My favorite linear time algorithms

Notes by Linus Tang.

These notes have not been thoroughly reviewed. Any errors below are my own responsibility.

Sources:

- Bowdoin CS231 lecture notes by Laura Toma
  - https://tildesites.bowdoin.edu/~ltoma/teaching/cs231/fall16/Lectures/07-selection/selection.pdf
- Codeforces user Olympia
  - https://codeforces.com/blog/entry/101010
- The Institute of Mathematical Sciences, Discrete Maths notes by Vikram Sharma
  - https://www.imsc.res.in/~vikram/DiscreteMaths/2011/tree.pdf

Problems

All problems below are to be solved using deterministic algorithms that run in $O(n)$ time. Assume that arithmetic operations on real numbers are $O(1)$ each.

1. **Tree isomorphism:** Given the edge sets of two trees T_1 and T_2 on vertex set $V = \{1, 2, ..., n\}$, determine whether T_1 and T_2 are isomorphic.
2. **Rank select:** Given integers $1 \leq k \leq n$ and a list of n distinct real numbers $a_1, ..., a_n$ in arbitrary order, compute the k-th smallest number among the n.
3. **Largest gap:** Let $t_1 < t_2 < ... < t_n$ be real numbers. Given an arbitrary permutation $a_1, ..., a_n$ of them, compute $\max_{1 \leq i < n}(t_{i+1} - t_i)$.

Solutions begin on the next page.

Solutions

Tree isomorphism

First solve the easier problem of rooted tree isomorphism. Given two rooted trees, determine in linear time whether they are isomorphic.

This comes down to finding, for each non-leaf vertex, a canonical order of its children. The details of how to do this in linear time are nontrivial.

T_1, T_2: trees with n vertices.

v_1, v_2: vertices of T_1 and T_2, respectively.

Output: True if tree T_1 rooted at v_1 is isomorphic to tree T_2 rooted at v_2; False otherwise.

```
RootedTreeIsomorphism($T_1, v_1, T_2, v_2$):
• Label every leaf of  with the number 1.
• Whenever all vertices at depth $i + 1$ of both trees have integer labels, give all vertices at depth $i$
  of both trees integer labels as follows:
    ‣ For every non-leaf vertex $v$ at depth $i$, assign the vertex a tuple consisting of the labels of its
      children in ascending order.
    ‣ In each tree, order the vertices at depth $i$ in lexicographically increasing order by their tuples,
      saying that leaves are the least, and shorter tuples are less than longer tuples.
    ‣ Now you can directly compare the sorted lists of tuples of depth-$i$ vertices in $T_1$ and $T_2$. If they
      don't match, return False. Otherwise, proceed.
    ‣ Replace all instances of the least tuple at depth $i$ with 1, all instances of the next least tuple at
      depth $i$ with 2, and so on. (Here, the label of a leaf counts as the least tuple if there are any.)
    ‣ Now all vertices at depth $i$ have integer labels, so decrement $i$ and repeat until the root has a
      label.
• If the algorithm gets to this point without returning False (and in particular the root labels of the
  trees are equal), return True.
```

When sorting numbers, instead of using $O(n \log n)$ comparison sort, use a sorting algorithm which sorts tuples of length $\ell_1, \ldots, \ell_k$ and entries in $\{1, \ldots, m\}$ in $O\left(m + \sum_{i=1}^{k} \ell_i\right)$ time.

The runtime analysis is tricky, but it can be proven that the above algorithm runs in $O(n)$ time.

The algorithm presented above is named AHU after Aho, Hopcroft, and Ullman.

Now that we've solved the problem of rooted tree isomorphism, we want to do the following: Find a "canonical" vertex from each tree to serve as its root, then apply rooted tree isomorphism.

- One such choice is called the *centroid* of the tree.
- A vertex v in a tree T with n vertices is called a centroid if removing v results in a forest, each of whose components has at most $\frac{n}{2}$ vertices.
- Every tree has either one or two centroids.
  - Existence: See the algorithm below which finds a centroid.
  - Proof that there are at most two: Suppose that there exists a pair of centroids which are not adjacent, and derive a contradiction.

T: a tree with n vertices.

Output: a centroid of T.

```
FindCentroid($T$):
• Root $T$ at an arbitrary vertex $r$.
• Recursively compute the number of descendants of every vertex.
• Initialize $v \leftarrow u$.
• While $v$ has a child $v'$ such that $v'$ has at least $\frac{n}{2}$ descendants:
    ‣ Set $v \leftarrow v'$.
• Return $v$.
```

It is not too hard to prove that this algorithm always returns a centroid of T.

Now if T_1 and T_2 are isomorphic as unrooted trees, they should still be so if we root them at their centroids. In particular, we can solve the problem as follows:

```
$T_1, T_2$: trees with $n$ vertices.

Output: True if tree $T_1$ rooted at $v_1$ is isomorphic to tree $T_2$ rooted at $v_2$; False otherwise.

TreeIsomorphism($T_1, T_2$):
• We have trees $T_1$ and $T_2$ and want to determine whether they are isomorphic.
• Find a centroid $v_1 \leftarrow$ FindCentroid($T_1$).
• Find the set $C$ of centroids of $T_2$.
    ‣ In particular, let $v \leftarrow$ FindCentroid($T_2$). If $v$ has exactly $\frac{n}{2}$ descendants and only one child $v'$,
      then $C = \{v, v'\}$. Otherwise, $C = \{v\}$.
• If there exists $v_2 \in C$ such that RootedTreeIsomorphism($T_1, v_1, T_2, v_2$) returns True, then $T_1$
  and $T_2$ are isomorphic, so return True.
• Otherwise, $T_1$ and $T_2$ are not isomorphic, so return False.
```

Rank select

Let $\operatorname{rank}(a_i) = r$ if a_i is the r-th smallest among $\{a_1, \dots, a_n\}$. So, we want to find a_i such that $\operatorname{rank}(a_i) = k$.

Note that the following can be done in linear time:

- Given an a_i, compute $\operatorname{rank}(a_i)$.
- Given an a_i, determine the set of elements of $\{a_1, \dots, a_n\}$ greater than a_i and the set of elements less than a_i.

Consider the following framework for a recursive algorithm.

$[a_1, \dots, a_n]$: A list of n real numbers in arbitrary order.

k: A positive integer at most n.

Output: the k-th smallest element of the list.

```
RankSelect($[a_1, \dots, a_n], k$) :
• Choose some $a_i \leftarrow$ SelectElement($[a_1, \dots, a_n]$) by a method to be specified later.
• Compute $r \leftarrow \operatorname{rank}(a_i)$ and partition the other elements of $\{a_1, \dots, a_n\}$ into those less than $a_i$
  and those greater than $a_i$, call these $[l_1, \dots, l_{r-1}]$ and $[u_1, \dots, u_{n-r}]$.
• If $r = k$, return $a_i$.
• If $r > k$, return RankSelect($[l_1, \dots, l_{r-1}], k$).
• If $r < k$, return RankSelect($[u_1, \dots, u_{n-r}], k - r$).
```

A worst-case scenario is if k is large and you always choose a_i to be the smallest remaining element, or if k is small and you always choose a_i to be the greatest remaining element. In these cases, RankSelect runs in time quadratic in n.

Thus, a desirable property of a_i is for its rank to not be close to 1 nor n. In particular, if we make the size of the set partition in the recursive call decay exponentially, then the overall runtime will be linear in n.

It turns out that the solution involves using another recursive call to RankSelect to choose a_i. Paramaters need to be chosen to balance the cost of this additional recursive call with the rate of aforementioned exponential decay.

$[a_1, \dots, a_n]$: A list of n real numbers in arbitrary order.

Output: an element of $[a_1, \dots, a_n]$ with a desirable property to be specified later.

```
SelectElement($[a_1, \dots, a_n]$) :
• For convenience, pad $[a_1, \dots, a_n]$ to $[a_1, \dots, a_{n'}]$ for the minimal $n' \geq n$ with $n' \equiv 5 \bmod 10$.
• Arbitrarily partition $[a_1, \dots, a_{n'}]$ into $2s - 1$ groups of 5 elements. Let $m_1, \dots, m_{2s-1}$ be the the
  medians of these groups of 5 elements.
• Find the median of these medians, given by $m \leftarrow$ RankSelect($[m_1, \dots, m_{2s-1}], s$).
• Return $m$.
```

The output has the property that $\frac{3}{10}n \leq \operatorname{rank}(m) \leq \frac{7}{10}n$.

Now we modify the main algorithm and analyze its runtime.

$[a_1, \dots, a_n]$: A list of n real numbers in arbitrary order.

k: A positive integer at most n.

Output: the k-th smallest element of the list.

```
RankSelect([$a_1, ..., a_n$], $k$) :
• If $n \leq 100$, sort the list and return the $k$-th smallest element. Otherwise:
• Set $a_i \leftarrow$ SelectElement([$a_1, ..., a_n$]).
• Compute $r \leftarrow \text{rank}(a_i)$ and partition the other elements of $\{a_1, ..., a_n\}$ into those less than $a_i$
  and those greater than $a_i$, call these $[l_1, ..., l_{r-1}]$ and $[u_1, ..., u_{n-r}]$.
• If $r = k$, return $a_i$.
• If $r > k$, return RankSelect([$l_1, ..., l_{r-1}$], $k$).
• If $r < k$, return RankSelect([$u_1, ..., u_{n-r}$], $k - r$).
```

The call to SelectElement involves some linear time operations and a call back to RankSelect of size $\frac{1}{5}n + O(1)$.

The call to RankSelect at the end has size at most $\frac{7}{10}n$ because $\frac{3}{10}n \leq r \leq \frac{7}{10}n$.

Since the proportions ofthe recursive calls sum to $\frac{1}{5} + \frac{7}{10} < 1$, and since all other operations can be done in linear time, it follows that the overall runtime of RankSelect is linear in n.

Largest gap

Consider the minimum and maximum elements s and t. We will divide the interval between them into $n+1$ equally-sized buckets. For convenience, first apply a linear function

$$x \mapsto (n+1)\frac{x-s}{t-s}$$

to each of the a_i that maps s to 0 and t to $n+1$.

In particular, the buckets are now $[0,1], (1,2], (2,3], \dots, (n, n+1]$.

The key observation is that there is an empty bucket, which means that the largest gap has size at least 1. In particular, the largest gap is between two elements in different buckets.

This leads to the following algorithm:

```
$[a_1, \dots, a_n]$: A list of $n$ real numbers in arbitrary order

Output: The smallest gap between two consecutive elements of the (unprovided) sorted list.

LargestGap($[a_1, \dots, a_n]$)
• Determine the maximum and minimum elements $s$ and $t$.
• For convenience, let $b_i = (n+1)\frac{a_i - s}{t-s}$ for each $i$ and work with $[b_1, \dots, b_n]$ instead.
    ‣ Now the minimum and maximum elements are $0$ and $n+1$.
• Consider buckets $I_1, \dots, I_{n+1} = [0,1], (1,2], \dots,, (n, n+1]$.
• Iterate over the $b_i$, and determine which bucket each belongs to. Keep track of the minimum
  element $m_j$ and maximum element $M_j$ that belongs to each bucket.
• Find the candidates for the largest gaps:
    ‣ For each $M_j$ with $j < n+1$, find the smallest $j' > j$ such that there is a point in bucket $I_{j'}$.
      Compute the gap $m_{j'} - M_j$.
• Return the largest of these candidate gaps.
```