HW2

Problem 1

(1) O(n) is the answer:

Assigning 0 to x and assigning 0 to y takes constant time: c_1 and c_2, respectively. The body of the for loop is executed n - 1 times (minus 1 because i starts at 1); a single execution takes a constant amount of time c_3. And 'return (x - y)' is executed once and takes a constant amount of time c_4. So, the worst-case runtime can be determined: $f(n) = c_1 + c_2 + (n - 1) * c_3 + c_4$ And the function f(n) can be rewritten as $f(n) = (n - 1) * c_3 + c_5$ And if the lower order term is discarded and the coefficient ignored, it is O(n).

(2) $O(n^3)$ is the answer:

In this case it is easiest to first determine the running time of the three while loops. The first while loop is executed n times (i = 0 to < n; and at the end of the first while body i is increased by one – each iteration). The running time of the second and third while loops are easiest to explain together. The second while loop, like the first, is executed n times (j = 0 to < n; and at the end of the second while body j is increased by one – each iteration). The third while loop is in the body of the second while loop. If the second while loop is executed for the first time (j = 0), then the third while loop is executed once. If the second while loop is executed the second time (j = 1), then the third while loop is executed twice. And so on. If the second while loop is executed the last time (j = n -1), then the third while loop is executed n times (k <= j). In other words, the third while loop is executed in the worst case: f(n) = 1 + 2 + … + (n - 2) + (n - 1) + (n) times, which can be rewritten as f(n) = n * (n + 1) / 2.

Since the constants are dropped*, the following running time is obtained:

f(n) = running time first while loop * running time second while loop * running time third while loop

gives f(n) = n * n * (n + 1) / 2. Gives f(n) = (1 / 2) * (n^3 + n^2). So, it is $O(n^3)$.

*: for a single execution, each of the following code parts has a constant running time:

- Assigning 0 to x and assigning 0 to i takes constant time, respectively
- Assigning 0 to y and assigning 0 to j, respectively
- Assigning 0 to k
- Getting a[k], calculating y + a[k], and assigning to y, respectively
- Incrementing k, j, and i by 1, respectively
- Execution of if block

(3) O(n) is the answer:

The recursive method is initially called with i = n - 1. The method body of the recursive method consists of an if and else block. Checking the if condition i == 0 takes constant running time. And executing the if body has a constant running time too (accessing a[0], assigning to p[0], accessing a[0], and assigning to p[1] have each a constant running time). The if body is only executed once for i == 0, otherwise the else body is executed. At the beginning of the else body, the function calls itself recursively - with the parameter i decreased by one. That means, the recursive method calls itself n

times: method3(…, n-1,…), then method3(…, n-2,…), …, then method3(…, 1,…), and finally method3(…, 0,…). Thus, the else body is executed n - 1 times and the if body once. Since all lines of code in the else body have a constant running time (accessing a[i], accessing p[0], comparing: a[i] < p[0] and so on), the result is a running time of

f(n) = 1 * running time if body + (n - 1) * running time else body, which gives f(n) = c_1 + (n - 1) * c_2, which gives O(n).

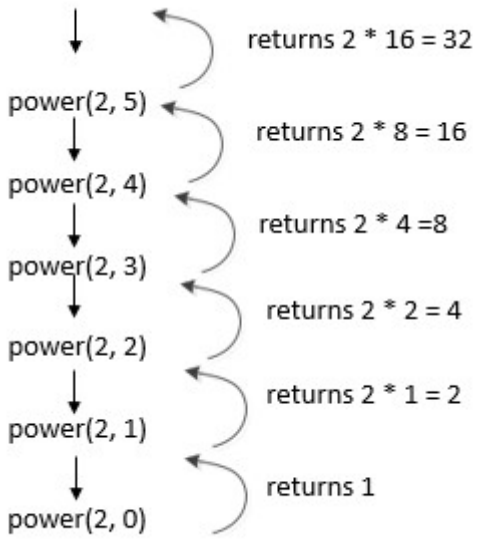
(4) O(n) is the answer:

For the initial call of the recursive method, in the worst case, the condition of the if block evaluates to false (the entire if block requires constant running time) and then the else block is executed. The integer division and assignment as well as the if else block in the else block require constant running time.

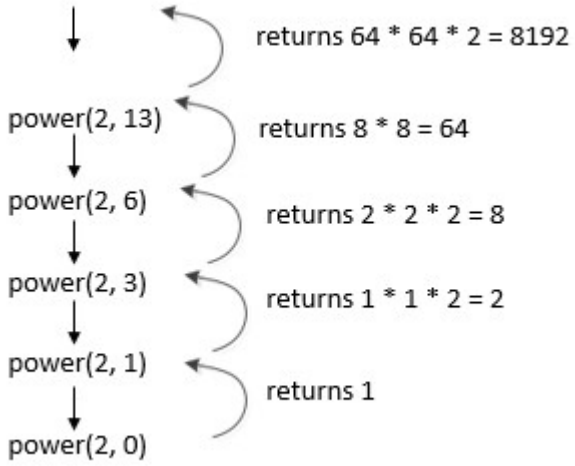
Now it has to be considered how many function calls are done. At the first level of the recursion 2 calls are done, which can be written as 2^1 calls (for method4(a, x, z) and method4(a, z+1, y)). At the second level of the recursion each of these methods does two more calls – this makes 2^2 calls in total. And so on. Thus, at level j of the recursion 2^j function calls are made. Now it has to be answered how many levels j there are. Initially (initial method call), the number of elements to be searched per call is n. In the first recursive invocation per method, the number of elements to be searched is at most n/2. In the second recursive invocation, the number of elements to be searched is at most $n/2^2$. In the third recursive invocation, the number of elements to be searched is at most $n/2^3$. And so on. So, in the j-th invocation, at most $n/2^j$ are searched. This can be expressed as $n/2^j < 1$, which can be rewritten as $n < 2^j$. Taking $\log_2$ of both sides gives $\log_2 n < j$. This turns 2^j into $2^{\log n}$, it is still log base 2, which gives n. And since the rest of the operations in each recursive call are constant: O(n).

Problem 2

(1) Recursion trace:



(2) Recursion trace:



Problem 3

(1) Stack

Operation	Return Value	Stack Contents
push(10)	-	(10)
pop( )	10	()
push(12)	-	(12)
push(20)	-	(12, 20)
size( )	2	(12, 20)
push(7)	-	(12, 20, 7)
pop( )	7	(12, 20)
top( )	20	(12, 20)
pop( )	20	(12)
pop( )	12	()
push(35)	-	(35)
isEmpty( )	false	(35)

(2) Queue

Operation	Return Value	Queue Contents (first ← Q ← last)
enqueue(7)	-	(7)
dequeue( )	7	()
enqueue(15)	-	(15)
enqueue(3)	-	(15, 3)
first( )	15	(15, 3)
dequeue( )	15	(3)
dequeue( )	3	()
first( )	null	()
enqueue(11)	-	(11)
dequeue( )	11	()
isEmpty( )	true	()
enqueue(5)	-	(5)