall running time is **O(n log n)**.

Correctness:

- Divide strip S into small squares with side length d / (sqrt(2). 4) -- we choose the squares with this side length to make sure each square contains at most 2 points. For a square with side length d/ (sqrt(2) . 4), the length of farthest points in the square is d/4 (these are the endpoints of the diagonal).
- Each such square can contain at most 2 points. Otherwise, in case if a particular square contains 3 or more points then there exists a triangle lying entirely in one of the halfs with perimeter less than d (using the fact that a pair of points inside any square will be at most d/4 distance away).
- We only need to compare the points which are at most d distance apart from each other.