Randomized Algorithms

We now wish to consider a *modified* notion of algorithms that can make use of a source of randomness at some step. The simplest approach to this is to imagine a "black-box" `rand` that, given a positive integer k returns an element `rand`(k) of $[1, k]$ which is *guaranteed* to be chosen *uniformly* at random from this set.

At first it may be difficult to see how this is useful! Here is a practical example. Given a large tuple $(a_1, \ldots, a_k)$ of elements from $\{0, 1\}$. we want to find an i such that a_i is 0. The simple algorithm to do this is as follows:

define `find0`$(\underline{a})$
  Set k *as the length of* $\underline{a}$.
  for i *in* $[1, k]$
    if a_i *equals* 0
      return i

An algorithm that uses `rand` could be as follows:

define `ranfind0`$(\underline{a})$
  Set k *as the length of* $\underline{a}$.
  Set t *as* 0.
  while t *equals* 0.
    Set i *to be* `rand`(k).
    if a_i *equals* 0
      Set t *as* i.
  return t

This algorithm terminates with probability 1 if there *is* an i such that $a_i = 0$, but there is no bound on the number of steps. We can modify it as follows:

define `ranfind0`$(\underline{a}, b)$
  Set k *as the length of* $\underline{a}$.
  Set t *as* b.
  while $t > 0$.
    Set i *to be* `rand`(k).
    Decrement t.
    if a_i *equals* 0
      return i

While this algorithm *does* stop after b calls to `rand`, it may not return an answer even when there *is* such an i! Here is an algorithm that *definitely* stops and gives an answer when such an i exists:

define `ranfind0`$(\underline{a})$
  Set k *as the length of* $\underline{a}$.
  Set t *as* k.
  while $t > 0$.
    Set i *to be* `rand`(t).
    Decrement t.

if a_i *equals* 0
    return i
if a_t *equals* 0
    return t

Thus, we see that there are three types of randomized algorithms:

- Those that terminate with probability 1 with the correct answer, but there is no bound on the number of steps.
- Those that terminate in a bounded number of steps, but with some probability give an erroneous answer.
- Those that terminate in a bounded number of steps *and* give a correct answer.

In each case, we are also interested in the *expected* number of steps; since the algorithm uses a random input, the number of steps it runs for is a *random variable* T and we want to find the expectation $E[T]$. We are especially interested in the case when this value is significantly smaller than the worst case bound for the number of steps.

Randomized Quick Sort

A brief description of the Quick Sort algorithm to sort a tuple $\underline{a} = (a_1, \ldots, a_k)$ is as follows:

1. If $k \leq 1$ return $\underline{a}$.
2. Choose p in the range $[1, k]$.
3. Create the lists λ (respectively ρ) which consists of a_j with $j \neq p$ such that $a_j \leq a_p$ (respectively $a_j \not\leq a_p$).
4. Sort the lists λ and ρ by a recursive call.
5. Combine to get the answer.

In the description given in an earlier section, we took $p = k$ (some versions take $p = 1$) in (2) above. We now analyse what happens if p is chosen randomly from $[1, k]$.

As mentioned above the time complexity of the algorithm is now a random variable T and we wish to determine its expectation $E[T]$. As discussed earlier, the running time is determined by the number C of comparisons made in the execution; this too is a random variable and so we wish to determine $E[C]$.

Let $X_{i,j}$ denote the random variable that counts the number of comparisons between a_i and a_j in the algorithm. It is clear that

$$E[C] = \sum_{i=1}^{k} \sum_{j=i+1}^{k} E[X_{i,j}]$$

and $X_{i,j} = X_{j,i}$.

Note that the above algorithm is "efficient" in the sense that a_i and a_j are compared *at most once*. Comparisons only happen in (3) so that either i or j must be the chosen p for $X_{i,j}$ to be non-zero. After that, since a_p is not a part of λ or ρ, it is never compared with *any* other element of the tuple again. Thus, $X_{i,j}$ only takes the values 0 or 1 and so

$$E[X_{i,j}] = 1 \cdot P[X_{i,j} = 1] + 0 \cdot P[X_{i,j} = 0] = P[X_{i,j} = 1]$$

It follows that

$$E[C] = \sum_{i=1}^{k} \sum_{j=i+1}^{k} P[X_{i,j} = 1]$$

Let $(b_1, \ldots, b_k)$ be the *ordered* version of the tuple so that there is a permutation σ of $[1, k]$ so that $b_i = a_{\sigma(i)}$.

Now, suppose that $b_i < b_j$. If p is chosen in step (2) so that a_p that *strictly* lies between b_i and b_j (i.e. $b_i < a_p = b_{\sigma^{-1}(p)} < b_j$), then $a_{\sigma(i)}$ is put in λ and $a_{\sigma(j)}$ is put in ρ and they are *never* compared! Thus, if we look at the *first* p chosen in (2) so that $\sigma^{-1}(p)$ is in the interval $[i, j]$, the only way this leads to a comparison between $a_{\sigma(i)}$ and $a_{\sigma(j)}$ is if $p = \sigma(i)$ or $p = \sigma(j)$. Since each element of this interval is equally likely to be chosen, we see that if $b_i < b_j$, then

$$P[X_{\sigma(i),\sigma(j)} = 1] = \frac{2}{(j - i) + 1}$$

where the numerator is 2 to account for the two cases when the comparison *does* happen.

Since σ is a bijection, we see that

$$E[C] = \sum_{i=1}^{k} \sum_{j=i+1}^{k} P[X_{i,j} = 1] = \sum_{i=1}^{k} \sum_{j=i+1}^{k} P[X_{\sigma(i),\sigma(j)} = 1]$$

Thus we see that

$$E[C] = \sum_{i=1}^{k} \sum_{j=i+1}^{k} \frac{2}{(j - i) + 1}$$

Note that

$$\sum_{i=1}^{k} \sum_{j=i+1}^{k} \frac{2}{(j - i) + 1} \leq 2k \sum_{j=2}^{k} \frac{1}{j} \leq 2k \log(k)$$

for $k \geq 2$. Thus, the *expected* running time of the randomized quick sort is $O(k \log(k))$.

Average case analysis of Quick Sort

The above analysis can *also* be seen as an argument for the *average case* running time of the *original* quick sort algorithm (where a fixed choice of $p = k$ is made for each call to `quicksort`). Since all possible inputs are characterised by all possible permutations σ, the average number of comparisons is given by

$$A(k) = \frac{\sum_{\sigma \in \mathcal{S}_k} C(\sigma)}{k!}$$

where $\mathcal{S}_k$ is the group of permutations of $[1, k]$ and $C(\sigma)$ counts the number of comparisons that `quicksort` makes when the input is the permutation σ.

Given any *fixed* permutation μ, the permutation $\mu\sigma$ runs over all permutations in $\mathcal{S}_k$ as σ runs over all permutations in $\mathcal{S}_k$. Moreover, as $\mu\sigma$ varies, we get all possible elements in the first (or last!) position of each call to `quicksort`. Thus, the above average is also the average for a *fixed* input μ over all possible choices of p in (2). However, the latter is precisely, the expected running time for the randomized algorithm for the given *fixed* input μ. Thus, the two values are the same!

In other words, we see that the average number of comparisons made by `quicksort` is $O(k \log(k))$.