UNBC Laboratory Assignment
Computer Science 200—Fall 2007

Stopwatches, loops, and graphs

Purpose:

Practice. To create a timing function and use that timing function to plot the time behaviour of various functions.

Due Date:

This assignment is due *Thursday, 27 September, 2006* at the beginning of class.

Hand-in Procedure:

A complete assignment is due at the beginning of class and must contain (a) all of the code used to create the program, (b) `Makefile`'s if any, (c) something showing the output of the compilation process, and (d) test runs.

Assignment:

Stop-watches

The first part of this assignment consists of creating a `StopWatch` class. You are going to need the `StopWatch` class in later assignments, so make sure that you create a separate `StopWatch.h` file, *etc.*

The `StopWatch` class should behave like a mechanical three-button stop-watch. That is, the interface should look something like:

```
class StopWatch
{public:
    StopWatch() ;
    // The three buttons.        // and queries
    StopWatch& start() ;         double elapsed() const ;
    StopWatch& stop() ;          bool is_running() const ;
    StopWatch& reset() ;
private: ...} ;
```

Even though you may not need it for this assignment, your stop-watch should function correctly regardless of the order in which the three buttons are pressed. Your grade depends in part on how well you accomplish this.

There is more than one notion of elapsed time for a multi-process operating system because single processes don't get all of the processor. What you want to measure is the elapsed CPU time, not the elapsed wall-clock time.

The **man**-pages for `clock` and `times` would seem to suggest using the `clock()` function to obtain the elapsed CPU-time. Unfortunately, this only gives you resolution of about 10 ms, which is much too long for many timing situations in this course.

There is another function in the standard C library that is much more appropriate: `gethrvtime`. This function is declared in the old-style `<sys/time.h>` header. For more information see `man -s3c gethrvtime` and look at `/usr/include/sys/time.h`.

⇒ Write and test your stop-watch program. Attempt to determine the granularity of the system clock. `gethrvtime` always returns an answer that is measured in nanoseconds, but on many systems the virtual process timer is not accurate to ±1 ns.

⇒ Use your stop-watch to determine the time behaviour of the following functions, and plot your results on graph paper. Make sure that each of your graphs has a title, and labels, scale, and units for each axis.

Functions to plot

- The time taken by `new` to perform n allocations of integers on the heap.
- The time taken by `new` to perform n allocations of integer arrays, each of size n.
- The time taken for the first Bell algorithm described below as a function of n. You should be able to compute a $\Theta(n)$ estimate for this algorithm.
- The time taken for the second (recursive) Bell algorithm described below as a function of n.

More functions to plot — the Bell numbers The Bell numbers are a mathematical series important to Computer Science that start out 1, 1, 2, 5, ..., and grow rapidly. They are defined by the equations

$$B_0 = 1 \tag{1}$$

$$B_n = \sum_{i=0}^{n-1} B_i B_{n-1-i} \qquad \text{for } n > 0. \tag{2}$$

The first Bell algorithm should use a vector, and look something like the code shown in Figure 1.

The second function should use recursion, and look something like the code shown in Figure 2.

Write nicely documented, clean, organized code!

Figure 1: Loop bell algorithm

```
#include <vector>

int bell(int n)
{
    std::vector<int> answers(n+1) ;
    answers[0] = 1 ;
    for(int =1;i<=n;++i)
        {
        // compute answers[i] ....
        }
    return answers[n] ;
}
```

Figure 2: Recursive bell algorithm

```
int bell(int n)
{
    if (0==n)
        return 1 ;
    else
        {
        int answer = 0 ;
        for(int i=0;i<n;++i)
            {
            answer += bell(i)*bell(n-1-i) ;
            }
        return answer ;
        }
}
```