Algorithms
Homework 5

Saad Mneimneh
Computer Science
Hunter College and Graduate Center of CUNY

Problem 1: Finding right triangles
Consider an array of n points, where each point has integer coordinates in two dimensions (for simplicity, assume that all coordinates are non-negative).

(a) Design an efficient algorithm to determine whether any three of the points make a right triangle of the form $\{(x, y), (x, y'), (x', y)\}$. You should seek an algorithm with linear time (randomized or deterministic).

(b) Assume now that all coordinates are generated uniformly at random in the range $[0, \alpha n]$ for some positive constant α. If we require the right triangle to also be isosceles, describe how we can determine whether such triangle exists in expected linear time.

Problem 2: Hashing as a fingerprint
(This problem is classical, it is included for educational purposes.)

Given a text of length n, $T[1 \ldots n]$, and a pattern of length $m \ll n$, $P[1 \ldots m]$, we would like to find all s such that $T[s+1 \ldots s+m] = P$ (all occurrences of P in T).

The brute force approach requires $O(mn)$ time.

```
for s ← 0 to n − m do                                  ▷ O(n)
    if T[s + 1 . . . s + m] = P[1 . . . m] then        ▷ O(m) check
        output s
```

Ideally, if there are v occurrences of P in T, we would like an algorithm that runs in

$O(n+m+v)$ time. This is possible using a data structure called suffix tree. Instead, we will shoot for $O(n+mv)$ by treating a text as an integer and using a hashing technique.

For simplicity, assume that text is binary and compute:

$$t_s \leftarrow T[s+1]2^{m-1} + \ldots + T[s+m]$$

$$p \leftarrow P[1]2^{m-1} + \ldots + P[m]$$

The algorithm becomes:

```
p ← P[1]2^{m-1} + ... + P[m]                          ▷ O(m)
t_0 ← T[1]2^{m-1} + ... + T[m]                        ▷ O(m)
for s ← 0 to n − m do                                 ▷ O(n)
    if t_s = p then                                   ▷ O(1) check?
        output s
    t_{s+1} ← T[(s+1)+1]2^{m-1} + ... + T[(s+1)+m]    ▷ O(m)
```

Assuming that arithmetic operations take $O(1)$ time, the above algorithm still runs $O(nm)$ time because t_s is computed in $O(m)$ time.

(a) Suggest a way to overcome the bottleneck described above and compute t_s in $O(1)$ time. *Hint*: t_{s+1} and t_s are related.

With part (a) done, the algorithm runs in $O(n+m)$ time. However, when m is large, it is not reasonable to assume that arithmetic operations on m bit integers take $O(1)$ time.

(b) We will pick a prime q with $O(\log n)$ bits (so q is comparable to n) and perform all arithmetic operations modulo q. Our algorithm becomes:

```
p' ← P[1]2^{m-1} + ... + P[m]  (mod q)      ▷ fingerprint of P
t'_0 ← T[1]2^{m-1} + ... + T[m]  (mod q)    ▷ fingerprint of T[1...m]
for s ← 1 to n − m + 1 do
    if t'_s = p' then                       ▷ O(1)
        output s
    t'_{s+1} ← your solution from (a)  (mod q)   ▷ fingerprint
```

We can now claim that the algorithm runs in $O(n+m)$ time. But the algorithm is no longer correct, in particular it can output false positives. Explain why and suggest a solution to make a correct algorithm run in $O(n+m(v+fp))$ time, where fp is the number of false positives. Luckily, if q is chosen from a large enough set of primes, one can prove that $E[fp] = O(1)$.

(c) Do the following to finish up:

- The prime number theorem states that the number of primes in $[1, N]$ is $\Theta(\frac{N}{\log N})$. Let $N = mn \log mn$. Show that we have $\Theta(mn)$ primes in $[1, N]$.
- We repeatedly pick a number uniformly at random in $[1, N]$ until we find a prime q. Assuming that we can test if an $O(\log n)$-bit number is prime in $O(\log^k n)$ time for some constant k, show that the expected time to find q does not affect the asymptotic time of the algorithm. *Hint*: what is the success probability of picking a prime in $[1, N]$?
- If $t_s \neq p$, show that the probability that $t'_s = p'$ (which means $t_s - p \equiv 0 \pmod{q}$) is at most $1/n$. *Hint*: show that $|t_s - p|$ has less than m prime factors. Conclude that $E[fp] = O(1)$.

Problem 3

(a) Consider the following alternative to the in-order-tree-walk on a tree T:

```
x ← minimum(T.root)                          ▷ find minimum of tree
while x ≠ NIL do
    output x.key
    x ← successor(x)                         ▷ find successor
```

It is obvious that this algorithm runs in $O(nh)$ time where h is the height of the tree, since each call to successor requires $O(h)$ time. Show why this algorithm runs in $O(n)$ time, independent of the height. (You have to know how successor works.)

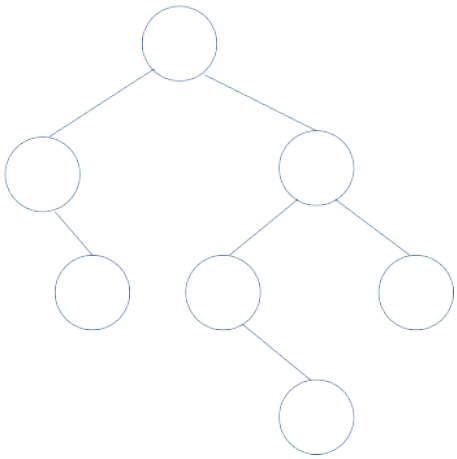
(b) Describe a linear time algorithm to transform a binary search tree into a min-heap.

- In-place, by using $O(1)$ additional memory, and reusing the tree nodes and pointers (this will produce a heap implemented as a tree).
- Not in-place, just produce an array that is a min-heap containing all the keys in the tree (this is easier of course).

(c) Can we transform a min-heap into a binary search tree in linear time? If your answer is yes, provide an algorithm to do it. If your answer is no, explain why.

Problem 4

(a) Place the keys $\{1, 2, 3, 4, 5, 6, 7\}$ in the following binary search tree.



(b) Color the nodes red and black to make the tree a valid red-black tree.

(c) Perform a right rotation at the node containing key 6, and draw the binary search tree that results. Can you color the nodes to make the tree a valid red-black tree? Explain.

Problem 5

Describe how you will augment a redblack tree to support the operation even successor(x), which finds the node y in the tree with the smallest key k such that $k > x.key$ and k is even. We assume all keys are distinct. The even successor operation should run in $O(\log n)$ time, where n is the number of keys stored in the tree. Describe the operation clearly in English and support it with pseudocode.