Problem Set 1

Kevin Peng
Collaborators: Genghis Chau, Anubhav Jain, Xubo Sun

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 binary_search(A, s, t, x):
 1  if t - s < 0:
 2    return t + 1
 3  if s == t:
 4    if A[s] >= x:
 5      return t
 6    else:
 7      return t + 1
 8  mid = (s + t) / 2
 9  y = A[mid]
10  if y >= x:
11    return binary_search(A, s, mid, x)
12  if y < x:
13    return binary_search(A, mid + 1, t, x)
```

This algorithm uses a divide-and-conquer approach to return the least integer i such that $s \leq i \leq t$ and $A[i] \geq x$ (or returns $t+1$ otherwise). First, the algorithm checks if $s > t$ explicitly and returns $t+1$ if it's true. Then, the algorithm checks a base case: when there is only one index to check. If the element at that index is greater than or equal to x, we return that index. Otherwise, we return the next index. If we have not yet hit the base case, using a recursive call, the algorithm reduces the number of elements it checks until it hits the base case. It does this by picking the middle element and comparing it to the element x. If the middle element is smaller than x, it will search the right half of the subarray $A[s \ldots t]$, which is $A[mid+1 \ldots t]$. Otherwise, it will search the left half, which is $A[s \ldots mid]$.

(b) [5 points]

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

The algorithm first checks for the base cases of whether $t - s < 0$ or $t - s = 0$. If it is either of these two cases, then the algorithm immediately terminates. Otherwise, the algorithm recursively calls itself on one of the ranges $[s, \lfloor \frac{s+t}{2} \rfloor]$ or $[\lfloor \frac{s+t}{2} \rfloor + 1, t]$ and we have the fact that $t - s > 0$. We conjecture that in each of these recursive calls, the new range is strictly less than the range $[s, t]$, so $t - s$ is always decreasing, and will eventually satisfy one of the base cases.

Case 1: The algorithm recursively calls itself on the range $[s, \lfloor \frac{s+t}{2} \rfloor]$. `(Line 11)`
We want to show that $t - s > \lfloor \frac{s+t}{2} \rfloor - s$.

$$\begin{gathered} t > s \\ t - s > 0 \\ \frac{t-s}{2} > 0 \\ t - s > \frac{t-s}{2} \\ t - s > \frac{s+t}{2} - s \\ t - s > \lfloor \frac{s+t}{2} \rfloor - s \end{gathered}$$

Case 2: The algorithm recursively calls itself on the range $[\lfloor \frac{s+t}{2} \rfloor, t]$. `(Line 13)`
We want to show that $t - s > t - (\lfloor \frac{s+t}{2} \rfloor + 1)$.

$$\begin{gathered} t > s \\ t - s > 0 \\ \frac{t-s}{2} > 0 \\ t - s > \frac{t-s}{2} \\ t - s > t - \frac{s+t}{2} \\ t - s > t - (\lfloor \frac{s+t}{2} \rfloor + 1) \end{gathered}$$

In both cases, we also note that the old difference and the new difference are integers, so the new difference is an integer amount less than the old difference. This means that one of our base cases will be reached eventually, so the program will terminate.

(c) [5 points]

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

The algorithm only indexes into the array in `line 4` and `line 9`. In the initial call in `line 4`, we access $A[s]$, which is in the range of $A[s \dots t]$. In each recursive call, we reduce the range to either $[s, \lfloor \frac{s+t}{2} \rfloor]$ or $[\lfloor \frac{s+t}{2} \rfloor + 1, t]$.

Invariant: We let s_0 denote the initial s and t_0 denote the initial t. We conjecture that in each recursive call, the new s and t maintain the property that $s_0 \leq s \leq t \leq t_0$.
We know that $t - s \geq 1$ since we have reached `line 8`.
We split this up into cases:
Case 1: Recursive call in `line 11`

$$\begin{aligned} t - s &\geq 1 \\ s &\leq t - 1 \\ 2s &\leq s + t - 1 \\ s &\leq \tfrac{s+t-1}{2} \\ s &\leq \lfloor \tfrac{s+t}{2} \rfloor \end{aligned} \qquad \begin{aligned} s &\leq t \\ \tfrac{s+t}{2} &\leq t \\ \lfloor \tfrac{s+t}{2} \rfloor &\leq t \end{aligned}$$

Case 2: Recursive call in `line 13`

$$\begin{aligned} s &\leq t \\ s &\leq \tfrac{s+t}{2} \\ s &\leq \tfrac{s+t-1}{2} + 1 \\ s &\leq \lfloor \tfrac{s+t}{2} \rfloor + 1 \end{aligned}$$

If $t - s$ is odd, then we have

$$\begin{aligned} s + 1 &\leq t \\ s + t + 1 &\leq 2t \\ \tfrac{s+t+1}{2} &\leq t \\ \tfrac{s+t-1}{2} + 1 &\leq t \\ \lfloor \tfrac{s+t}{2} \rfloor + 1 &\leq t \end{aligned}$$

The last line comes from the fact that $\frac{s+t-1}{2} = \lfloor \frac{s+t}{2} \rfloor$ when $t - s$ is odd.
If $t - s$ is even, then we have

$$\begin{aligned} s + 2 &\leq t \\ s + t + 2 &\leq 2t \\ \tfrac{s+t}{2} + 1 &\leq t \\ \lfloor \tfrac{s+t}{2} \rfloor + 1 &\leq t \end{aligned}$$

From this, we have shown that $s \leq mid = \lfloor \frac{s+t}{2} \rfloor \leq t$. Thus, the algorithm does not access an element outside of $A[s]$ in `line 9`. In each of the recursive calls, the new s and t (which are either $\lfloor \frac{s+t}{2} \rfloor$ or $\lfloor \frac{s+t}{2} \rfloor + 1$ are smaller than the old s and t, so the invariant, $s_0 \leq s \leq t \leq t_0$, holds. Thus, with an initial call on the array $A[s \dots t]$, no values out of this range are accessed.

(d) [5 points]

Explain why your algorithm terminates with the correct answer.

We split this analysis into cases.
Case 1: $t - s < 0$. The problem tells us to specially handle this case. We return $t + 1$ as the problem states.

Case 2: $t - s \geq 0$ and x is an element in the array.

Let i be the smallest integer such that $s \leq i \leq t$ and $A[i] \geq x$. Thus, $A[i]$ is the element at the index that is the solution to the search algorithm. We note that a solution must exist since x is an element in the array.
We conjecture that the following invariant holds: in every recursive call, the solution i is in the range of indices checked by the recursive call.
In `line 11`, we check if $A[mid] \geq x$ and if it is, we recursively call this function on $A[s \ldots mid]$. In this case, $A[mid]$ could potentially be a solution since it is $\geq x$, so the solution must exist in $A[s \ldots mid]$. The indices to the right of mid are ruled out because they are larger than mid. Otherwise, we recursively call this function on $A[mid + 1 \ldots t]$. x must exist in $A[mid + 1 \ldots t]$ since A is given to be sorted (so x is greater than all elements in $A[s \ldots mid]$).
From part `2b` we showed that $t - s$ is always decreasing and from our comparisons in `2c` we can easily conclude that the $t - s$ is always ≥ 0 if initially $t - s \geq 0$. From this, we can conclude that eventually our recursive call will break down into the base case of a 1 element array, which must contain the solution by our invariant. We return the index of this element in `line 4`.

Case 3: $t - s \geq 0$ and x is not an element in the array.

We break this case up into subcases.
Subcase 1: There exists an element that is $\geq x$.
Then there exists a solution in $A[s \ldots t]$. Knowing this, we can proceed in the exact same manner as case 2.
Subcase 2: If there doesn't exist an element that is $\geq x$.
Then only the second recursive call executes repeatedly. Again from part `2b` and `2c`, it will terminate when $s == t$ (1-element array), and thus we will return $t + 1$ as a result of `line 6`. Thus, we would get $t + 1$ overall as the answer, which is correct for the case of when all elements in the array are smaller than x.

These 3 cases cover all possibilities, so our algorithm terminates with correctness.

(e) [5 points]

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

Let $T(n) = T(t-s)$ denote the time it takes to run this algorithm for a problem that starts at index s and ends at index t.

We analyze the runtime of this algorithm by looking at the runtime for each line. Lines 1 - 10 and `line 12` each take constant time. `Line 11` takes $T(\lfloor \frac{s+t}{2} \rfloor - s) < T(\frac{s+t}{2} - s) = T(\frac{t-s}{2}) = T(\frac{n}{2})$ time. `Line 13` takes $T(t - (\lfloor \frac{s+t}{2} \rfloor + 1)) < T(t - \frac{s+t}{2}) = T(\frac{n}{2})$. `Lines 11 and 13` are mutually exclusive: one will only execute if the other doesn't.

Thus, we have the recurrence relation:

$$T(n) < T(\frac{n}{2}) + \theta(1) \text{ or } T(n) < T(\frac{n}{2}) + c$$

Based on the master theorem, we can solve:

$$T(n) = T(\frac{n}{2}) + c$$

using case 2 since $c = \Theta(n^{\log_2 1} = n^0)$ The master theorem yields $T(n) = \Theta(\log n)$, but since we have a less than sign in our recurrence relation instead of equality in our actual recurrence relation, $T(n) = O(\log n)$ is more appropriate as a solution.

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.)

`fact1`: 2^{20}: 708.398478031 seconds
`fact1`: 2^{19}: 159.622743845 seconds
`fact2`: 2^{20}: 683.857939959 seconds
`fact2`: 2^{19}: 154.835020065 seconds
`fact3`: 2^{23}: 512.044433832 seconds
`fact3`: 2^{22}: 163.830906153 seconds
`fact3` appears to be asymptotically the fastest.

(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 (**??**) justifies equations (**??**) and (**??**); 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.

$$s_1 \approx \lg \frac{708.398478031}{159.622743845} \approx 2.149$$
$$r_1 \approx \frac{708.398478031}{(2^{20})^{2.149}} \approx 8.16598 \times 10^{-11}$$

$$s_2 \approx \lg \frac{683.857939959}{154.835020065} \approx 2.143$$
$$r_2 \approx \frac{683.857939959}{(2^{20})^{2.143}} \approx 8.56683 \times 10^{-11}$$

$$s_3 \approx \lg \frac{512.044433832}{163.830906153} \approx 1.644$$
$$r_3 \approx \frac{512.044433832}{(2^{23})^{1.644}} \approx 2.12208 \times 10^{-9}$$

(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".

For the multiplication of two n-bit integers, there are two steps:
1. The multiplication of the first integer by each digit in the second integer, producing n integers, each of which is at most $2n$-bits in length.
2. The addition of each of these results.

We assume each 1-bit by 1-bit multiplication takes constant time. There are a total of n^2 of these multiplications, so this step takes $O(n^2)$ time.
We assume that adding two 1-bit numbers takes constant time. Based on "high-school multiplication," to compute the overall answer, we add the numbers in each column. There are n numbers in one column at most, so adding the numbers in one column takes $O(n)$ time. Since there are at most $2n$ columns, the addition step also takes $O(n^2)$ time.
Thus, overall multiplication of two n-bit integers takes $O(n^2) + O(n^2) = O(n^2)$ time.

(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 $k \lg n$.

We define a new notation $T_b(n)$ as the time it takes to multiply n integers, each of length b bits, where b is fixed.
We assume $T(n) = O(T_{\lg n}(n))$. We split the multiplication into two groups in our divide-and-conquer method: $1 \ldots \lfloor \frac{n}{2} \rfloor$ and $(\lfloor \frac{n}{2} \rfloor + 1) \ldots n$. Since each of these are less than $\lg n$ bits in length, we can bound the time to compute the product for both groups with $T_{\lg n}(\frac{n}{2})$. The divide cost to split it into two groups is $O(1)$. The combine cost is equal to the cost to multiply the two products together. Using the given approximation, we find that the length of the first product is $\lfloor \frac{n}{2} \rfloor \lg n \leq \frac{n}{2} \lg n$ and the length of the second product is also $\leq \frac{n}{2} \lg n$. Thus, the combine cost is $\leq M(\frac{n}{2} \lg n)$.
Putting this all together, we have the recurrence $T_{\lg n}(n) \leq 2T_{\lg n}(\frac{n}{2}) + M(\frac{n}{2} \lg n)$.
Since $T(n) = O(T_{\log n}(n))$, we can substitute to get $T(n) \leq 2T(\frac{n}{2}) + M(\frac{n}{2} \lg n)$.

(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??)

My results for `fact3` support the stated hypothesis. Based on part `3c`, $T(n) \approx 2.12208 \times 10^{-9} \times n^{1.644}$, so the algorithm is *sub-quadratic* running time. This supports the hypothesis that the multiplication of large integers in Python is performed in *sub-quadratic* running time because if it weren't, then the running time of $T(n)$ would be at least quadratic. However, our results from `3c` reveal that `fact3`, whose final multiplication is the multiplication of two large integers, is *sub-quadratic*, so we conclude that the multiplication algorithm is also *sub-quadratic*.