Assignment 1

Question 1

a)

The for loop on line 3 is $n - 1$ steps and $i \leq n$ on every iteration, so j on line 4 iterates up to at most $n - 1$. Therefore, there are at most $(n - 1)(n - 1) = n^2 - 2n + 1$ steps between the two for loops on lines 3 and 4. Lines 1, 2, 5, 6 and the assignment on line 4 all take constant time. This constant time shall be called c. Therefore, there are at most $n^2 - 2n + 1 + c$ steps which is $O(n^2)$. Therefore, $T(n)$ of strange() is $O(n^2)$.

b)

When $i = 2$ on line 3, the loop on line 4 over j calls the if statement on line 5 once. When $i = 3$, the loop on line 4 over j calls the if statement on line 5 twice. So the if statement on line 5 is executed $1 + 2 + 3 + \ldots + n - 2 + n - 1$ times. Using an easy summation formula, it is evident that $1 + 2 + 3 + \ldots + n - 2 + n - 1 = n(n - 1) / 2$. Using this information, $n(n - 1) / 2 = (n^2 - n) / 2 \geq n^2 / 4$. Lines 1, 2, 5, 6 and the assignment on line 4 all take constant time. This constant time shall be called c. Therefore, $T(n) = (n^2 - n) / 2 + c$. $n^2 / 4$ is $\Omega(n^2)$ and since $(n^2 - n) / 2 + c \geq n^2 / 4 + c$, and $T(n)$ of strange() is equal to $(n^2 - n) / 2 + c$, $T(n)$ of strange is $\Omega(n^2)$.

Since $T(n)$ is both $O(n^2)$ and $\Omega(n^2)$, $T(n)$ is $\Theta(n^2)$.

Question 2

a)

The root of the heap is placed at index 1 of A. The first element of A has the greatest priority in the heap. The next element of A is the left child of the root, then the middle child of the root and then the right child of the root. The elements of A will follow this pattern for every node in the heap. Given index i of any node except the root, the index of the parent is equal to $\lfloor (i + 1) / 3 \rfloor$. Given index i of any node, the index of the left child is equal to $3i - 1$, the index of the middle child is equal to $3i$ and the index of the right child is equal to $3i + 1$.

b)

The first $\lceil (n - 1) / 3 \rceil$ nodes of A represent the internal nodes of a heap with n nodes. When $n = 1$, there are no internal nodes and $\lceil (1 - 1) / 3 \rceil = \lceil 0 \rceil = 0$, so it satisfies one base case. When $n = 2$, there is one internal node and $\lceil (2 - 1) / 3 \rceil = \lceil 1 / 3 \rceil = 1$, so it satisfies both base cases. For every three additional nodes to a heap which initially has 2 nodes, the amount of internal nodes increase by one. Let h_1 be a heap with n nodes and n be a value greater than 2 by a factor of 3 and let i be the amount of internal nodes in h_1. Let h_2 be a heap with $n + 3$ nodes, then the first $\lceil ((n + 3) - 1) / 3 \rceil$ nodes are internal nodes.

$$
\begin{aligned}
\lceil ((n + 3) - 1) / 3 \rceil &= \lceil (n - 1 + 3) / 3 \rceil \\
&= \lceil (n - 1) / 3 + 3 / 3 \rceil \\
&= \lceil (n - 1) / 3 \rceil + \lceil 3 / 3 \rceil \\
&= i + 1 && \text{* } \lceil (n - 1) / 3 \rceil = i \text{ since } h_1 \text{ has n nodes and i internal nodes}
\end{aligned}
$$

Therefore, h_2 has one more internal node than h_1 and the equation is valid.

The height of a heap with n nodes is $\lfloor \log_3(2n - 1) \rfloor$. $\log_3$ is called because every complete row has exactly 3 times the amount nodes as the previous row. The node most left on any row has the index 3i - 1where i is the index of its parent node. Suppose this is the nth node, then

$$\begin{aligned} 2n - 1 &= 2(3i - 1) - 1 \\ &= 6i - 2 - 1 \\ &= 6i - 3 \end{aligned}$$

Since A[i] is farthest to the left on its respective row, 6i - 3 produces a factor of 9, therefore, $\log_3(2n - 1)$ produces a whole number which is the height of the heap. Now suppose the nth node is farthest to the right on the last row. Since 6i - 3 on a node most left in a row produces a factor of 9, 6i - 3 on a node most right in a row produces one less than a factor of 9 which means that $\lfloor \log_3(2n - 1) \rfloor$ for the node most right in a row produces the same value as $\log_3(2n - 1)$ for the node most left in a row. Therefore, a heap with n nodes has a height of $\lfloor \log_3(2n - 1) \rfloor$ regardless if the final row is complete or not.

c)

Let n = A.Heapsize

```
Insert(A, key)
1       A[n + 1] = key
2       //n += 1
3       i = ⌊(n + 1) / 3⌋
4       j = n
5       while i > 0 and key > A[i]
6               A[j] = A[i]
7               A[i] = key
8               i = ⌊(i + 1) / 3⌋
9               j = ⌊(j + 1) / 3⌋
```

The loop on line 5 will never execute more than $\lfloor \log_3(2n - 1) \rfloor$ times as this is the height of a heap with n nodes and if key has the highest priority in the heap then the loop will execute exactly $\lfloor \log_3(2n - 1) \rfloor$ times. Lines 1, 3, 4, 6, 7, 8, 9 all take constant time represented by c. Therefore T(n) for Insert() is $O(\log_3 n)$ and $\Omega(\log_3 n)$ so T(n) for Insert() is $\Theta(\log_3 n)$.

Extract_Max(A)

```
1	A[1] = A[n]
2	A[n] = NIL
3	//n -= 1
4	i = 1
5	max_node = max of [A[3i - 1], A[3i], A[3i + 1]]
6	j = max_node.index
7	while j ≤ n and max_node > A[i]
8		A[j] = A[i]
9		A[i] = max_node
10		max_node = max of [A[3j - 1], A[3j], A[3j + 1]]
11		i = j
12		j = max_node.index
```

The loop on line 7 will never execute more than $\lfloor \log_3(2n - 1) \rfloor$ times as this is the height of a heap with n nodes and if the last key in A has a smaller priority than one of its children after every switch then the loop will execute exactly $\lfloor \log_3(2n - 1) \rfloor$ times. Lines 1, 2, 4, 5, 6, 8, 9, 10, 11, 12 all take constant time represented by c. Therefore T(n) for Extract_Max() is $O(\log_3 n)$ and $\Omega(\log_3 n)$ so T(n) for Extract_Max() is $\Theta(\log_3 n)$.

```
Update(A, i, key)
1    A[i] = key
2    j = ⌊(i + 1) / 3⌋
3    parent = A[j]
4    while j > 1 and A[i] > parent
5        A[j] = A[i]
6        A[i] = parent
7        i = j
8        j = ⌊(j + 1) / 3⌋
```

The loop on line 4 will never execute more than $\lfloor \log_3(2n - 1) \rfloor$ times as this is the height of a heap with n nodes, and if key is higher than any priority currently in the heap and i = n then the loop will execute exactly $\lfloor \log_3(2n - 1) \rfloor$ times. Lines 1, 2, 3, 5, 6, 7, 8 all take constant time represented by c. Therefore T(n) for Update() is $O(\log_3 n)$ and $\Omega(\log_3 n)$ so T(n) for Update() is $\Theta(\log_3 n)$.

```
Remove(A, i)
1   A[i] = A[n]
2   A[n] = NIL
3   //n -= 1
4   max_node = max of [A[3i - 1], A[3i], A[3i + 1]]
5   j = max_node.index
6   while j ≤ n and max_node > A[i]
7       A[j] = A[i]
8       A[i] = max_node
9       max_node = max of [A[3j - 1], A[3j], A[3j + 1]]
10      i = j
11      j = max_node.index
```

The loop on line 6 will never execute more than $\lfloor \log_3(2n - 1) \rfloor$ times as this is the height of a heap with n nodes and if the last key in A has a smaller priority than one of its children after every switch then the loop will execute exactly $\lfloor \log_3(2n - 1) \rfloor$ times. Lines 1, 2, 4, 5, 7, 8, 9, 10, 11 all take constant time represented by c. Therefore T(n) for Remove() is $O(\log_3 n)$ and $\Omega(\log_3 n)$ so T(n) for Remove() is $\Theta(\log_3 n)$.