3.2*b* For 6 to have been popped, it would have been necessary for 5 to be pushed onto the stack first, before 6. Consequently, a pop following the case where 6 was popped would result in 5 being popped, not 4.

3.2*c* Hint: just update the appropriate pointers and be sure to set the pointers in the argument `list` (which is passed by reference) to `nullptr` and the size to zero.

3.2*d* If the argument was passed by value, a copy would be made and therefore the run time would be $\Theta(n)$, even if we did not observe that in our code.

3.2*f* An undo operation follows that of a stack: it tracks a stack of the web pages that we have previously visited. For the redo operations, we could continue to keep the visited web pages in the stack, but above the top pointer. Now, to go forward (or redo), we just go up, not down, the stack. The only proviso is that we cannot "redo" an operation following a visit by the user. For example, suppose the user visits:

http://uwaterloo.ca
http://ece.uwaterloo.ca/~dwharder
http://www.wikipedia.org
http://www.theskepticsguide.org

So, now the top of the stack points to the Skeptic's Guide to the Universe (you're escape to reality). If the user now goes back in the history three times, the top pointer is now at uwaterloo.ca again. If the user goes forward, we just move up the stack again to the author's web site, but now if we visit a new web site, say skepchick.org, everything above the author's web site is removed and replaced with skepchick.org, yielding:

http://uwaterloo.ca
http://ece.uwaterloo.ca/~dwharder
http://skepchick.org

3.2*h* The result would be:

```
template <typename Type>
void Stack<Type>::push( Type const &obj ) {
    if ( size() == capacity() ) {
        double_capacity();
    }

    array[stack_size] = obj;
    ++stack_size;
}
```

3.2*j* In the first case, note that 3/3, 4/3 and 5/3 all equal 1, and yet, some programmers may not realize that `1 >= 5/3` will return true.

In the second case, floating-point division can cause round-off errors. For example, using decimal arithmetic, 1/3 = 0.3333333333···, but floating-point numbers only record a finite number of digits (say, 10). When we multiply the result again by 3, we get 0.9999999999.

In the third case, we do not have issues with either integer division or floating-point division.

2.2*l* The number of copies made per insertion will be

$$\frac{\sum_{k=1}^{\left\lfloor \frac{n}{13} \right\rfloor} 13k}{n} = \frac{13\left\lfloor \frac{n}{13} \right\rfloor \left\lfloor \frac{n}{13} + 1 \right\rfloor}{2n}.$$

This is maximized whenever *n* is as small as possible for a given fixed numerator, namely, when *n* is of the form 13*k* + 1, in which case we have

$$\frac{13k(k+1)}{2(13k+1)} \approx \frac{k}{2} + \frac{6}{13} + \cdots = \frac{n-1}{26} + \frac{6}{13} + \cdots,$$

Therefore, *n*/26 is a reasonable estimate as to the number of copies per insertion. The worst-case number of unused memory locations is 12.

3.2*n* The state of the stack is:

```
<xhtml>
<body>
<p>
<i>
```

3.2*p* The state of the stack is

```
{
{
{
(
[
```

(Indenting would make this question too easy…which is probably why you should use proper indentations in your source code.)

3.2*r* The number of function calls is nine:

```
F(4)
        F(3)
                F(2)
                        F(1)
                        F(0)
                F(1)
        F(2)
                F(1)
                F(0)
```

3.2*t* The results are 31, 31 and 35.

3.2*u* `a.t = 3 + 4*sin(s)`