CSC345 Discussion, 09/29/17

Find kth smallest element of array using heaps

Assume A has n elements with $n > k$. How can we find the kth smallest element in $O(n \log k)$ time?

Create a *max* heap of size k. First build the heap using the first k elements of A. This takes $O(k)$ time.

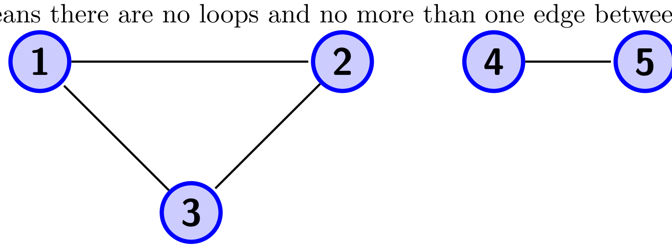
Next, for $i = k + 1$ to n: if $A[i] >$ root (max) of heap, ignore (continue). This is because if $A[i]$ is larger than k other elements of A, it can't be the kth smallest. (If $A[i] =$ root, we can also ignore since deleting a max and inserting the same value is not a very fruitful operation.) If $A[i] <$ root, then the root is larger than at least k elements and we must delete max and insert $A[i]$ into the heap.

This algorithm uses the invariant that after j iterations, the root of the heap is the kth smallest element of the subarray $A[1..k + j]$.

After the loop, extract the max: this is the kth smallest element of A. This algorithm runs in $O(n \log k)$ time. This is not as good as the *expected* $O(n)$ time "quickselect" algorithm (i.e., quicksort, but only recurse on one half), but it's much higher-level and easier to implement mistake-free (provided you have access to a heap library).

Number of simple graphs with n vertices

A graph $G = (V, E)$ is a collection of vertices and edges between them. A simple graph means there are no loops and no more than one edge between two vertices. For example:



Here, $V = \{1, 2, 3, 4, 5\}$ and $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{4, 5\}\}$.

Question: how many *simple* graphs are there with n nodes?

Let's consider a "full" graph — i.e., one where there's an edge between every pair of nodes. There are $n - 1$ edges from vertex 1, $n - 2$ from vertex 2, $n - 3$ from vertex 3, $\ldots$, 2 from vertex $n - 3$, and 1 from vertex $n - 2$.

Thus a "full" (the proper term is *complete*) graph has

$$1 + 2 + \ldots + (n - 1) = \frac{n(n-1)}{2}$$

edges.

But this doesn't answer how many graphs are possible on n vertices. To answer this, consider every possible edge from the complete graph. There are two choices — this edge is either in a given graph or not. So the total number of graphs is

$$2^{n-1} 2^{n-2} \cdots 2^2 2^1 = 2^{1+2+\ldots+(n-1)} = 2^{n(n-1)/2}.$$

Pollard's Rho algorithm for factoring integers

Suppose we want to factor $n = 15811$. (This is a product of two primes.) Let's pick a simple quadratic polynomial, such as $f(x) = x^2 + 1$, and set $x_1 = y_1 = 2$.

Keep on computing $x_i = f(x_{i-1}), y_i = f(f(y_{i-1}))$. Our goal is to find $\gcd(|x_i - y_i|, n) =$ some prime number (then we can divide n by this prime and obtain the other prime).

So compute (these are all mod n):

$$x_2 = 5, y_2 = 26$$

$$x_3 = 26, y_3 = 15622, \gcd(15622 - 26, 15811) = 1$$

$$x_4 = 677, y_4 = 2908, \gcd(2908 - 677, 15811) = 97$$

Thus 97 is a factor of 15811. In fact, $15811 = 97 \times 163$. 163 is prime, so n is fully factored (if it weren't, we could repeat this idea to factor a smaller number, which is generally easier).

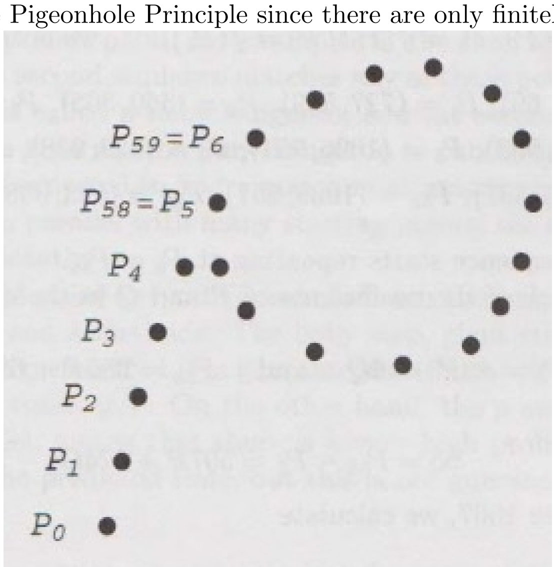
What happened here? Let $P_0 = 2, P_1 = f(P_0), P_2 = f(P_1)$, etc. We have two sequences P_i, P_{2i}. Applying these sequences, we compare:

P_1 with P_2

P_2 with P_4

P_3 with P_6

At some point, we're going to get equality. Why? We certainly get equality $P_i = P_j$ by the Pigeonhole Principle since there are only finitely many points.



(Image source: *Elliptic Curves: Number Theory and Cryptography* by Washington)

Can you prove we get an equality where $j = 2i$?

Notes:

(1) This is a *probabilistic* algorithm in the sense that the gcd will probably yield a factor of n within $O(\sqrt{p})$ iterations for some prime factor p of n (i.e., $O(n^{1/4})$). We may only get $\gcd = n$, in which case we can try a different seed (instead of 2) or a different function (e.g., $x^2 + 2$).

(2) I said that we have $P_i = P_j$ at some point. What this really means is $\gcd(P_i - P_j, n) = d$ for some divisor d of n (hopefully a prime!). Equality means that $P_i \equiv P_j \pmod{d}$. If $P_i \equiv P_j \pmod{d}$, then $P_i - P_j = dk$ for some integer k. Let's suppose $n = dc$. As long as k isn't a multiple of c, we get a proper factor of n. Otherwise, we get $\gcd(P_i - P_j, n) = n$ and we have to choose a different seed or different function.

(3) gcd can be computed very quickly (logarithmic time). Primality checking can also be done very efficiently (but **factoring** is hard).

(4) See `https://github.com/ablumenf/factoring/blob/master/factoring.py` if you want to play around with Pollard Rho for factoring integers.