Problem Set 1

Scott Robinson
Collaborators: Itamar Belson

This problem set is due **Tuesday, September 17** at **11:59PM**.

Solutions should be turned in through the course website. **Please download the solution templates (there is one LaTeX template and two Python templates) which are available on the course website.** Modify both, and submit them at our submission site. [1]

Programming questions will be graded on a collection of test cases (including example test cases to help you debug). Unless you see an error message, *you will be able to see your grade immediately.* Your grade will be based on the number of test cases for which your algorithm outputs a correct answer within certain time and space bounds. You may submit as many times as you'd like, and only the final submission counts! **Therefore, make sure your final submission is what you want it to be.**

For written questions, full credit will be given only to correct solutions that are described clearly *and concisely.*

---

[1] Register an account, if you haven't done so. Then go to Homework, Problem Set 1, and upload your files.

Problem 1-1. [15 points] **Asymptotic Growth**

This problem should be submitted using `pset1_1_solution_template.py`.

Problem 1-2. [30 points] **Binary Search Variant**

In this problem the input includes an array A such that $A[0\,..\,n-1]$ contains n integers that are sorted into non-decreasing order: $A[i] \leq A[i+1]$ for $i = 0, 1, \ldots, n-2$. The array A *may contain repeated elements*, e.g.

$$A = [0, 1, 1, 2, 3, 3, 3, 3, 4, 5, 5, 6, 6, 6, 6]$$

(a) [10 points]

Describe carefully an algorithm $L(A, s, t, x)$ that, given an array A, an integer s, an integer t, and an integer x, returns the least integer i such that $s \leq i \leq t$ and $A[i] \geq x$ (or returns $t+1$ otherwise).

In other words, if at least one element $A[i]$ is equal to x (with $s \leq i \leq t$), then L returns the least (leftmost) such index i. If no element is equal to x but there is an element that is greater than x, then L returns the least such index i such that $A[i] > x$.

If all elements in A are less than x, your algorithm should return $t+1$. If $s > t$, your algorithm should also return $t+1$.

Your algorithm should be a form of binary search for efficiency.

The initial call on an n-element array A would be of the form $L(A, 0, n-1, x)$.

Binary search is notoriously difficult to get correct; you may find it useful to test your algorithm in Python, but we do not require you to do so. You may describe your algorithm in pseudo-code or in Python, as you prefer. (If you give Python, just include it in your PDF; we will not run this code.)

```
def binSearch(A,s,t,x):
        if s == t:
                return s
        if(t >= len(A)):
                t = len(A)-1
        if s > t or x > A[t]:
                return t+1

        i = int((t + s)/2)

        if A[i] >= x:
                return binSearch(A,s,i-1,x)
        else:
                return binSearch(A,i+1,t,x)
```

(b) [5 points]

Explain carefully why your algorithm always terminates (doesn't loop forever).

There is a conditional to check whether s equals t, where the two are converging toward one another recursively. When they are equal, the algorithm knows it has reached the end of the tree, so it returns s, the index of the lowest element matching the criteria.

(c) [5 points]

Explain why your algorithm doesn't access any elements of A outside of $A[s\,..\,t]$.

In order to meet the criteria given, the index must fall between s and t, so there will never be a lower bound outside of the indexes of s and t. This is a time-saving heuristic, which was achieved by passing s and the current middle index -1 for recursing backward, and the middle index + 1, t when recursing forward.

(d) [5 points]

Explain why your algorithm terminates with the correct answer.

As stated in part B, the program knows to stop recursing when s equals t, and of these two, s will always represent the index of the lowest matching element. This is because is every step of the recursion, if the criteria were matched, we attempted to move further leftward in the list, to reach the minimum index that would work.

(e) [5 points]

Explain why your algorithm runs in time $O(\log n)$ on an n-element input array $A[0\,..\,n-1]$.

This is because when using recursion, in order to reach the point we want, we divide the list by two each time; and the number of times we need to divide by 2 to reach 1 element is the log of n (because log is essentially what power 2 must be raised to to reach n).

Problem 1-3. [55 points] **Factorial Function**

In this problem, we ask you to experiment with three different implementations of the factorial function. The key points are to ensure your familiarity with Python, to emphasize the power of divide and conquer approaches, to note that the time required to multiply big numbers depends on how big those numbers are, and to use the fact that with polynomial running times a constant-factor change in the input size produces a constant factor change in the running time.

The factorial function is well known:

$$n! = \text{factorial}(n) = 1 \cdot 2 \cdots n$$

Part (a) is a Python (version 2.7) script to be uploaded. Parts (b)–(f) are part of your PDF submission, with the other problems in this pset.

(a) [15 points] This problem should be submitted using `pset1_3a_solution_template.py`.

(b) [15 points] Measure the running time of each method: for $k = 0, 1, \ldots,$ compute factorial(2^k) using that method, and measure its running time.

If you import the Python module `time`, then a call to `time.time()` returns the current time. (There are other approaches to measuring time in Python, but this one is simple and sufficient on most systems; you may wish to use `time.clock()` or `timeit` instead—see the Python documentation.)

You may stop when the running time exceeds 10 minutes for that method, although we encourage you to attempt to try up to $k = 20$ at least, and more if you have time.

For each method, give the running time for the largest n for which you were able to measure the running time, and also give the running time for the second-largest value $n/2$. (Be sure to say what n and $n/2$ are as well.)

Which of the three method(s) appears to be asymptotically the fastest, based on your experiments?

(We note that you might have obtained different answers if you had run Python 3; they have changed the math library between versions.)

For Fact1, running 2^20 took 1072s, (~18m). 2^10 took only .0005s
For Fact2, running 2^20 took 1062s (~18m). 2^10 took only .0005s For Fact3, running 2^20 took 494s (~8m). 2^10 took only .0009

Therefore, the recursive method is the fastest of the three.

(c) [10 points] Determine a rough "rate of growth" of the running time for each method as follows, using your experimental data and the following method.

Assume that the running time is approximately of the form

$$T(n) = r \cdot n^s \tag{1}$$

for some constants r, s (this ignores log factors and low-order terms, which is OK for this rough estimation).

For each method, estimate r and s by

$$s \approx \lg \frac{T(n)}{T(n/2)}, \tag{2}$$

$$r \approx \frac{T(n)}{n^s}, \tag{3}$$

where n is the largest input for which that method was timed, and $T(n)$ and $T(n/2)$ are the measured running times of that method for those input values. Here $\lg n$ denotes the logarithm of n to the base 2. Make sure you understand why the assumption of polynomial-time rate of growth in equation (1) justifies equations (2) and (3); with polynomial growth rates a constant-factor change in the input size yields a exponent-s-dependent constant-factor change in the running time.

Turn in your estimates for r and s for each method, approximated using the above equations.

fact1: s = 21.027, r = 4.703e-25 fact2: s = 21.027, r = 4.703e-25 fact3: s = 19.065, r= 7.766e-23

(d) [5 points]

Let $M(n)$ denote the time required to multiply two n-bit integers together.

Sketch an argument that multiplication has running time that is at most quadratic; that is, that $M(n) = O(n^2)$, based on "high-school multiplication".

High-school multiplication requires that you multiply each digit of the first number by each digit of the second number, meaning that it will become nxm operations, which is quadratic.

(e) [5 points]

Give a recurrence for the running time $T(n)$ for computing $n!$ using your divide-and-conquer method.

Your recurrence may involve both T and M on the right-hand side, where $M(n)$ denotes the time needed to multiply two n-bit numbers.

As an approximation, you may assume when computing $n!$ that each of the integers $1, 2, \ldots, n$ is $\lg n$ bits long, and that the length of the product of two integers is equal to the sum of their lengths, so that the product of any k integers between 1 and n may be assumed to have length $k \lg n$.

T(N) = 2T(n/2) + M((n/2)logn)

(f) [5 points]

Do your results for `fact3` support the hypothesis that the multiplication of large integers in Python is performed using some method with *sub-quadratic* running time? Explain. (Hint: consider only the last multiplication your method uses...)

(Perhaps it is only the sub-quadratic algorithm for multiplying large integers that helps make divide-and-conquer so effective for computing factorials??)

Yes. The final multiplication of the recursion is the product of all numbers less than n/2 times the product of all numbers greater than n/2, which are both huge numbers. This is different from the normal method in fact1 where it is only n multiplied by the product of the rest of the numbers. Therefore, it is very likely that python uses a subquadratic algorithm when both numbers being multiplied are huge.