Problem 1

Have to order jobs in order of minimizing the weighted sums

Result: Suppose there is set of jobs $J = (1, 2, 3, \ldots, n)$, and for each job in set J there is a corresponding weight and time, for example, job i will have w_i, t_i, and a completion time of c_i. We want to minimize $\sum_{i=1} w_i c_i$. Our greedy algorithm will do the job with the larger $\frac{w_i}{t_i}$ first.

- Sort J by w_i/t_i in decreasing order
- complete jobs in this order

Proof: Once again the jobs are in the set J. This time, there is an adjacent pair that $\frac{w_i}{t_i} < \frac{w_{i+1}}{t_{i+1}}$, or $w_i t_{i+1} < w_{i+1} t_i$. The only completion times that will be affected are i and $i+1$.

Let T be equal to $\sum_{i=1}^{n} t_i$ for all prior jobs of

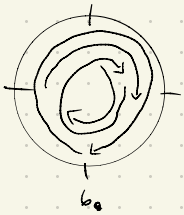
Greedy Algorithm: $= w_i(T + t_i) + w_{i+1}(T + t_i + t_{i+1})$

Other ordering: $= w_{i+1}(T + t_{i+1}) + w_i(T + t_{i+1} + t_i)$

Subtracting the greedy from the other ordering gives you $(w_i t_{i+1} - w_{i+1} t_i)$. We subtract to see if the completion time will be less than 0, which it will be because it's out of order. You can always swap a pair of elements and get a smaller sum of c_i. This can be done in a set of unsorted elements, until the set is sorted. This shows that our greedy algorithm is optimal.

Runtime: Sorting takes $O(n \log n)$ at the worst, so this algorithm should take $O(n \log n)$.

Problem 2



Accept as many jobs as possible

Result: $I_1, \ldots, I_n$ is the set of n intervals. The goal is to create a situation like the Interval Scheduling Problem.

Let t be a time in I_i. We will then delete I_i and all jobs that overlap this job. We can now start the timeline at this time t, as the remaining intervals won't contain this time. This is an instance of the Interval Scheduling Problem

We must now compute a solution of max size for each $i = 1, \ldots, n$. We pick largest of these solutions for the solution.

Proof: We consider the optimal solution for this problem. The optimal solution should contain a form of I_i in the set of jobs in the solution.

Runtime: Assuming intervals are sorted by ending time, the first part of the algorithm takes $O(n)$. Computing the max size for each i will take $O(n^2)$, so, overall, it is polynomial time

Problem 3

Result:

We will simply divide and conquer the set very similarly to the merge sort alg.
It will be recursive.

```
Given set C of cards
if C.length == 1
    return only card in set
if C.length == 2
    if both cards are equal
        return one of them
Set1 is first half of C
Set2 is last half of C
If recursively calling this function with Set1 returns a card
    Compare returned card with Set1
        if it matches:
            return cards that matches
If recursively calling this function with Set2 returns a card
    Compare returned card with Set2
        if there is match:
            return cards that matches
```

If the returned set has more than $n/2$ equivalent there will be a majority equivalent.

Runtime: The recurrance relation for this algorithm is $T(n) = 2T(n/2) + 2n$.
It has been shown that this will equal $O(n \log n)$.

Analysis: It is very similar to merge sort, where it splits the set into two and keeps doing so until the set gets to one or two values. If more than $n/2$ are equivalent, there will be at least one of the split sets where at least half the cards are equivalent to the majority prefrence. There will be a card returned that will get compared to the rest of the set.

Problem 4

We need to show how to find a local minimum of G only using $O(n)$ probes. G is a $n \times n$ grid graph.

Result: Given a $n \times n$ grid graph G
- Probe all grids of column in middle of G (smallest is V_i)
- Probe surrounding columns of V_i,
  - if all larger, v_i is local minimum
  - else, one of the two is smaller than V_i, choose this subgraph to continue
- Probe all grids of row in middle of the subgraph (smallest is v_j)
  - if $v_j > v_i$
    - we choose square subgraph with v_i
  - else if $v_j < v_i$
    - Probe surrounding rows of v_j
      - if all larger, v_j is local minimum
      - else, one of the two is smaller than v_j, choose this subgraph for continuing
        - Probe all grids of column in middle of subgraph
        - (Keep repeating until local minimum is found)

Runtime: $T(n) = T(n/2) + 3n/2 + 4 = 3n + 4\log n = O(n)$.
It will run in linear time. We unroll this recurrence relation.

Analysis: The first step divides G into two $n/2 \cdot n$ rectangular subgraphs, from this step we find the next rectangular subgraph if we cannot find the local min, and keep probing from there. The graph continues to get smaller, going to $n/2 \cdot n/2$ subgraphs, which are now square. Eventually the local minimum will be found, and in linear time. We have satisfied the conditions that were necessary for the problem.

Problem 5

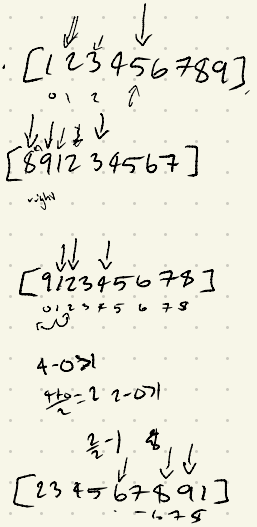
Have to find how many positions an array has shifted.

Result:

Given array shifted[ ], and size of shifted n
let i=0, j=n
while j-i > 1 do
  let l = $[\frac{j+i}{2}]$
  if shifted[l] < shifted[l-1]
    output l ——> end
  else if shifted[l] > shifted[l+1]
    output l+1 ——> end
  else if shifted[l] < shifted[j]
    j=l
  else
    i=l
return 0 —> end

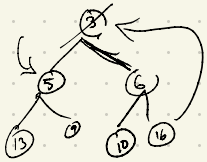
Runtime: This is a modified version of binary search, which runs at O(log n). Our algorithm is very similar and will run in the same time: O(log n). All of the calculations run in O(1), and the if in the beginning is also O(1).

Analysis: The main idea is to find the minimum element, but using binary search. The minimum will be when the selected index is lower than the previous index. This is the condition we check with each iteration. We also check the index in front as well. If it is not, we have to get rid of half the array, so, when choosing the left or right half we just check if the end is greater. If it is, we check the left half. If not, the left half.



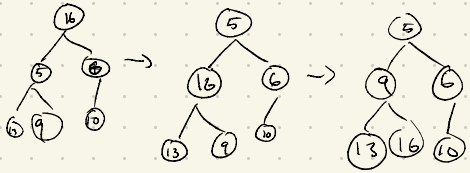
Problem 6:

> 6. Consider a (balanced) heap on n nodes. Show details of how you extract the minimum, insert a new number, and change a number (along with the corresponding post heapify process). Analyze the time complexity of your three algorithms.



a) Extracting the minimum

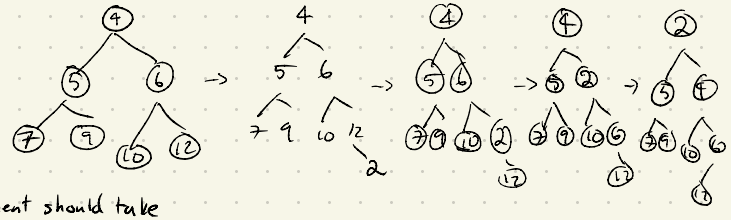
Result: Locate minimum (root)
Delete it, replace it with farthest right node
heapify(node):
if node is smaller than it's two children, stop
else if:
compare two children, replace with smallest



Runtime: Finding and deleting the min will take O(1). Heapify process takes O(log n).
In total, extracting the minimum will take O(log n).

b.) Inserting number

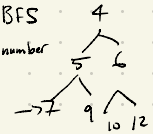
Result: Add new element to far right of heap.
While element is lesser than its parent:
swap element with its parent



Runtime: At the worst case adding an element should take O(log n). This would be where we swap all the way to the root, only going through half the heap.

c) Change a number

Result: Search for number going to be changed using BFS
Delete found number and replace with new number
heapify(replacement):
if number is smaller than parent:
swap with parent
recurse heapify
else if number is larger than one of its children
swap with lowest parent
recurse heapify
else
number is in correct place → end



Runtime: O(log n), as searching for number takes O(log n) and heapifing takes O(log n).
All in all, it will take O(log n).