CS240E S26	Tutorial 6 (Extension)	June 19

The following questions may be challenging, since they are mostly extension content. Question 1(i), (ii), (iii)(a), and 2(i) should be solvable using in class knowledge only. Familiarity with contest programming will help.

Q1. Answering Range Queries. Suppose we have a list of n numbers $a[1..n]$. Given a range $[l, r]$, we want to compute the **minimum** for the subarray $a[l, r]$.

(i) Given a range $[l, r]$, give an algorithm that answers the query in $O(n)$ time.

(ii) Suppose instead we want to find the k^{th} smallest item in each range. Given a range $[l, r]$ and k, give an algorithm that answers in $O(n \log k)$ time.

(iii) Suppose we have a list of q queries $[l, r]$, and we are finding the minimum.

  (a) Suppose we have $q \in \Theta(n^2)$ queries. Give an algorithm that answers all q queries in $O(n^2)$ time.

  (b) Suppose we have $q \in \Theta(n)$ queries. Give an algorithm that answers all q queries in $O(n \log n)$ time.

  (c) Suppose we have $q \in \Theta(\sqrt{n})$ queries. Give an algorithm that answers all q queries in $O(n)$ time.

(iv) What if we want to update values in the list?

(v) What if we want to support insertion or deletion to the list?

Q2. Non-associative, Offline Range Queries. Suppose we have a list of numbers $a[1..n]$. Given a range $[l, r]$, we want to compute the **mode** (most common element) for the subarray $a[l, r]$. Notice that unlike Q1, this operation is **non-associative**, and can't be computed directly based on the modes of the sub-ranges.

(i) We want to compute the mode for ranges form $a[1..i]$ for $1 \leq i \leq n$. Give an algorithm to do this in $O(n \log n)$.

(ii) Suppose you are given a list of queries $Q[1..q]$, where each query is a pair of integers $[l, r]$ asking for the mode within the range $a[l..r]$. You may sort the queries in $O(q \log q)$ time prior to your algorithm (this is an "offline" algorithm). Come up with an algorithm to, after sorting, answer all queries in $O((n + q)\sqrt{n} \cdot \log n)$.

(iii) Again, assume that you may sort the queries before processing them. Is it possible to do part (b) in $O(n\sqrt{q} \cdot \log n)$?

T04Q6. vEB tree This question is based on CLRS, and past tutorial materials prepared by other ISAs. Delayed from 2 tutorial ago.

A *van Emde Boas* (vEB) tree is a data-structure that supports operations search, insert, delete, successor operations (and their immediate derivatives) in worst-case $O(\log \log u)$ under the assumption that all keys come from the set $\{0, 1, \ldots, u-1\}$, which we call the *universe*.

In this question, we motivate and develop the vEB tree.

(a) For the following data structures with a fixed universe size, what is the time complexity of each of the `search`, `insert`, `delete`, and `successor` operations? You can assume $u = 2^{2k}$ for some appropriate k.

(i) An array of size u, each slot representing whether that integer is in the set.

(ii) A complete binary tree.

(iii) A tree of height 2, where each node has $\sqrt{u}$ children. Each non-leaf node contains a summary of its children - namely, if any single value is a 1.

(b) Consider (iii) from part (a). We note that the summary itself is a $\sqrt{u}$ bit array, and so is each of the children. Suppose $u = 2^{2^k}$ for convenience. What if we recursively apply the structure where each node of size n has $\sqrt{n}$ children each of size $\sqrt{n}$? (CLRS calls this a proto-vEB-tree.)

(i) How tall is the tree?

(ii) Consider the successor algorithm on the following page. Provide a recursive formula, and analyze its runtime. Note that high(x) gets the upper half of x's bits, while low(x) gets the lower half. We use $V[i]$ to indicate the ith cluster of V. Assume minimum takes $O(\log u)$ time - with the current structure, we can't do better.

Note that when returning (high(x), offset), we are returning the integer with high(x) as high bits and offset as low bits.

(iii) What about implementing the other operations, such as insertion? Deletion? Are there any issues?

(c) Suppose we also keep track of the minimum element and the maximum element. How can you improve on the operations, such as the example successor() operation, to get the desired $O(\log \log u)$ time complexity?

(d) What if we relax the restriction to $u = 2^k$? How do we still achieve the same height / time complexity?

Algorithm 1: *successor*(V, x)

```
1  if V.u == 2 then
2  |  if x == 0 and V[1] == 1 then
3  |  |  return 1;
4  |  end
5  |  return ∅;
6  end
7  offset ← successor(V[high(x)], low(x));
8  if offset ≠ ∅ then
9  |  return high(x), offset;
10 end
11 successor-cluster ← successor(V.summary, high(x));
12 if successor-cluster == ∅ then
13 |  return ∅;
14 end
   // assume minimum takes O(log u) time.
15 offset = minimum(V[successor-cluster]);
16 return successor-cluster, offset;
```

Number theoretic functions. We will discuss several well-known functions from number theory. If these are of interest, take ~~MATH145~~ PMATH341.

Motivation. For increasing the table size in hashing, we wanted to find a prime of a certain size. Specifically, given an integer $M > 1$, we would like to find a prime of size at least $2M$.

Our approach is based on *Bertrand's postulate*:

> For all $n > 1$, there is a prime p such that
>
> $$n < p < 2n.$$

Q3. Give an $O(M\sqrt{M})$ algorithm to find the next prime that is at least $2M$.

Q4. Sieve of Eratosthenes Consider the sieve on the following page. What is its runtime?

Q5. Give an $O(n \log n \log \log n)$ algorithm to count the number of factors in the prime factorization for each of $2, \ldots, n$.

For example, $20 = 2 \cdot 2 \cdot 5$, so 20 has 3 factors in its factorization. Similarly, $35 = 5 \cdot 7$ has 2 factors in its factorization.

Q6. How can the Sieve of Eratosthenes be adapted to compute all prime numbers in a particular range $[a, b]$ when b is far too big to compute all primes in $[0, b]$?

Algorithm 2: *sieve-of-eratosthenes*(n)

```
A ← array of size n, initialized to 1, except A[1] = 0;
for i from 2 to n do
    if A[i] == 0 then
        continue
    end
    for j = i^2; j < n; j += i do
        A[i] ← 0;
    end
end
```

Not-A-Q7. Other primality tests There are a few other primality tests of interest.

(a) Miller-rabin, a probabilistic primality test. What is its runtime?

Algorithm 3: *miller-rabin*(n)

```
s ← ν_2(n − 1);
// the number of times 2 divides n − 1
d ← (n−1)/2^s;
for some number of predetermined rounds r do
    a ← random(2, n-2);
    a ← a^d (mod n);
    for i from 1 to s do
        if a ≡ n − 1 (mod n) then
            break // probably prime
        end
        if a ≡ 1 (mod n) then
            return "composite"
        end
        a ← a^2 (mod n);
    end
end
return "prime, or composite with probability ≤ (1/4)^r"
```

(b) AKS, which runs in polynomial of $\log n$ time. Omitted since it's rather long.

(c) Lucas-Lehmer, used for Mersenne primes.

Algorithm 4: *lucas-lehmer*(p)

$s \leftarrow 4$ `// some values like` 4, 10, $\frac{2}{3}$ `work here for all primes`
for i *from* 1 *to* $p-2$ **do**
  $s \leftarrow s^2 - 2 \pmod{2^p - 1}$
end
if $s \equiv 0 \pmod{2^p - 1}$ **then**
  return "prime"
end
return "composite"

For the next problems, assume that $n = p_1^{a_1} \cdots p_k^{a_k}$ where $p_1, \ldots, p_k$ are the first k primes and all $a_i \geq 0$. We also assume we already computed the 1-indexed array `primes` with the first (sufficiently many) primes.

The *number of* ***prime*** *factors* of n, which is $\sum a_i$, is easy to find. We have computed it from scratch in Q5.

Q8. Give a way to compute the number of prime factors of n (assuming we already have the array `primes`).

Hint: intended solution has time $\Theta(\pi(\sqrt{n}))$.

Q9. Using Q8, implement `factor`, which given an integer n returns a dynamic array containing all the prime factors of n. The time complexity should be the same.

For example, if $n = 12$, then `factor(n)` returns `[2, 2, 3]`.

We should note that the worst case for the algorithm in Q8 is very slow. If n is a prime, then the algorithm tests all the first $\pi(\sqrt{n})$ primes. Although there are optimizations to the algorithm in Q8, we still do not have a computationally feasible way to factor large numbers, which serves as the basis for cryptography[1] It is fun to think about some special cases nonetheless.

Fermat's factorization method. We can write an odd composite number $n = p \cdot q$ as a difference of squares:

$$n = \underbrace{\left(\frac{p+q}{2}\right)^2}_{a^2} - \underbrace{\left(\frac{p-q}{2}\right)^2}_{b^2}.$$

This factorization method guesses the first square a^2 (as $\sqrt{n}$, the smallest it can be), and then checks if $b^2 = a^2 - n$ is *also* a square. If it is return, otherwise increase a and try again.

[1] See, for example, CO487.

```
fermat(n):
    // returns a factor of n
    a = ceil(sqrt(n))

    b_squared = a*a - n
    b = round(sqrt(b_squared))

    while b * b != b_squared: // while b is not a perfect
        square
        ++a // try the next guess
        b_squared = a*a - n
        b = round(sqrt(b_squared))
```

Q10. Give the asymptotic runtime of this algorithm. When does this algorithm perform well?

It is interesting to note that implementations of encryption that relies on difficulty of factoring are made extremely vulnerable if consecutive primes are used in key generation because of this Factorization method.

We denote by $\tau(n)$ the *number of factors* of n. For example, $n = 60$ has four prime factors: 2, 2, 3, 5. Given the prime factorization of n we may compute it as, $\tau(n) = \prod(a_i + 1)$.

Q11. Give a way to count the number of divisors of n given only the array `prime`.

References

[CLRS] Cormen T., Leiserson C., Rivert R., Stein C. *Introduction to Algorithms.* Third edition.