Solutions

Part a.

if we assume sum++ takes the majority of the work then 1 work is being done each iteration of the while loop

i value	Iteration number	i(rewritten)	work done
n	1	$n/2^0$	1
$n/2$	2	$n/2^1$	1
$n/4$	3	$n/2^2$	1
...	...	...	...
2	$k-1$	$n/2^{k-1}$	1
1	k	$n/2^k$	1

In this above table we use K to represent the number of iterations

we know it terminates when $n/2^k <= 0$ using integer rounding we know $n/2^k = 0$ when $2^k > n$ so:

$$2^k > n$$

using properties of logs we get

$$k > log_2(n)$$

becuase we know k is an integer, and the minimum to satisfy the above equation we know

$$k - 1 = log_2(n)$$

$$k = log_2(n) + 1$$

We know there are $log_2(n) + 1$ iterations and 1 work done each iteration so total work is $1 * (log_2(n) + 1) = log_2(n) + 1$ so our runtime is

$$O(log(n))$$

(ps. a more exact run time is $floor(log_2(n) + 1)$)

Part b.

if we assume sum++ takes the majority of the work then 1 work is being done each iteration of the inner for loop

i value	work done
0	1
1	1
2	1
...	...
$n-2$	1
$n-1$	1

so work for inner for loop is $n - 1 - 0 + 1 = n$ (the plus one is from 0 to 0 is still 1 loop) the outer while look is like part a.

j value	Iteration number	j(rewritten)	work done
N	1	$N/2^0$	1
$N/2$	2	$N/2^1$	1
$N/4$	3	$N/2^2$	1
...	...	...	...
2	$k-1$	$N/2^{k-1}$	1
1	k	$N/2^k$	1

here we still use K to be number of iterations. useing the math from part 1 we can skip to:

$$k = log_2(N) + 1$$

we know that the inner loops work is always n so total work is $n * k$ so total work is $n * (log_2(N) + 1) = n * log_2(N) + N$ so order is:

$$O(n * log_2(N))$$

(ps. a more exact run time is $n * floor(log_2(N)) + n$)

Part c.

We will also assume " O(N/2)" means exactly N/2 (This will not affect the end bigO complexity)

The inner for loops runs as shown:

K value	work done
0	$N/2$
1	$N/2$
2	$N/2$
...	...
$n-2$	$N/2$
$n-1$	$N/2$

so we have n iterations of $N/2$ work so total work is $nN/2$

we will use n_0 to mean the initial n value for the outerloop we have

n value	Iteration number	n(rewritten)	work done
n_0	1	$n_0/2^0$	$N/2 * n_0/2^0$
$n_0/2$	2	$n_0/2^1$	$N/2 * n_0/2^1$
$n_0/4$	3	$n_0/2^2$	$N/2 * n_0/2^2$
...	...	...	...
2	$k-1$	$n_0/2^{k-1}$	$N/2 * n_0/2^{k-1}$
1	k	$n_0/2^k$	$N/2 * n_0/2^k$

There are 2 ways to Interpret this, if we take N and n to be 2 parameters we get

$$N/2 * n_0/2^0 + N/2 * n_0/2^1 + N/2 * n_0/2^2 + ...$$
$$= N/2(n_0/2^0 + n_0/2^1 + n_0/2^2 + ...)$$
$$= N * n_0/2(1/2^0 + 1/2^1 + 1/2^2 + ...)$$
$$= N * n_0/2(1/(1 - 1/2))$$
$$= N * n_0/2 * 2$$
$$= N * n_0$$

so we get the bigO to be:

$$O(n * N)$$

However if we Assume n and N are the same thing,and N is updated with n we get:

$$n_0/2^0/2 * n_0/2^0 + n_0/2^0/2 * n_0/2^1 + n_0/2^0/2 * n_0/2^2 + ...$$
$$= 1/2 * (n_0^2/4^0 + n_0^2/4^1 + n_0^2/4^2 + ...)$$
$$= n_0^2/2 * (1/4^0 + 1/4^1 + 1/4^2 + ...)$$
$$= n_0^2/2(1/(1 - 1/4))$$
$$= N * n_0/2 * 4/3$$
$$= 2n_0^2/3$$

so we get the order to be:

$$O(n^2)$$

Part d

for ease of reading we will refer to each loop by what indexes it, that is the inner loop is loop-k, the middle is loop-j and the outer is loop-i we will also assume(rightfully so) that the system.out.println takes by far the most time

loop k work time:

k value	work done
n	1
$n-1$	1
$n-2$	1
...	...
$j+2$	1
$j+1$	1

so the work in this loop is $n - (j + 1) + 1 = n - j$

loop j work time:

j value	work done
$i+2$	$n-(i+2)$
$i+1$	$n-(i+1)$

so the work in this look is $n-(i+2)+n-(i+1)=2n-2i-3$

loop i work time:

i value	work done
0	$2n-3$
1	$2n-2-3$
2	$2n-4-3$
...	...
$n-4$	$2n-2*(n-4)-3$
$n-3$	$2n-2*(n-3)-3$
$n-2$	$n-(n-2+1)^*$
$n-1$	0^*

the last 2 loops differ because loop-j activates loop-k once on n-2 and not at all on n-1

so if we sum the work done:

$$(2n-0-3)+(2n-2-3)+(2n-4-3)+...+(2n-2*(n-3)-3)+(n-(n-2+1))$$

we know we have n-2 full iterations (the last 2 are special) so we multiply by that when we pull out constants:

$$=(n-2)(2n-3)-(0+2+4+...+2*(n-3))+n-(n-1)$$

$$=(2n^2-7n+6)-2(0+1+2+...n-3)+1$$

$$=(2n^2-7n+7)-2((n-3)(n-2)/2)$$

$$=(2n^2-7n+7)-(n^2-5n+6)$$

$$=n^2-2n+1$$

$$=(n-1)^2$$

so we get bigO:

$$O(n^2)$$

(ps. a more exact run time really is $(n-1)^2$)

Part e.

There will later be a picture that makes this more readable

we will also start with:

lemma 1:$1+2+4+8+...+2^k=2^{k+1}-1$

this should be apparent as written in binary this sum would produce k ones in a row, adding one to this would cascade causing it to be one followed by k zeros, which has a value of 2^{k+1} so:

$$(1+2+4+8+...+2^k)+1=2^{k+1}$$

$$1+2+4+8+...+2^k=2^{k+1}-1$$

we can also treat this method as a recurrence:

$$aMethod(1)=0$$

$$aMethod(n)=1+aMethod(n/2)+aMethod(n/2)$$

we simplify the recurrence to

$$aMethod(n) = 1 + 2 * aMethod(n/2)$$

we can unwind the recurrence:

$$aMethod(n) = 1 + 2(1 + 2 * aMethod(n/4))$$

$$aMethod(n) = 1 + 2(1 + 2 * (1 + 2 * aMethod(n/8)))$$

a bit of algebra and a pattern immerges

$$aMethod(n) = 1 + 2(1 + 2 * (1 + 2 * aMethod(n/8)))$$

$$aMethod(n) = 1 + 2(1 + 2 + 4 * aMethod(n/8))$$

$$aMethod(n) = 1 + 2 + 4 + 8 * aMethod(n/8)$$

we also know the 8 in $n/8$ came from $n/2/2/2$ or$n/2^3$ so we know

$$aMethod(n) = \sum_{i=0}^{3-1}(2^i) + 2^3 * aMethod(n/2^3)$$

unwinding once more would have yielded:

$$aMethod(n) = \sum_{i=0}^{4-1}(2^i) + 2^4 * aMethod(n/2^4)$$

and in general unrolling k times (so long as we don't unroll past aMethod's base case) yields:

$$aMethod(n) = \sum_{i=0}^{k-1}(2^i) + 2^k * aMethod(n/2^k)$$

we know amethod keeps this behaviour until it's called with n<=1 which would happen when $2^k > n$ if we pick k to be the smallest integer this is valid for we get

$$aMethod(n) = \sum_{i=0}^{k-1}(2^i) + 2^k * 0$$

$$aMethod(n) = \sum_{i=0}^{k-1}(2^i)$$

because K is the smallest integer such that $2^k > n$ we also know

$$2^{k-1} <= n$$

$$k - 1 <= log_2(n)$$

so we know

$$aMethod(n) = \sum_{i=0}^{log_2(n)}(2^i)$$

with lemma 1

$$aMethod(n) = 2^{log_2(n)+1} - 1$$

$$aMethod(n) = 2n - 1$$

so the bigO is:

$$O(n)$$

(ps that $log_2(n)$ is really $floor(log_2(n))$ so a more exact runtime the next strictly greater power of 2 -1. ie: 7->7, 8->15, 65->127)