1. Suppose the functions $g_1, \ldots, g_7$ are defined as follows: $g_1(n) = \sqrt{n}$, $g_2(n) = 2^n$, $g_3(n) = n^{1/3}$, $g_4(n) = (\log n)^2$, $g_5(n) = n^{\log n}$, $g_6(n) = n/\log n$ and $g_7(n) = (1/3)^n$. Arrange these seven functions in ascending order by their growth rate. In other words, if you place g(n) after f(n) then show that f(n) = O(g(n)). Fully justify your answer! [25 marks]

$f_1(n) = O(\sqrt{n})$ $f_2(n) = O(2^n)$ $f_3(n) = O(n^{1/3})$ $f_4(n) = O((\log n)^2)$

$f^5(n) = O(n^{\log n})$ $f^6(n) = O(n/\log n)$ $f^7(n) = O((1/3)^n)$

Ascending Order by growth rate:
$f_7 >> f_2 >> f_4 >> f_3 >> f_6 >> f_5 >> f_1$

2. Consider the Pesky algorithm, for an integer n ≥ 0.

(a) What is the value returned by the function Pesky? Express your answer as a function of n and give the closed form.
f(0) = 0, because the loop never satisfies in line 2 (int i=1; i<=0; i++)
f(1) = 2, f(5) = 70, f(10) = 440

f(n) = [n(n+1)(n+2)]/3

(b) Using O() notation, give the worst-case running time of the function Pesky.
O((n^3)/3)

3*. Consider the recursive algorithm.
if n ≤ 1 then return n

else
return 5·g(n–1)–6·g(n–2)

end if

(a) Set up a recurrence for this function's values and solve it to determine what this algorithm computes.

g(n) = { n, if n ≤ 1;
{ 5*g(n-1)-6g(n-2) otherwise.

g(1) = 1,
g(2) = 5*g(2-1)-6*g(2-2) = 5*g(1)-6*g(0) = 5*1-6*0 = 5-0 = 5
g(3) = 5*g(3-1)-6*g(3-2) = 5*g(2)-6*g(1) = 5*[5*g(1)-6g(0)]-6 = 5*[5*1-6*0]-6 = 5*5-6 = 19
g(4) = 65

(b) Set up and solve a recurrence relation for the number of multiplications made by this algorithm.

4. Let T(n) be defined recursively as:

$$T(n) = \begin{cases} 4 & \text{if } n=1 \\ T(n-1) + 4 & \text{otherwise} \end{cases}$$

Show by induction that T (n) = 4n.

Base Case:

T(1) = 4 by definition, and 4n=4 when n=1
in conclusion T(n)=4n when n=1

Induction Step:

Assume T(k) = 4k for arbitrary integer k ≥ 1.
Then, the statement will hold true for k+1
By definition T(k + 1) = T(k) + 4.
By the induction hypothesis, T(k) = 4k
Thus T(k+1) = 4k + 4 which is equal to 4(k+1).
Hence, the result remains true for n = k+1 and by proof of induction, it will therefor hold true for all n ≥ 1

5. You're at the base of a staircase consisting of n stairs. If each step you make takes you either one or two stairs higher, what is the number of different ways in which you can climb the staircase? Explain how you derived your mathematical expression in words.

There are two special cases,
f(0)=0, which has 0 combinations of climbing the stairs
f(1)=1, which means that you only have one choice of the number of stairs to step
f(n) can be realized by the following logic:
After stepping one stair, you must proceed in stepping up the remaining n-1 stairs
or, After stepping two stairs, you must proceed in stepping the remaining n-2 stairs.
This results in the total amount of combinations: f(n-1) + f(n-2)
After examination of this problem, it is clear that this *algorithm* mimics that of the fibonacci series.
Because of this, the number of different ways in which you can climb n stairs is as follows: $F_n = f_{n-1} + f_{n-2}$

6. Design a divide-and-conquer algorithm for computing the number of levels in a binary tree. In particular, the algorithm should return 0 and 1 for the empty and single-node trees, respectively. What is the efficiency class of your algorithm?
My implementation, in some-what pseudocode

```
height(T: BinaryTree) {
        if (T==null) {
                return 0;
        }
        else {
                return max(height(tree.left), height(tree.right)) + 1;
        }
}
```

The efficiency of this algorithm is O(n), because each iteration spends a constant time in each node, with each node being visited only once.

7. Given a pristine chocolate bar with n × m pieces making up the bar. You need to break it into nm 1 × 1 pieces to share with nm people. You can break the bar only in a straight line, and once broken, only one piece at a time can be further broken. Design an algorithm that solves the problem with the minimum number of bar breaks. What is this minimum number? Justify your answer by using properties of a binary tree. (Hint: Think divide-and-conquer.)

We can represent the chocolate bar using a binary tree. The parent nodes represent a breakable piece of chocolate, where the leaves are 1x1 pieces which require no further breaking. These are represented by nm.
The number of breakable chocolate pieces is represented by nm-1.

No matter which algorithm is implemented, the minimum number of breaks will always be equal to: n*m - 1 breaks.

8. Read the notes on OpenMP (file OpenMP.pdf) compiling and executing the sample programs included in the zip file (file openmp.zip) as you go to familiarize yourself with the various OpenMP directives. Take screenshots of each activity as you work your way through the exercise to include in your solution, including answers to any questions asked.

. **(a) For the program in the file parfor.cc vary the value of the variable n and the number of threads specified in the num threads clause. How are the iterations distributed among threads? Be sure to try out fewer iterations than threads, and more iterations than threads, etc.**
num_thread(10), n=5: one per, 5 sleeping threads
num_thread(10), n=15: two for first 5 threads, 1 for rest (split *evenly*)
num_thread(3), n=6: two per, as expected

num_thread(3), n=7: two for first, 1 for rest.

Looks like the first threads always take the odd numbers of threads, and the remaining threads take the ideal split of operations.

- **(b) For the program in the file private.cc explore the values of the variables a and i before, after, and inside the parallel region. Try initializing the variables before the #pragma and also not initializing them.**
  The variables are already initialized before the pragma, but I have played with their values to see the outcome changes. The values are unchanged from before to after the for loop, since the for loop is specified to have its own variables.

  When the variables are not initialized, nothing really happens. They are still 0 after the loop.

  When we initialize the variables inside the parallel region, they take over the argument variables a and i, causing the loop to repeat its output with the same number each time, at least in my tests.

- **(c) For the program in the file reduction.cc explore the run time of the reduction clause by varying the number of threads in the num threads clause. (Thinking ahead: Can you use a reduction to implement the parallel sum in the ParallelAverage command in Project #1?)**

  I have edited the program to keep track of runtime. After changing the threads, I am trying a few combinations to see where the advantages of multiple threads are.

num_threads(10)

n=10; time(ms)=0.431

n=50; time(ms)=0.434

n=100; time(ms)=0.432

n=1000; time(ms)=0.547

num_threads(2)

n=10; time(ms)=3.947

n=50; time(ms)=4.248

n=100; time(ms)=4.024

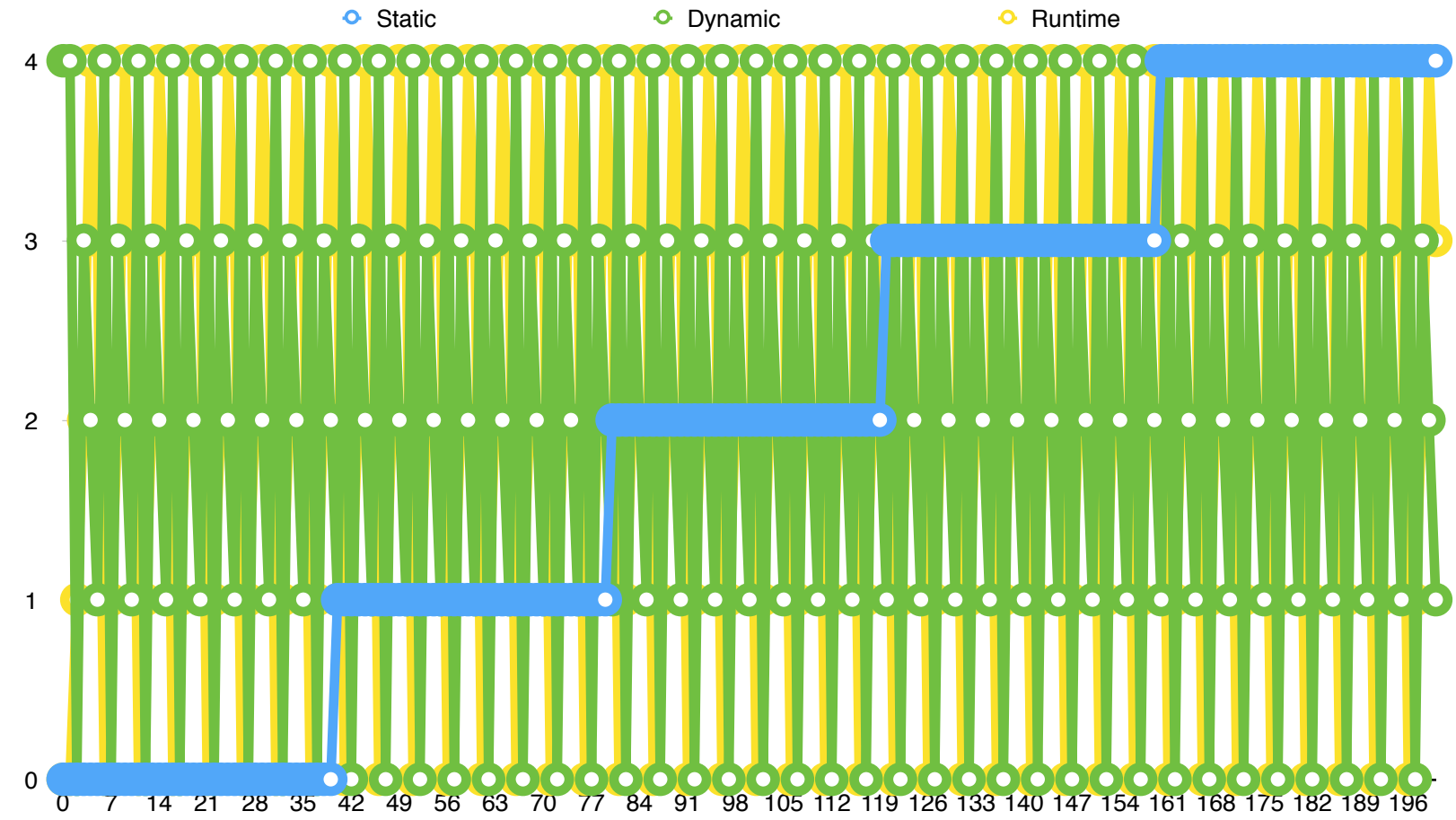
n=1000; time(ms)=4.326

It appears, as I assumed, that the more threads allocated (to an extent) provides better performance; especially with a higher n. It is very hard to test the validity of this, however, because the general.asu.edu server is providing widely different elapsed times for every run. This results in unreliable statistics.

Edited program, including elapsed time implementation

```
#include    <stdio.h>
#include    <stdlib.h>
#include    <omp.h>
#include    <sys/time.h>
#include    <string>
#include    <fstream>
#include    <iostream>

using namespace std;

int main()
{
    double time; // Statistical values used to track algorithm efficency
    struct timeval begin, end; // Struct for gettimeofday call
    int i, n, *a, local_sum, sum, tid;
    printf( "Example of the reduction construct\n" );
    printf( "Give an upper bound on the array a:\n" );
    scanf( "%d", &n );
    printf( "n = %d\n", n);

    if( ( a = (int * ) malloc( n * sizeof(int) ) ) == NULL )
        perror( "Error allocating a\n" );

    for( i = 0; i < n; i++ )
        a[i] = i;

    sum = 0;

    /* The reduction clause gets the OpenMP compiler to generate code that  */
    /* performs the summation in parallel and that the variable sum will    */
    /* the result of a reduction.                                           */
    gettimeofday(&begin, NULL); // Record starting time of insertionsort

#pragma omp parallel for default(none) shared(n,a) reduction(+:sum) num_threads(2)
    for( i = 0; i < n; i++ )
        sum += a[i];
    /* -- End of parallel reduction -- */

    printf( "Sum should be n(n-1)/2 = %d\n", n*(n-1)/2 );
    printf( "Value of sum after parallel region: %d\n", sum );
    free(a);
    gettimeofday(&end, NULL); // Record ending time of insertionsort

    // Sum of elapsed insertionsort time in ms
    time = ((end.tv_sec - begin.tv_sec) +
            ((end.tv_usec - begin.tv_usec)/1000000.0))*1000;
    cout << "Time to run the selection (ms): " << time << '\n';
    return(0);
}
```

. **(d) For the program in the file schedule.cc explore the impact of the schedule kind and chunk size on how chunks are assigned to threads. Investigate the schedules when kind is static, dynamic, and runtime. Specifically, for n=200 iterations and num threads(5) plot a graph for each schedule kind: give the iteration number along the x-axis (1, . . . , 200) and the thread identifier along the y-axis (0, . . . , 4). This graph should indicate which thread was assigned to which loop iteration, graphically illustrating the differences between the kinds of schedules. Explain the results of your graphs. (Hint: To make schedule.cc run faster, put the threads to sleep for less time.)**



This graph took a while to make, but provided a great understanding on how the thread scheduling works.