Basic Algorithms — Fall 2020 — Problem Set 1
Due: Tuesday, Sept. 29, 11am

INSTRUCTIONS FOR SUBMISSION (read carefully):

- You will submit these on NYU Classes.
- You will find on NYU Classes assignments called PS1.1, PS1.2, PS1.3, ..., PS1.9.
- Your solution for Problem 1 should be submitted to PS1.1, for problem 2 to PS1.2, for problem 3 to PS1.3, and so on.
- Each submitted solution should be a PDF file and should contain only the solution to the corresponding problem, and no other solutions.
- Each PDF file may be obtained from a photo/scan of hand-written work, or generated by some digital system (such as LaTex).
- For hand-written work:
  - To meet the other submission requirements, you should start each solution on a separate piece of paper.
  - Your writing must be legible.
  - It is recommended you try one of the free scanning apps that are available, rather than just snapping photos.
- No late submissions will be accepted.
- **To repeat:** no credit will be given for submissions that are either (1) not uploaded properly to the correct NYU Classes assignment, (2) not in PDF format, (3) not legible, or (4) late.

TIPS:

- Get an early start! Don't wait until the day before it is due.
- Some problems might take you a few minutes, while others might take you a couple of hours to grapple with. If you get stuck on a problem, set it aside and move on to something else, and let the problem bounce around inside your head for a couple of days.

1. **Rates of growth.** For each pair of functions, $f(n)$ and $g(n)$, compute $\lim_{n\to\infty} f(n)/g(n)$, and then apply the Limit Comparison Theorem for Rates of Growth from class to determine if $f = o(g)$, $g = o(f)$, or $f = \Theta(g)$. Show your work.

$f(n)$	$g(n)$
$n(\log_2 n)^2$	$n^2(\log_2 n)$
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	$5^{(n^2)}$
$n!$	$(n+1)!$
n	$2n + (-1)^n\sqrt{n}$

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

Show your work. However, you can use an online calculator (such as Wolfram Alpha) to compute antiderivatives.

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

4. **Integral test.** Consider an infinite series $\sum_{i=k}^{\infty} a_i$, where all of the a_i's are nonnegative.

   - To show convergence, we can use the technique of estimating the sum $S_n := \sum_{i=k}^{n} a_i$ by an integral to find a constant U such that $S_n \leq U$ for all sufficiently large n.
   - To show divergence, we can use the technique of estimating the sum S_n by an integral to find a function $L(n)$ such that $S_n \geq L(n)$ for all sufficiently large n, and $\lim_{n\to\infty} L(n) = \infty$.

   Use this method to determine if the following series converge or diverge:

   (a) $\sum_{i=1}^{\infty} 1/i^{1.1}$
   (b) $\sum_{i=2}^{\infty} 1/(i \ln(i))$

   Show your work. However, you can use an online calculator (such as Wolfram Alpha) to compute antiderivatives.

   Note: the ratio test is inconclusive on these examples.

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] ← 0
for i in [1..n] do
    j ← i
    while j ≤ n do
        A[j] ← A[j] + 1
        j ← 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 $m \leq n - 1$.

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

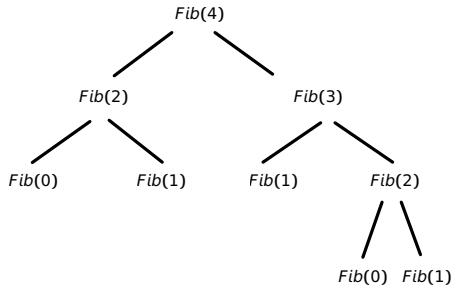
$$P_0, P_1, \ldots, P_{k-1} \implies P_k.$$

   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. **Computing Fibonacci numbers.** Consider the following recursive algorithm for computing the Fibonacci number $F(n)$ on input $n \geq 0$:

```
Fib(n):
    if n ≤ 1
        then return n
        else  return Fib(n − 2) + Fib(n − 1)
```

(a) A *recursion tree* is a way of representing all of the invocations of a recursive algorithm, starting from an initial invocation. Each recursive invocation corresponds to a node in the tree, and the recursive calls made by that invocation correspond to the children of that node. The initial invocation corresponds to the root of the tree. For example, the recursion tree for $Fib(4)$ is:



Draw the recursion tree for $Fib(5)$.

(b) For $n \geq 1$, define $G(n)$ to be the total number of recursive calls made to Fib when initially invoked as $Fib(n)$. For example, looking at the recursion tree for $Fib(6)$ above, we see that $G(4) = 9$. Prove, by induction on n, that $G(n) = 2F(n+1) - 1$ for all $n \geq 0$.

Note: this result, together with part (c) of the previous exercise, implies that algorithm Fib on input n runs in time *exponential* in n.

(c) Give an algorithm that computes $F(n)$ using $O(n)$ arithmetic operations.

Note: even though your algorithm uses just $O(n)$ arithmetic operations, the numbers involved in these operations get very large, resulting in integer overflow unless you use something like Java's `BigInteger` class. In Python, this is not an issue, as it automatically uses a `BigInteger`-like implementation behind the scenes.

9. **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-1] = val; }
20 }
```

(a) Suppose that the following code is executed:

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

Show that the number of times line 13 is executed is $\Theta(n^2)$. To do this, 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 number of times line 13 is executed is $O(n)$. To do this, 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.