Problem 1

a) The worst case is when the algorithm partitions the array in the most uneven fashion, if the algorithm partitions it such that the first partition only has 1 element, and the other partition has $n-1$ elements. Since, the time it takes for the algorithm to partition an array, is the current size. Our runtime may look something like this

$$n + (n-1) + (n-2) + (n-3) + (n-4)... + 2$$

We end the summation at 2 since the minimum array size to be partitioned is 2. From here, we know that the closed form of the arithmetic sum looks like this

$$\sum_{k=1}^{n} k = n + (n-1) + (n-2) + ... + 1 = \frac{n^2+n}{2}$$

So we can write the summation of the quicksort runtime as such

$$n + (n-1) + (n-2) + (n-3) + (n-4)... + 2 = \frac{n^2+n}{2} - 1$$

From here we can see that n^2 is the overpowering term in the equation. So we can safely say that the worst-case runtime of quicksort is $\Theta(n^2)$

b) The best case scenario is when the algorithm perfectly partitions the array in half each time. If we have an input of length n, then the maximum number of times we can half that array is $log_2(n)$. For each partition, we still have to go through all n elements. So we have it that our best case runtime is $\Theta(n \log n)$. We can also represent this in a recurrence relation. Since we perfectly partition the array of size n in half each time, and for each level of the recursion, we go through all n elements. We can write our recurrence as such

$$T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$$

From here, we can use the simplified masters theorem where $a = 2$, $b = 2$ and $d = 1$. Since $a = b^d$ we can say that this recurrence has a runtime of $\Theta(n \log n)$

Problem 2

a) Let us use the guess $O(n^4)$. We need to show that $T(n) \leq cn^4$ for $n > n_0$. We can assume that $T(m) \leq c \cdot m^4$ such that $m < n$. We can let $m = n - 1$

$$\begin{aligned} T(n) &= T(n-1) + n \\ &\leq c \cdot (n-1)^4 + n \\ &= c \cdot (n^4 - 4n^3 + 6n^2 - 4n + 1) + n^3 \\ &\leq c \cdot (n^4 + 6n^2 + 1) + n^3 \end{aligned}$$

b) Let us use the guess $O(n^2)$. We need to show that $T(n) \leq cn^2$ for $n > n_0$. We can assume that $T(m) \leq c \cdot m^4$ such that $m < n$. We can let $m = \frac{n}{2}$

$$\begin{aligned} T(n) &= 2 \cdot T(\frac{n}{2}) \\ &\leq 2c(\frac{n}{2}) + 2n \\ &\leq c \cdot \frac{n^2}{2} + 2n \\ &\leq c \cdot n^2 + 2n \end{aligned}$$

Problem 3

```
def mergeSort(arr, temp_arr, left, right):
    count = 0
    if left < right:
        mid = (left + right)/2 # this is floored
        count += mergeSort(arr, temp_arr, left, mid)
        count += mergeSort(arr, temp_arr, mid + 1, right)
        count += merge(arr, temp_arr, left, mid, right)
    return count

def merge(arr, temp_arr, left, mid, right):
    i = left
    j = mid + 1
    k = left
    count = 0

    while i <= mid and j <= right:
        if arr[i] <= arr[j]: #checking for inversions
            temp_arr[k] = arr[i]
            i, k += 1
        else: #if there is an inversion
            temp_arr[k] = arr[j]
            count += (mid-i + 1)
            j, k += 1

  #merge the array
    while i <= mid:
        temp_arr[k] = arr[i]
        i, k += 1
    while j <= right:
        temp_arr[k] = arr[j]
        j, k += 1

    for n in range(left, right + 1):
        arr[n] = temp_arr[n]

    return count
```

The algorithm is exactly the same as mergeSort, except that we add a check during the merge for inversions. If we see that there is an inversion whilst iterating through the array, we add the number of elements between mid and the index we find the inversion in j. In this version we simply return the number of inversions instead of the array during the merge sort. So the runtime should be exactly the same, which is $\Theta(nlogn)$

Problem 4

recall the definition of $\Omega(g(n))$

$$\Omega(g(n)) = \{f(n) | \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq cg(n) \leq f(n)\}$$

a) show that if $a > b^d$ then $\Omega(n^{\log_b(a)})$

Proof. let $r = \dfrac{a}{b^d} > 1$. We can write out our geometric sum as such

$$\begin{aligned}\sum_{i=0}^{\log_b(n)} r^i &= \frac{r^{\log_b(n)+1} - 1}{r-1} \\ &\leq \frac{r^{\log_b(n)+1}}{r-1} \\ &= r^{\log_b(n)} \cdot \frac{r}{r-1} \\ &= \Omega(r^{\log_b(n)})\end{aligned}$$

since we know that $r > 1$ we can do the following

$$\begin{aligned} r^{\log_b(n)}) &= n^{\log_b(r)} \\ &= n^{\log_b(\frac{a}{b^d})} \\ &= n^{\log_b(a) - \log_b(b^d)} \\ &= n^{\log_b(a) - d}\end{aligned}$$

Now, if we multiply it with $c \cdot n^d$ to complete the workload equation we get

$$T(n) \geq c \cdot n^d \cdot n^{\log_b(a)-d} = c \cdot n^{log_b(a)}$$

Since it is dependent on the constants, there exists constants c such that $T(n) \geq c \cdot n^{\log_b(a)}$. So if $a > b^d$, then $T(n) = \Omega(n^{\log_b(a)})$ □

b) show that $a < b^d$ then $\Omega(n^d)$

Proof. From here, we can do the same as what we did during lecture, which is if $a < b^d$ then we know that $\dfrac{a}{b^d} < 1$. From here we can use the total work formula and have $r = \dfrac{a}{b^d}$. We then have the geometric progression of r such that $r < 1$ which can be written as

$$\sum_{t=0}^{\infty} r^t = \frac{1}{1-r}$$

If we multiply the summation with $c \cdot n^d$ we get

$$c \cdot n^d \cdot \frac{1}{1-r}$$

Now by the definition of $\Omega(g(n))$ there will exist constants such that $cg(n) \leq f(n)$. So we can say that if $a < b^d$ then $T(n) = \Omega(n^d)$

□

Problem 5

a) Throughout the for loop, the values of $maxSize$ and $minSize$ will increase to a maximum of $\frac{n}{2}$. From here the if statement inside the for loop only occurs every other iteration, so $\frac{n}{2}$ times. Since each function inside the for loop runs in $\Theta(log(m))$ time, where m is the size of the input. We can write out our runtime as such

$$\begin{aligned}
&n \cdot [\log(min) + \log(min) + \log(max)] + \frac{n}{2} \cdot [\log(max) + \log(min)] \\
&= n \cdot [\log(\frac{n}{2}) + \log(\frac{n}{2}) + \log(\frac{n}{2})] + \frac{n}{2} \cdot [\log(\frac{n}{2}) + \log(\frac{n}{2})] \\
&= n \cdot \log(\frac{n^3}{8}) + \frac{n}{2} \cdot \log(\frac{n^2}{4}) \\
&= 3n \cdot \log(\frac{n}{2}) + n \cdot \log(\frac{n}{2}) \\
&= \Theta(n \log n)
\end{aligned}$$

From here we can see that the runtime of $Strange(A, n)$ will become $\Theta(n \log n)$

b) The function should return the element at index $\frac{n}{2}$ floored plus 1. So we can write it as

$$A\left[\lfloor\frac{n}{2}\rfloor + 1\right]$$