Problem 0

1) $2T(\frac{n}{2}) + O(n)$; In the first level we know that it branches off into 2 pathways or $T(\frac{n}{2})$ so we can get this by dividing $2T(\frac{n}{2}) + O(n)$ by 2 to get $2T(\frac{n}{2^2}) + \frac{n}{2}$ we then plug this back into the original equation for T… $2[2T(\frac{n}{2^2}) + \frac{n}{2}] + n$ to get $2^2T(\frac{n}{2^2}) + 2n$. We can repeat the process for the third level to get $2^3T(\frac{n}{2^3}) + 3n$. For k we get $2^kT(\frac{n}{2^k}) + k$ where k is the number of levels in the recursive tree. We are aiming to try to derive the base case at the first level of the tree so if we assume $T(\frac{n}{2^k})$ is trying to reach 1 which is the base base so $T(\frac{n}{2^k}) = T(1)$ which gives us $n = 2^k$ for the base case then we get $k = log_2 n$. Plugging back into the equation we get $2^{log_2 n}T(1) + nlog_2 n$. $2^{log_2 n} = n$ so the equation simplifies to $T(n) = nT(1) + nlog(n)$. Using big O notation we get $T(n) = O(nlogn)$. Therefore, we get even distribution for each level of work that we are doing so we are increasing by a constant factor.
2) $2T(\frac{n}{2}) + O(1)$; Using the same process as the previous question we find the first level by dividing $2T(\frac{n}{2}) + O(1)$ by 2 which gets us $2T(\frac{n}{2^2}) + \frac{n}{2}$. We then plug this back into the original equation for k to get us $2^2T(\frac{n}{2^2}) + 2$. We repeat for this for k levels we get $2^kT(\frac{n}{2^k}) + k$. Assuming $T(\frac{n}{2^k}) = T(1)$ for the base case and $k = log_2 n$ from the previous derivation we get $T(n) = nT(1) + log_2 n$. Using big-O notation we get $T(n) = O(n)$. This is bottom heavy because the work done on each level is O(1) and the number of levels is O(logn). This means that the most work is done at the bottom which comes out to O(n) for the total work.
3) $7T(\frac{n}{2}) + O(n^3)$; Using the same equation from Question (1), we look at the first level which is dividing the equation by 2 to get $7T\left(\frac{n}{2^2}\right) + \frac{n^3}{2}$ plugging back into the original equation for T we get $7^2T\left(\frac{n}{2^2}\right) + 7(\frac{n^3}{2}) + n^3$. For k levels in the recursive tree we get $T(n) = 7^k\left(\frac{n}{2^k}\right) + 7^{k-1}\left(\frac{n^3}{n^{k-1}}\right) + \cdots + 7\left(\frac{n^3}{2}\right) + n^3$. Assuming $T(\frac{n}{2^k}) = T(1)$ for the base case and $k = log_2 n$ from (1), we plug back in for $T(n) = 7^{\log_2 n}T(1) + 7^{(log_2 n)-1}\left(\frac{n^3}{n^{(log_2 n)-1}}\right) + \ldots + 7\left(\frac{n^3}{2}\right) + n^3$. This simplifies to $T(n) = n^{\log_2 7}T(1) + n^3$. Using big-O notation this simplifies to $O(n^3)$. Work is top-heavy because the work done at each level is $O(n^3)$ which is much more than $n^{\log_2 7}$ and the number of levels O(logn). Therefore, the total work is $O(n^3)$.
4) $7T(\frac{n}{2}) + O(n^2)$; This is almost exactly the same proof as the last one with an interesting result. We start by finding the work down for the first level by dividing our equation by 2 for $T\left(\frac{n}{2}\right)$ to get $7T\left(\frac{n}{2^2}\right) + \frac{n^2}{2}$ and plugging back to the original equation we get

$7^2T\left(\frac{n}{2^2}\right) + 7(\frac{n^2}{2}) + n^2$. For k levels of recursion we get $T(n) = 7^k\left(\frac{n}{2^k}\right) +$ $7^{k-1}\left(\frac{n^2}{n^{k-1}}\right) + \cdots + 7\left(\frac{n^2}{2}\right) + n^2$. Assuming T($\frac{n}{2^k}$) = T(1) for the base case and k = $log_2 n$ from (1), we plug back in for $T(n) = 7^{\log_2 n}\mathrm{T}(1) +$ $7^{(log_2 n)-1}\left(\frac{n^2}{n^{(log_2 n)-1}}\right) + \ldots + 7\left(\frac{n^2}{2}\right) + n^2$. This simplifies to $T(n) = \mathrm{n}^{\log_2 7}\mathrm{T}(1) + n^2$. Since $\mathrm{n}^{\log_2 7} > n^2$, there is more work being done at the bottom of the recursion, so we get a bottom-level recursion with overall time complexity O($\mathrm{n}^{\log_2 7}$).

5) $4T\left(\frac{n}{2}\right) + O(n^2\sqrt{n})$; We do the same math and processes of all the prior parts by finding the first level by dividing by 2. For the first level we get $4T\left(\frac{n}{2^2}\right) + \frac{n^2\sqrt{n}}{2}$ which we plug back into the original equation for T to get $4^2T\left(\frac{n}{2^2}\right) + 3n^2\sqrt{n}$. To solve for k levels we get $T(n) = 4^kT\left(\frac{n}{2^k}\right) + (2^k - 1)n^2\sqrt{n}$. Assuming T($\frac{n}{2^k}$) = T(1) for the base case and k = $log_2 n$ from (1), we plug back in for $T(n) = 4^{log_2 n}T\left(\frac{n}{2^{log_2 n}}\right) + (2^{log_2 n} - 1)n^2\sqrt{n}$. Assuming $4^{log_2 n} = n^{log_2 4}$, $4^{log_2 n} = n^2$ and assuming $2^{log_2 n} = n^{log_2 2}$, $(2^{log_2 n} - 1) = (n - 1)$ plugging back in respectively we get $T(n) = n^2T\left(\frac{n}{n}\right) + (n - 1)n^2\sqrt{n}$. This means that $T(n) = O(n^2\sqrt{n})$ since $n^2\sqrt{n} > n^2$. Therefore this recurrence is top heavy because the occurance at each level is greater than n^2. This means the total work done is $O(n^2\sqrt{n})$.

6) 4T($\frac{n}{2}$) + O($nlog_2 n$), following the last steps lets divide by 2 to find the first level of occurrence for $T\left(\frac{n}{2}\right)$. For the first level we get $4T\left(\frac{n}{2^2}\right) + \frac{nlog_2 n}{2}$ and we plug in to get $4^2T\left(\frac{n}{2^2}\right) + nlog_2 n^2 + nlog_2 n$. If we repeat this k times we get $T(n) = 4^kT\left(\frac{n}{2^k}\right) +$ $nlog_2 n^{2^{k-1}} + \cdots + nlog_2 n$. Assuming T($\frac{n}{2^k}$) = T(1) for the base case and k = $log_2 n$ from (1), we plug back in for $T(n) = 4^{log_2 n}T(1) + nlog_2 n^{2^{(log_2 n)-1}} + \cdots + nlog_2 n$ Assuming $4^{log_2 n} = n^{log_2 4}$ this becomes $T(n) = n^{log_2 4}T(1) + nlog_2 n^{2^{(log_2 n)-1}} + \cdots + nlog_2 n$. If we look at n^2 and $nlog_2 n$ we see that n^2 grows faster meaning $T(n) = O(n^2)$ meaning that this is bottom heavy recursion because the work done at each level is O(n^2) and the levels are O(logn). The total work is O(n^2).\

Problem 1

We must see if this problem is solvable so we look at if we can cover the entire board using an L-shape covering 3 pieces. We can see that solving for ((2^k * 2^k) – 1) % 3 always results in 0. We can now continue to solve the problem.

The first step is to think about the base case which would be to have one square removed and then to fill in the remaining squares.

Now that we have our base case defined, we must make an algorithm that has a reasonable runtime that uses recursion and preferably memorization in a dynamic programming mindset to get to squares that all resemble the base case. Fortunately, the base case can be modeled on a 2**k by 2**k board because the board can be divided in 2x2 squares. That does nothing though without having one square filled to make the L-shape. Therefore, an efficient solution is to divide the 2**k by 2**k into four quadrants of equal size, we can then place a L-shape in the middle and then look inside and divide one of the respective four quadrants into another four quadrants of equal size again until we reach the actual base case of 2x2 with one square filled in the top left of the board. We can then fill the squares respectively and recursively search every pathway through the matrix. We can see in the code that the orientation is rotated respectively, and that the board is properly filled. The time complexity will be $O(n^2)$ because we divided the board into subproblems that each take $O(n^2)$ time to complete resulting in **overall time complexity of $O(n^2)$.**

Algorithm:

1. Remove the top left square in the k**2 by k**2 matrix (if k = 2 then 4x4, k = 3 then 8x8)
2. Divide the board into four quadrants
3. Place an L-shaped piece in the middle of that board for every time you divide the board
4. Recursively solve 2 & 3 until the base case of 2 x 2 is reached with one square filled

```
def fill_bd(bd, i, j, size):
    # base case
    if size == 2:
        # place the pieces in remaining squares
        for x in range(i, i + size):
            for y in range(j, j + size):
                if bd[x][y] != -1:  # left square reached
                    bd[x][y] = 1

        #print matrix at each occurence to see it filling correctly
        for squares in bd:
            for square in squares:
                print(square, end=' ')
            print() # new line for each row
        print()  # new line for each matrix

        return #end recursion instance

    #put L-shaped piece in the middle of the four quadrants on the board
    bd[i + size // 2][j + size // 2] = 1
    bd[i + size // 2 - 1][j + size // 2] = 1
    bd[i + size // 2 - 1][j + size // 2 - 1] = 1
```

```
    #print matrix at each occurence to see it filling correctly
    for squares in bd:
        for square in squares:
            print(square, end=' ')
        print()
    print()

    #divide the board into four quadrants and put an L-shaped piece in the middle of
the board and then split again
    #recursively solving the four quadrant sub problems until we reach the base case
2x2 matrix
    fill_bd(bd, i, j, size // 2)
    fill_bd(bd, i + size // 2, j, size // 2)
    fill_bd(bd, i, j + size // 2, size // 2)
    fill_bd(bd, i + size // 2, j + size // 2, size // 2)

k = int(input('enter k value: '))
bd = [[0 for i in range(2**k)] for j in range(2**k)]

# remove the top left corner square
bd[0][0] = -1
fill_bd(bd, 0, 0, 2**k)
```

Output for K = 2 and K = 3, final output of k = 5:

joeyn256@Josephs-MacBook-Pro CSE 3500 % /usr/local/bin/python3 "/Users/joeyn256/CSE 3500/Chess_Algorithm.py"
enter k value: 2

-1 0 0 0	-1 1 0 0	-1 1 0 0
0 1 1 0	1 1 1 0	1 1 1 0
0 0 1 0	0 0 1 0	1 1 1 0
0 0 0 0	0 0 0 0	1 1 0 0

-1 1 1 1	-1 1 1 1
1 1 1 1	1 1 1 1
1 1 1 0	1 1 1 1
1 1 0 0	1 1 1 1

joeyn256@Josephs-MacBook-Pro CSE 3500 % /usr/local/bin/python3 "/Users/joeyn256/CSE 3500/Chess_Algorithm.py"
enter k value: 3

-1 0 0 0 0 0 0 0	-1 0 0 0 0 0 0 0	-1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0	0 1 1 0 0 0 0 0	1 1 1 0 0 0 0 0

0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

0 0 1 0 0 0 0 0
0 0 0 1 1 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

0 0 1 0 0 0 0 0
0 0 0 1 1 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

-1 1 0 0 0 0 0 0
1 1 1 0 0 0 0 0
1 1 1 0 0 0 0 0
1 1 0 1 1 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

-1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 0 0 0 0 0
1 1 0 1 1 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

-1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 1 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

-1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 1 0 0 0
1 1 0 0 1 0 0 0
1 1 1 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0

-1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 1 0 0 0
1 1 0 0 1 0 0 0
1 1 1 0 0 0 0 0
1 1 1 0 0 0 0 0
1 1 0 0 0 0 0 0

-1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 1 0 0 0
1 1 1 1 1 0 0 0
1 1 1 1 0 0 0 0
1 1 1 0 0 0 0 0
1 1 0 0 0 0 0 0

-1 1 1 1 0 0 0 0
1 1 1 1 0 1 1 0
1 1 1 1 0 0 1 0
1 1 1 1 1 0 0 0
1 1 1 1 1 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0

-1 1 1 1 1 1 0 0
1 1 1 1 1 1 1 0
1 1 1 1 0 0 1 0
1 1 1 1 1 0 0 0
1 1 1 1 1 0 0 0
1 1 1 1 0 0 0 0

-1 1 1 1 1 1 0 0
1 1 1 1 1 1 1 0
1 1 1 1 1 1 1 0
1 1 1 1 1 1 0 0
1 1 1 1 1 0 0 0
1 1 1 1 0 0 0 0

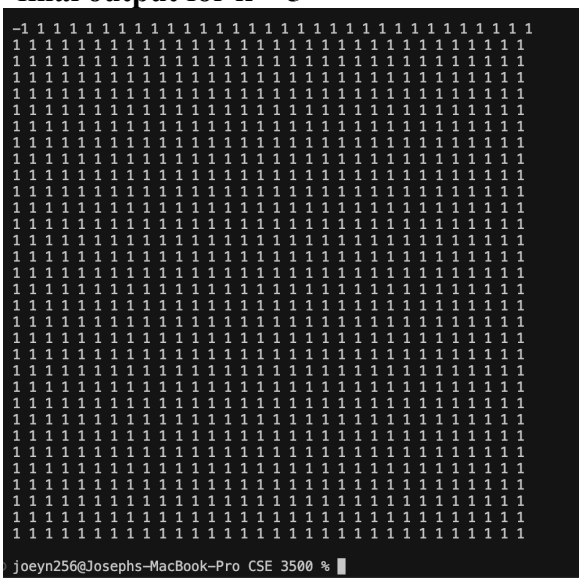
-1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0
1 1 1 1 1 1 0 0
1 1 1 1 1 0 0 0
1 1 1 1 0 0 0 0

1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 0

-1 1 1 1 1 1 1 1	-1 1 1 1 1 1 1 1	-1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1
1 1 1 1 1 0 0 0	1 1 1 1 1 0 0 0	1 1 1 1 1 1 0 0
1 1 1 1 0 0 0 0	1 1 1 1 0 1 1 0	1 1 1 1 1 1 1 0
1 1 1 1 0 0 0 0	1 1 1 1 0 0 1 0	1 1 1 1 0 0 1 0
1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 0	1 1 1 1 0 0 0 0

-1 1 1 1 1 1 1 1	-1 1 1 1 1 1 1 1	-1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 0	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0	1 1 1 1 1 1 1 0	1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 0	1 1 1 1 1 1 0 0	1 1 1 1 1 1 1 1

final output for k = 5



Problem 2:

Merge sort can be used to solve this problem in O(nlogn) time. We first divide the original list into two sublists for the first layer. We continue to divide until we reach the last layer with

sublists of size 1. We then merge the sublists of size 1 together in order and count the number of inverted pairs per merge. Every merge creates a list of 2*the previous size which can count the number of inverted pairs. We continue until we have merged all the sublists together back to a sorted list. The algorithm takes O(n) runtime to split the list into sublists of size 1 and then merging the list back together in order takes O(nlogn) so the overall time complexity is **O(nlogn).**

The algorithm:

1. Divide the list into sublists until we reach sublists of size 1.
2. Merge the respective sublists together in order.
3. Count the number of inverted pairs per merge in each list
4. Repeat 2 & 3 until we have merged the sorted list back together

Problem 3

1. To devise an O(n) algorithm, we begin by dividing the list into two sublists. Next, using pointers, we find the maximum value in each sublist. While traversing each sublist, we keep track of the maximum value and its index. By summing the maximum values from both sublists, we determine the maximum value for the combined sublists, focusing on the middle of the list. If we encounter a value greater than the current maximum for the combined sublists, we update it. This iterative process continues until we've examined every element in the list, ensuring a time complexity of O(n) for finding the maximum combined value of the sublists.

```
2. def BSM(nums):
3.     n = len(n)
4.
5.     # initialize pointers for the two sublists
6.     left_pointer = n // 2
7.     right_pointer = n // 2 + 1 if n % 2 == 0 else n // 2
8.
9.     # initialize max value of the pointers
10.    max_left_value = nums[left_pointer]
11.    max_right_value = nums[right_pointer]
```

```
12.
13.     # calculate the max value for the first sublist
14.     while left_pointer >= 0:
15.         if nums[left_pointer] > max_left_value:
16.             max_left_value = nums[left_pointer]
17.         left_pointer -= 1
18.
19.     # calculate the max value for the right sublist
20.     while right_pointer < n:
21.         if nums[right_pointer] > max_right_value:
22.             max_right_value = nums[right_pointer]
23.         right_pointer += 1
24.
25.     # sum the max values of the left and right sublists
26.     max_sum = max_left_value + max_right_value
27.
28.     return max_sum
```

2. We can develop a recursive algorithm to address the best subset problem with a runtime of O(nlogn) by using our BSM function in variation of mergesort. Given that we've already established a O(n) algorithm for the BSM problem, we can utilize it by continuously splitting the list into two sublists until each sublist contains only one element. We then merge these sublists together in ascending order and during each merge, we identify the max value within the resulting sub list storing it. As we continue merging sublists, we update the maximum value if a higher one is encountered. This process continues until all sublists are merged. Consequently, we achieve a runtime of O(nlogn) for finding the maximum value of the sublist. This is because iterating through the indivual list is O(n) runtime and subsequently merging the sublists in order takes **O(nlogn) time.**

3. We can demonstrate both the correctness and the O(nlogn) runtime of our algorithm. To prove its correctness we prove how common our program is to mergesort. The algorithm splits the problem into subproblems, each handling smaller lists of size 1 like mergesort. Consequently, there are O(logn) subproblems due to this recursive division. At each level of recursion, O(n) operations are executed which results in a total of O(nlogn). Likewise, it is common knowledge that mergesort runs in O(nlogn) runtime due to this method. Therefore, we have establish edits runtime as O(nlogn), and below is working code to prove the algorithm:

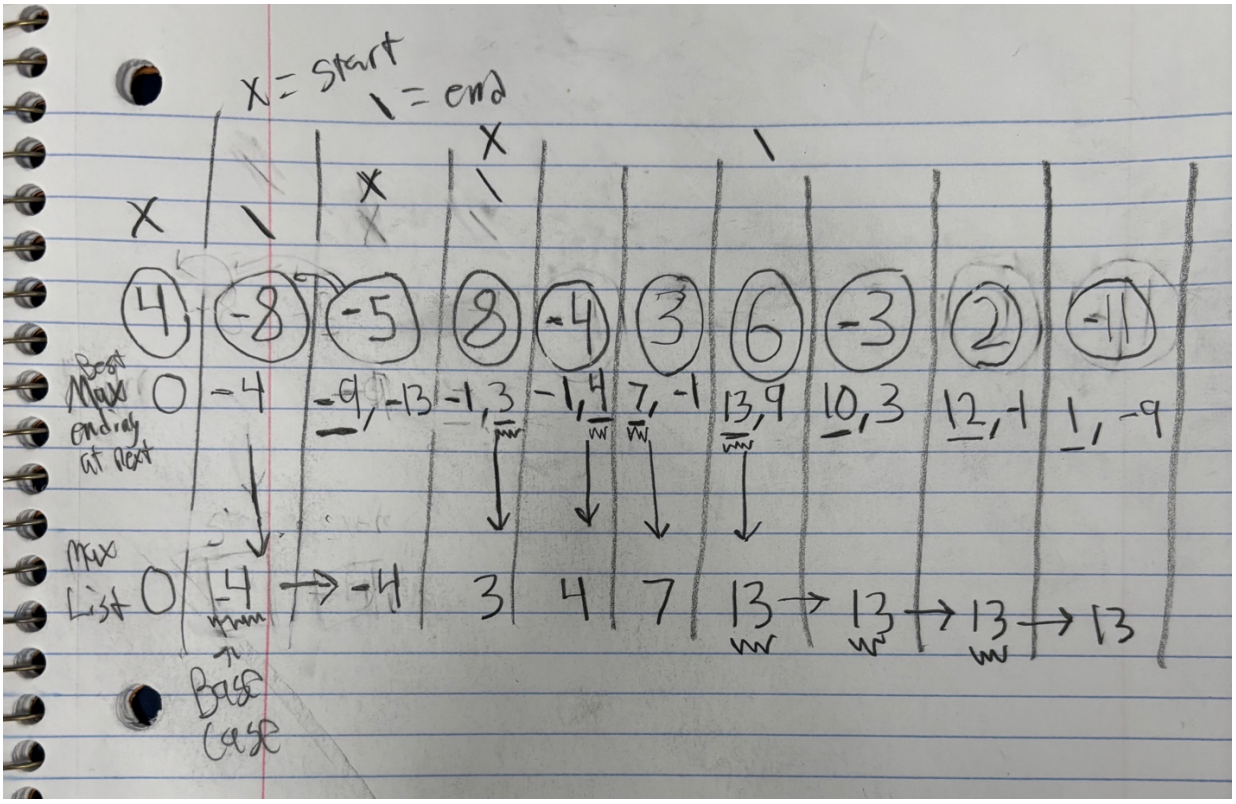
```
def BSM(nums):
    n = len(nums)

    # base case: if the list has only one element, return it
    if n == 1:
        return nums[0]
```

```
    # split the list into two sublists
    mid = n // 2
    left_sublist = nums[:mid]
    right_sublist = nums[mid:]

    # recursive calls to BSM on the left and right sublists
    max_left = BSM(left_sublist)
    max_right = BSM(right_sublist)

    # find the maximum sum across the middle
    max_left_middle = max_left_right = 0
    current_sum = 0
    for i in range(mid - 1, -1, -1):
        current_sum += nums[i]
        max_left_middle = max(max_left_middle, current_sum) # compute from middle to
start

    current_sum = 0
    for i in range(mid, n):
        current_sum += nums[i]
        max_left_right = max(max_left_right, current_sum) # computer from middle to
end

    max_middle = max_left_middle + max_left_right # this will take the max of the lsit
iterating from the left to the right through the middle of the list

    # the max of the three sums found recursively will be the largest sublist of the
values
    return max(max_left, max_right, max_middle)

#test using example given
nums = [4, -8, -5, 8, -4, 3, 6, -3, 2, -11]
result = BSM(nums)
print("max value of the sublist:", result)
```

Result:

```
joeyn256@Josephs-MacBook-Pro CSE 3500 % /usr/local/bin/python3 "/Users/joeyn256/CSE 3500/
Problem3.py"
Maximum value of the sublist: 13
```

4.



We can solve this problem by keeping track of two counters while iterating through the list only once to get O(n) runtime. We start by creating our base case which is the sum of the first two and store it as a variable in **"best max ending at next"** and **"max list"**. We keep track of a variable of **"best max ending at next"** which uses left value of the pairs on the diagram which is determined by using the last **"best max ending at next"** value plus the next number in the list, and the right value of the pair is the last number in the list plus the new number in the list. **"best max ending at next"** takes the max of these two values and stores the new value represented by the **line underneath** in the diagram. The other variable is the **"max list"** which keeps track of the prior **"best max"** and compares it to the **new "best max ending at next"** storing the max of these two variables which is shown in the diagram as the **scribble line and arrow.** The start and ends of the best list kept track of is shown above the list and is a result on updating **"max_list"**. Since this program only runs the list once performing two O(1) max checks, we get a **total runtime of O(n).**

```
def find_max_sublist(nums):
    # edge case
    if len(nums) < 2:
        return sum(nums), nums

    max_ending_at_next = max_list = nums[0] + nums[1] #base case
    start = end = 0

    for i in range(2, len(nums)):
```

```
        max_ending_at_next = max(nums[i], max_ending_at_next + nums[i]) #keep counter
of ending at next as described in the algorithm
        max_list = max(max_list, max_ending_at_next) #keep running counter of the max
list as described in the algorithm,

        if max_list == max_ending_at_next: #update the new end of the running list by
the update of max_list
            end = i

        if max_list == nums[i]: #update the new start of a list by the update of
max_list
            start = i
            end = i

    return max_list, nums[start:end+1]
```