1. Big Oh notation: see solutions on canvas: [https://canvas.uw.edu/courses/1021503/ quizzes/877365](https://canvas.uw.edu/courses/1021503/quizzes/877365).

2. Consider an instance of the stable matching problem, and suppose that $M$ and $M'$ are two distinct stable matchings. Show that the men who prefer their match in $M$ to their match in $M'$ are matched in $M$ to women that prefer their match in $M'$ to their match in $M$.

   Proof by contradiction: Suppose that there is a man $m$ that is matched to $w$ in $M$ and to $w'$ in $M'$, $m$ prefers $w$ to $w'$, and $w$ prefers $m$ to her match $m'$ in $M'$. This contradicts the stability of $M'$, because the pair $m, w$ in $M'$ is unstable.

   ![Figure 1](image_url) Figure 1: This is the example, illustrating the proof above. The black matching is $M$ and the orange matching is $M'$. The lists beside $m, w$ shows their preference list. We know that $m$ prefers his match in the black matching, $w$, to his match in the orange matching, $w'$. For purposes of contradiction, we assumed that $w$ prefers $m$ to $m'$. As you can see, this implies that the pair $(m, w)$ is an instability, a contradiction.

3. Problem 2.5 (page 79 of online version of [DPV]), parts (a) (c) (e) and (g). For each part, specify the number of recursive subproblems at each level of the recursion tree, the size of each of these subproblems, and the work required to combine the results from the next level. Specify the depth of the tree, give an expression for the value of $T(n)$ in terms of these quantities (as I did in class on January 8), and a $\Theta$ bound for $T(n)$. 
<table>
<thead>
<tr>
<th>Function</th>
<th>nodes at level j</th>
<th>subproblem’s size</th>
<th>depth</th>
<th>work to combine</th>
<th>total work</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/3) + 1$</td>
<td>$2^j$</td>
<td>$n/3^j$</td>
<td>$\log_3 n$</td>
<td>$2^j$</td>
<td>$\sum_{i=1}^{\log_3 n} 2^i = \Theta(n^{\log_3(2)})$</td>
</tr>
<tr>
<td>$T(n) = 7T(n/7) + n$</td>
<td>$7^j$</td>
<td>$n/7^j$</td>
<td>$\log_7 n$</td>
<td>$7^j n/7^j = n$</td>
<td>$n \log_7 n = \Theta(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = 8T(n/2) + n^3$</td>
<td>$8^j$</td>
<td>$n/2^j$</td>
<td>$\log_2 n$</td>
<td>$8^j (n/2^j)^3 = n^3$</td>
<td>$n^3 \log_2 n = \Theta(n^3 \log n)$</td>
</tr>
<tr>
<td>$T(n) = T(n - 1) + 2$</td>
<td>1</td>
<td>$n - j$</td>
<td>$n$</td>
<td>2</td>
<td>$2n = \Theta(n)$</td>
</tr>
</tbody>
</table>

4. Read the $n \log n$ lower bound for sorting on page 59 of [DPV]. Then solve Problem 2.20 on page 81.

The idea of the algorithm is to create an array $A$ with enough entries for all of the integers between the minimum value in $x[1..n]$ and the maximum value. Then in one pass through $x$, we count the number of occurrences of each element. Finally, we create a sorted version of the array.

Let $\text{min} := \text{minimum value in array } x$;
Let $\text{max} := \text{maximum value in array } x$;
Let $M := \text{max} - \text{min}$;
(These can be found in one pass over the array.)

Let $A[0..M]$ be an array, with all entries initialized to 0.
for $i := 1$ to $n$ do
    increment $A[x[i] - \text{min}]$; // This tracks the number of times
    // the value $x[i]$ occurs in the array $x$

Create a new array $y[1..n]$ // This will contain the sorted output.
Let $j := 0$;
for $i := 0$ to $M$
    for $k := j$ to $j + A[i] - 1$; // copy all occurrences of $i$+min into the array $y$
        $y[k] := i + \text{min}$;
    $j := j + A[i]$

An example:

$x = \{8, 5, 10, 5, 6\}$

Before starting the algorithm:

$i + \text{min} = 5 6 7 8 9 10$
$A[i] = 0 0 0 0 0 0$
After the first loop:

\[
\begin{align*}
  i + \text{min} &= 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \\
  A[i] &= 2 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1
\end{align*}
\]

The sorted array:

\[
y = \{5, 5, 6, 8, 10\}
\]

Running time: \(O(n + M)\).

Proof of Correctness:

After the construction of \(A[0..M]\) is complete, \(A[i]\) stores the number of times the value \(\text{min} + i\) occurs in the array \(x[1..n]\). We then copy these values in sorted order into the array \(y[1..n]\).

The lower bound does not apply here because it applies only to algorithms that use comparisons. This algorithm never performs comparisons between elements, rather, just uses the fact that the integers lie in a small range.

5. Problem 2.23 (page 82 of online version of [DPV]).

(a) The algorithm partitions the array of length \(n\), \(A[0..n]\), into two halves, say \(L\) and \(R\), and recursively finds the majority in each half (if there is one). The algorithm then checks in linear time which of these two possible values is the majority in the overall array by a single pass through the entire array.

```plaintext
function majority (A[1..n]):
    if n = 1 then // Base case
        return A[1];
    Let L := A[1..floor(n/2)]; // The first half of the array
    Let R := A[floor(n/2)+1..n]; // The second half of the array
    Let x := majority (L); // compute the majority for the first half recursively
    Let y := majority (R);
    c(x) := number of occurrences of x in A; // this can be computed in linear time.
    c(y) := number of occurrences of y in A;
    if c(x) > n/2 then // Checks to see if x is the majority
        return x;
    else if c(y) > n/2 then
        return y;
    else // If the array does not have any majority
        return NULL;
```

Let \(f(n)\) be the running time of the algorithm on an array of length \(n\). We have

\[
f(1) = O(1)
\]

and for \(n > 1\), we have

\[
f(n) = 2f(n/2) + O(n)
\]
Therefore, by master theorem:

\[ f(n) = O(n \log n). \]

Proof of correctness:

By induction on the length of the array: For a length of 1, as a base case, the only element of the array is the majority.

Inductive hypothesis: The algorithm works correctly on an array of length strictly less than \( n \). We need to show that it is correct when there are \( n \) elements in the array. Let \( A \) be the array, with elements \( A[1], \ldots, A[n] \). Let \( x \) be the majority of \( A[1], \ldots, A[n/2] \), if there is one, and \( y \) the majority of \( A[n/2+1], \ldots, A[n] \), if there is one.

If there is no majority element in the entire array, then both \( c(x) \) and \( c(y) \) will be at most \( n/2 \), so we will output NULL.

If there is a majority element in the entire array, say \( m \), then by the pigeonhole principle, there must be a majority element in one of the two halves. (To see this, suppose that the length of the first part of the array is \( n_1 \) and the length of the second part is \( n_2 \), with \( n_1 + n_2 = n \).

Suppose also that the number of occurrences of \( m \) in the first part is \( k_1 \) and the number of occurrences of \( m \) in the second part is \( k_2 \). If \( k_1 \leq n_1/2 \) and \( k_2 \leq n_2/2 \), then the total number of occurrences of \( m \) in the array, which is \( k_1 + k_2 \leq (n_1 + n_2)/2 \leq n/2 \), contradicting the assumption that \( m \) was a majority element.

Therefore, \( m = x, m = y \) or \( m = x = y \). By the inductive hypothesis, the algorithm correctly determines the majority value for each half, and therefore the final scan, during which we count the number of occurrences of both \( x \) and \( y \), will correctly determine the overall majority.

(b) Using the idea suggested by the hint:

```plaintext
function majority (A[1..n]):
    if n = 1 then // Base case
        return A[1];
    Let B := an empty array of size 0;
    for i := 1 to floor(n/2) // We will omit the last element
        if A[2i - 1] = A[2i] then //If the two elements in the pair are equal
            add A[2i] to the end of B
    If n is odd then
        candidate := A[n];
    Let x := majority (B); //Compute the majority for the surviving elements
    if number of occurrences of x in A > n/2 //Check if the majority of B is
        return x;
    else if number of occurrences of candidate > n/2
        return candidate;
    else
        return NULL
```
Analysis of the algorithm:

Fact: The length of $B$ is at most half the length of $A$, since out of each pair of elements at least one is thrown away. So there is a single recursive call on an array of size at most $n/2$. The rest of the work done by the algorithm takes linear time. Thus, we obtain the recurrence

$$T(1) = O(1) \quad T(n) = T(n/2) + O(n) \quad n > 1.$$  

It follows from the master theorem that $T(n) = O(n)$. Alternatively, you can show this with a simple calculation: the number of nodes at level $j$ in the recursion tree is 1, the size of the subproblem at level $j$ is at most $n/2^j$, and the work at level $j$ is $O(n/2^j)$. Finally, using the fact that the geometric series converges, specifically,

$$\sum_{j=0}^{\infty} \frac{1}{2^j} = 2$$

we obtain the result.

Proof of correctness:

We prove the correctness by induction on length of the array. The base case is when the array is of length 1, in which case the algorithm returns the only element in the array, which is the majority.

Assume that the algorithm is correct for any array of length less than $n$, we will prove the correctness when the length is equal to $n$.

We consider three cases:

- There is no majority in $A$: In this case, NULL is returned, because both $x$ and candidate will fail the counting test.
- The array is of odd length and the majority element is $A[n]$. In this case, the algorithm will correctly return ”candidate” (which is $A[n]$).
- We have left only to deal with the case where the array has even length, say $n = 2k$, and there is a majority element, say $m$. In this case, we will show that the value $x$ returned by the recursive call on $B$ is equal to $m$. Since by hypothesis, the recursive call works correctly, we only need to show that in fact $m$ is the majority element of $B$.

To see this, consider a partition of $A[1..2k]$ into pairs and let

- $n_{m-m}$ be the number of pairs both of whose elements are $m$.
- $n_{m-x}$ be the number of pairs where one of the elements is $m$ and the other one is some non-majority value.
- $n_{x-x}$ be the number of pairs both of whose elements are a non-majority value and they are identical.
- $n_{x-y}$ be the number of pairs both of whose elements are non-majority value and they are not the same.

Clearly the number of pairs satisfies

$$n_{m-m} + n_{m-x} + n_{x-x} + n_{x-y} = k,$$
and $m$ is the majority element in $A$ so

$$2n_{m-m} + n_{m-x} > k.$$ 

Together, these two imply that

$$2n_{m-m} + n_{m-x} > n_{m-m} + n_{m-x} + n_{x-x} + n_{x-y},$$

so

$$n_{m-m} > n_{x-x} + n_{x-y}.$$ 

But the number of elements in $B$ is

$$n_{m-m} + n_{x-x},$$

so $m$ is the majority element in $B$.

Here is an example that shows that the surviving elements can have a majority element even when $A$ doesn’t. (This is one of the reasons we need to count the number of occurrences of $x$.)

$$A = \{1, 3, 4, 3, 2, 2, 3, 5, 6, 3\}$$

And we have the following pairs:

$$(1, 3), (4, 3), (2, 2), (3, 5), (6, 3)$$

Only the 2 from the pair $(2, 2)$ will survive, thus $B = \{2\}$ and the majority of surviving elements will be 2, although we know that $A$ does not have a majority element.