Part 1

Stack Frame

<table>
<tr><td>%rbp</td><td>Saved base pointer</td></tr>
<tr><td>-1(%rbp)</td><td>Return value (8-bit)</td></tr>
<tr><td>-2(%rbp)</td><td></td></tr>
<tr><td>-3(%rbp)</td><td></td></tr>
<tr><td>-4(%rbp)</td><td></td></tr>
<tr><td>-5(%rbp)</td><td rowspan="4">Current (32-bit)</td></tr>
<tr><td>-6(%rbp)</td></tr>
<tr><td>-7(%rbp)</td></tr>
<tr><td>-8(%rbp)</td></tr>
<tr><td>-9(%rbp)</td><td></td></tr>
<tr><td>-10(%rbp)</td><td></td></tr>
<tr><td>-11(%rbp)</td><td></td></tr>
<tr><td>-12(%rbp)</td><td></td></tr>
<tr><td>-13(%rbp)</td><td></td></tr>
<tr><td>-14(%rbp)</td><td></td></tr>
<tr><td>%rsp</td><td></td></tr>
</table>

All other values are initially accessed through the argument registers (except for array, which is located in the stack and explained in part 2), and assigned variable names which are then used to access those values.

The reason for the number of unused bytes is that x86-64 requires stack frames to be a multiple of 16 bytes, even though the function does not necessarily need that much memory.

The return value is a Boolean so only 8 bits are used to store it, while current is an integer so 32 bits are used instead.

Variables other than current are intialised in badrandom and referenced by name.

Recursion here is inefficient in terms of memory because a new stack frame is created with every recursive call, reserving another 16 bytes of memory each time.

Also note that %r14, %r15 and %rbx do not need to be saved in the unoptimised code because they are never used.

Part 2

Stack frame

%rbp	Saved base pointer
-1(%rbp)	
-2(%rbp)	
-3(%rbp)	
-4(%rbp)	
-5(%rbp)	
-6(%rbp)	
-7(%rbp)	Saved value of %r15 (64-bit)
-8(%rbp)	
-9(%rbp)	
-10(%rbp)	
-11(%rbp)	
-12(%rbp)	
-13(%rbp)	
-14(%rbp)	
-15(%rbp)	Saved value of %r14 (64-bit)
-16(%rbp)	
-17(%rbp)	
-18(%rbp)	
-19(%rbp)	
-20(%rbp)	
-21(%rbp)	
-22(%rbp)	
-23(%rbp)	Saved value of %rbx
-24(%rbp)	

The optimised assembly code never explicitly reserves memory space for the stack frame. Instead, the only space taken up on the stack is by pushing callee-saved values in indexes %r14, %r15 and %rbx, as well as for saving the previous base pointer. This is to follow the function call convention of preserving caller-owned registers if the callee wants to use them. The other values, which are all initialised in badrandom, are mostly stored in dedicated registers. The exception is the array pointer, which is in the stack above the base pointer for this function, and its address in the caller's stack frame is accessed using 16(%rbp) when needed (there are only 6 dedicated argument registers, and *in_array is the 7th argument to badrandom).

Start/current (32-bit)	%edi
Mult/in_mult	%esi
Add/in_add	%edx
Div/in_div	%ecx
Min/in_min	%r8d
*in_length	%r9
*array/*in_array	16(%rbp)

Optimisations

The variables *array, mult, add, div and min do not take up space in the stack frame. Instead, as they are all intialised to arguments to badrandom, the argument indexes (%edi to %r9) are used instead. While upon first glance this might seem problematic, as single registers are being used to track the values of two separate variables, it works out fine as the arguments are never needed again after being used to initialise the variables. For example, when badrandom_recurse(start) is called, current is set to the same value as start. After this point, the computer no longer uses the start variable, so instead treats the corresponding register (%edi) as storing the value of current. Note that this also eliminates the need for these values to be assigned variable names.

Another optimisation made is that this assembly code avoids recursion. While badrandom_recurse is written recursively in the C code and is treated as such in the unoptimised assembly, it is now treated as an iterative function. The callq instruction is never used to make a recursive call and additional space is never reserved in the stack frame. Instead, the assembly repeatedly jumps back to the .LBB0_3 label every time a recursive call is made, and runs the same operations on the new value of current. On each iteration, it checks the two termination conditions and exits the loop whenever one of them is met.This is why it never makes a call to badrandom_recurse, as it is effectively treating the whole process as a single function with an iterative loop inside of it.

The key difference between the call and jump instruction is that jump simply moves to a given label, while call pushes the current location it will return to in the stack before moving the instruction pointer. This means that each call uses up stack space while jumping does not.

This code also simplifies the handling of the if/else recursive call in badrandom_recurse (shown below for reference). The only difference between the if and else blocks is the sign of add. It is positive in the if block and negative in the else block. The unoptimised code compiles the below code into 3 sections, for the condition checking, if block and else block, which seems like a fairly straightforward way of handling it. However, the optimised assembly code recognises that the blocks are doing the exact same thing with slightly different values, and stores both +add and –add in separate registers to be used in the operation depending on what the if condition evaluates to.

```
if (current%min == 0) {
      return badrandom_recurse((current*mult+add)%div); }
   else {
      return badrandom_recurse((current*mult-add)%div);
```

```
    }
```