2.1 $2/N$, 37, $\sqrt{N}$, N, $N \log\log N$, $N \log N$, $N \log(N^2)$, $N \log^2 N$, $N^{1.5}$, N^2, $N^2 \log N$, N^3, $2^{N/2}$, 2^N. $N \log N$ and $N \log(N^2)$ grow at the same rate.

2.3 We claim that $N \log N$ is the slower growing function. To see this, suppose otherwise. Then, $N^{\epsilon/\sqrt{\log N}}$ would grow slower than $\log N$. Taking logs of both sides, we find that, under this assumption, $\epsilon/\sqrt{\log N} \log N$ grows slower than $\log \log N$. But the first expression simplifies to $\epsilon\sqrt{\log N}$. If $L = \log N$, then we are claiming that $\epsilon\sqrt{L}$ grows slower than $\log L$, or equivalently, that $\epsilon^2 L$ grows slower than $\log^2 L$. But we know that $\log^2 L = o(L)$, so the original assumption is false, proving the claim.

2.7 For all these programs, the following analysis will agree with a simulation:

(I) The running time is $O(N)$.

(II) The running time is $O(N^2)$.

(III) The running time is $O(N^3)$.

(IV) The running time is $O(N^2)$.

(V) j can be as large as i^2, which could be as large as N^2. k can be as large as j, which is N^2. The running time is thus proportional to $N \cdot N^2 \cdot N^2$, which is $O(N^5)$.

(VI) The *if* statement is executed at most N^3 times, by previous arguments, but it is true only $O(N^2)$ times (because it is true exactly i times for each i). Thus the innermost loop is only executed $O(N^2)$ times. Each time through, it takes $O(j^2) = O(N^2)$ time, for a total of $O(N^4)$. This is an example where multiplying loop sizes can occasionally give an overestimate.

2.12
(a) N=12000000
(b) If they can get to NlogN=log100 * 12000000, give full points
(c) N≈34641
(d) N4932

2.20 (a) Test to see if N is an odd number (or 2) and is not divisible by 3, 5, 7, . . . , $\sqrt{N}$

(b) $O(\sqrt{N})$, assuming that all divisions count for one unit of time

(c) $B = O(\log N)$.

(d) $O(2^{B/2})$.

(e) If a 20-bit number can be tested in time T, then a 40-bit number would require about T^2 time.

(f) B is the better measure because it more accurately represents the *size* of the input.

3.23
(a, b) This function will read in from standard input an infix expression of single lower case characters and the operators, +, −, /, *, ^ and (,), and output a postfix expression.

```
void inToPostfix() {
stack<char> s;
char token;
cin>> token;
while (token != '=') {
if (token >= 'a'   && token <= 'z')
cout<<token<<" ";
else
switch (token) {
```

```
case ')' : while(!s.empty() && s.top() != '(')
{ cout<<s.top()<<" "; s.pop();}
s.pop(); break;
case '(' : s.push(token); break;
case '^' : while(!s.empty() && !(s.top()== '^' ||
s.top() == '('))
{cout<<s.top(); s.pop();}
s.push(token); break;
case '*' :
case '/' : while(!s.empty() && s.top() != '+'
&& s.top() != '-' && s.top() != '(')
{cout<<s.top(); s.pop();}
s.push(token); break;
case '+' :
case '-' : while(!s.empty() && stop() != '(' )
{cout<<s.top()<<" "; s.pop();}
s.push(token); break;
}
cin>> token;
}
while (!s.empty())
{cout<<s.top()<<" "; s.pop();}
cout<<" =\n";
}
```

(c) The function converts postfix to infix with the same restrictions as above.

```
string postToInfix() {
stack<string> s;
string token;
string a, b;
cin>>token;
while (token[0] != '=') {
if (token[0] >= 'a' && token[0] <= 'z')
s.push(token);
else
switch (token[0]) {
case '+' : a = s.top(); s.pop(); b = s.top(); s.pop();
s.push("("+ a+" + " + b+")"); break;
case '-' : a = s.top(); s.pop(); b = s.top(); s.pop();
s.push("("+a+" - "+ b+")"); break;
case '*' : a = s.top(); s.pop(); b = s.top(); s.pop();
s.push("("+a+" * "+ b+")"); break,
case '/' : a = s.top(); s.pop(); b = s.top(); s.pop();
s.push("("+a+" / " + b+")"); break;
case '^' : a = s.top(); s.pop(); b = s.top(); s.pop();
s.push("("+a+" ^ " + b+")"); break;
}
cin>> token;
}
return s.top();
} //Converts postfix to infix
```