Basic Algorithms — Fall 2018 — Problem Set 1
Due: Thursday, Sept. 27, 3pm

1. **Rates of growth.** For each pair of functions, $f(n)$ and $g(n)$, place an "X" in the appropriate box indicating if $f = O(g)$ or $g = O(f)$. For each pair, you may mark one, both, or none of the boxes.

$f(n)$	$g(n)$	$f = O(g)$	$g = O(f)$
$n(\log_2 n)^2$	$n^2(\log_2 n)$		
n^2	$n \log_2 n$		
$n(\log_2 n)^4$	$n^{1.2}$		
$200n^2 + n^{1.5}$	$n^2/500$		
$\log_7 n$	$\log_5 n$		
$n/\log_2 n$	$\sqrt{n}\log_2 n$		
5^n	7^n		
7^n	$5^{(n^2)}$		
$n!$	$(n+1)!$		
$\sqrt{n}$	$(1+(-1)^n)n$		

2. **Estimating sums by integrals.** Using the method of estimating a sum by an integral, show that

   (a) $\sum_{i=1}^{n} \ln(i) = n\ln(n) + O(n)$

   (b) $\sum_{i=1}^{n} i\ln(i) = \frac{1}{2}n^2\ln(n) + O(n^2)$

   Note: you can use an online calculator (such as Wolfram Alpha) to compute an indefinite integral.

3. **Limit ratio test.** Use the limit ratio test to show that the infinite series $\sum_{i=1}^{\infty} i^2/2^i$ converges.

4. **Integral test.** For an infinite sum $\sum_{i=k}^{\infty} a_i$, we can test for convergence by estimating the finite sum $S_\ell := \sum_{i=k}^{\ell} a_i$ by an integral, and use this estimate to determine if the limit $\lim_{\ell\to\infty} S_\ell$ is finite or infinite. Use this technique to determine the convergence of the following series:

   (a) $\sum_{i=1}^{\infty} 1/i^{1.1}$

   (b) $\sum_{i=2}^{\infty} 1/\big(i\ (\ln(i))^2\big)$

   (c) $\sum_{i=2}^{\infty} 1/(i\ln(i))$

   Specifically, for each infinite sum $\sum_{i=k}^{\infty} a_i$, derive an explicit formula for the corresponding finite sum $S_\ell := \sum_{i=k}^{\ell} a_i$ as a function of ℓ, and then determine if $\lim_{\ell\to\infty} S_\ell$ is finite or infinite.

   Note: the limit ratio test is inconclusive on these examples; as above, you can use an online calculator to compute an indefinite integral.

5. **Mystery algorithm.** Consider the following algorithm, which operates on an array $A[1\,..\,n]$ of integers.

```
for $i$ in $[1\,..\,n]$ do
    $A[i] \leftarrow 0$
for $i$ in $[1\,..\,n]$ do
    $j \leftarrow i$
    while $j \le n$ do
        $A[j] \leftarrow A[j] + 1$
        $j \leftarrow j + i$
```

   (a) Show that the running time of this algorithm is $O(n \log n)$.

(b) Describe in words the value of $A[i]$ at the end of execution.

6. **Number of internal nodes in a 2-3 tree.** Prove that a 2-3 tree with n leaves has at most $n-1$ internal nodes.

   Prove this by induction on the height of the tree; that is, the induction hypothesis is:

   Q_h: For every 2-3 tree T of height h, if T has n leaves and m internal nodes, then $m \le n-1$.

   Prove Q_0 directly (this is really trivial), and then prove that $Q_{h-1} \implies Q_h$ for all $h > 0$. To do this, use the proof of Claim 1 in Section 2 of the *Notes on 2-3 trees* as a template as a template for your proof.

7. **Fibonacci numbers.** Recall the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and $F_{k+2} = F_k + F_{k+1}$.

   (a) Prove that for all $k \ge 0$: $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$.

   (b) Prove that for all $k \ge 0$: $F_{k+2} \le \phi^{k+1}$, where $\phi = (1+\sqrt{5})/2 \approx 1.618$ is a root of $x^2 = x + 1$.

   (c) Prove that for all $k \ge 0$: $F_{k+2} \ge \phi^k$, where ϕ is as in part (b).

   Prove these by induction on k. For each part, formulate an appropriate induction hypothesis P_k. For parts (b) and (c), use so-called "strong induction" to prove this, by proving the following statement holds for each $k \ge 0$:

   $$P_0, P_1, \ldots, P_{k-1} \implies P_k.$$

   Note that when $k = 0$, this statement is logically equivalent to saying that P_0 is true. However, for $k > 0$, you prove the statement by assuming $P_0, \ldots, P_{k-1}$ are true, and from this, showing that P_k must be true as well. Contrast this to "ordinary induction", where you have to prove that P_k is true assuming only that P_{k-1} is true.

8. **Dynamic vectors.** Consider the following implementation of a dynamic-sized vector class in Java:

```
 1  class Vector {
 2     int size;
 3     int capacity;
 4     double[] data;
 5
 6     Vector() { size = 0; capacity = 1; data = new double[1]; }
 7
 8     void resize(int newsz) {
 9        if (capacity < newsz) {
10           capacity = newsz;
11           double[] newdata = new double[capacity];
12           for (int i = 0; i < size; i++)
13              newdata[i] = data[i];
14           data = newdata;
15        }
16        size = newsz;
17     }
18
19     void append(double val) { resize(size+1); data[size] = val; }
20  }
```

   (a) Suppose that the following code is executed:

```
Vector vec;
for (int i = 0; i < n; i++) vec.append(1.0);
```

   Show that the number of times line 13 is executed is $\Theta(n^2)$. The best way to do this is to give an exact formula for the number of times it is executed.

   (b) Suppose that we modify line 10 so that it reads as follows:

```
            capacity = Math.max(2*capacity, newsz);
```

Suppose that we again execute the code in part (a). Show that the the number of times line 13 is executed is $O(n)$. Again, the best way to do this is to give an exact formula for the number of times it is executed.

Discussion. Many programming languages provide such a dynamic-sized vector as a part of a standard library. As you can see, the strategy of doubling the capacity gives much better overall performance than the naive strategy of increasing it by the minimum required. With this strategy, the time required to perform n append operations is $O(n)$. Because of this, it is sometimes said that the "amortized cost" of this strategy is $O(1)$, i.e., constant, which is to say that even though some append operations cost more and some cost less, averaging over a long sequence of such operations, each one takes a constant amount of time.