15-451/651 Assignment 1

Vy Nguyen
vyn@andrew.cmu.edu
Recitation: B
January 30, 2018

1: A Good Sort

(a)

Data: an n x n matrix M containing n^2 distinct numbers and each row is sorted
Result: sorted array A of n^2 elements
Algorithm:
Create an output array A;
Make a min-heap of size n and insert the 1st element of each row into the heap;
for $i = 1...n^2$ **do**
  Get the min element from the heap and store in the output;
  Replace the root with the next element from the same row as the output element;
  Heapify and put the minimum element at the root;
end

Runtime:
The algorithm starts with n elements in the heap (the first elements from each row of the matrix). Since the matrix is row-sorted, the minimum element of the list is in the heap at this point. In fact, the element that has rank i must be in the first c columns where $c = min(i, n)$. (Why? Because there are at most $i - 1$ elements smaller than element i).

So by the ith iteration, the ith-ranked element is already in the heap. The following is an informal inductive proof. The base case for the minimum element is proven above. By the $(i-1)$th iteration, the heap already outputs the $i - 1$ smallest elements, so one of those would have led to the ith element being added to the heap.

Since the heap is a complete binary tree, it takes $O(lg(n))$ comparisons for the heap to find the minimum element and output it. Then another element from the matrix is added to the root of the heap and at worst, another $O(lg(n))$ comparisons is needed to find the minimum and place it at the root. Since there is a total of n^2 elements, the algorithm has to do at most $n^2 lg(n)$ comparisons.

(b)
Consider an unsorted array of n^2 elements. To get these elements to the row-sorted matrix form, we can pick at random n elements from the array and sort these, then repeat for the remaining elements. Since any algorithm that sort n elements must have a lower bound of $\Omega(nlog(n))$ (proven in class), the matrix can be constructed using $\Omega(n^2 log(n))$ comparisons.

Suppose there exists a comparison-based algorithm X that can solve the problem in *1a* with less than $n^2 lg(n) - O(n^2)$ comparisons. As shown above, X must first do some preliminary work of $\Omega(n^2 log(n))$ to put n^2 elements into the row-ordered matrix form.

From lecture, we proved the result which says that any comparison-based algorithm must make at least $lg(m!)$ comparisons, where m is the number of elements. So the number of comparisons that X needs to sort n^2 elements, by the Stirling's approximation, is

Vy Nguyen
vyn@andrew.cmu.edu

$$lg(n^2!) > lg\left(\frac{n^2}{e}\right)^{n^2} = n^2lg(n^2) - n^2lg(e) = 2n^2lg(n) - n^2lg(e) > n^2lg(n) - O(n^2)$$

So in total, algorithm X must do at least $n^2lg(n) - O(n^2) + \Omega(n^2log(n))$ comparisons to solve problem *1a*, a contradiction to our initial assumption. Thus, by proof of contradiction, there exists no algorithm that can solve the above problem using less than $n^2lg(n) - O(n^2)$ comparisons.

Vy Nguyen
vyn@andrew.cmu.edu

2: Cut and Merge!

(a)

Data: an unsorted array A with n distinct elements and k distinct integers
$1 \leq i_1 < i_2 < ... < i_k \leq n$

Result: k elements in A having these ranks

Algorithm:

Make a max-heap of size k and insert the 1st k elements of A into the heap (A[0...$k-1$]);
Let max = max element of heap;
for $i = k...n-1$ **do**
  if *(A[i] < max)* **then**
    Replace heap root and max with A[i];
    Heapify and put the max element at the root;
  end
  Sort k elements in heap and return;
end

Runtime:
The algorithm starts with k elements from array A in the heap. Since it takes at most $O(lg(k))$ comparisons to find the maximum element within the heap (a binary tree structure) and there are n elements in total, it would take at most $O(nlg(k))$ comparisons in total. At the end of the *for* loop, the heap now contains k smallest elements. We can sort these with $O(klg(k))$ comparisons using a sorting algorithm, for example, MergeSort. So the total number of comparisons needed is $O((n+k)lg(k) \leq O(2nlg(k)) = O(nlg(k))$.

(b)
Suppose we have two sorted lists of length m and n where $m \leq n$. The number of possible merge outcomes is the number of ways to choose m positions out from $n+m$ positions, (i.e.) $\binom{n+m}{m}$. In the worst case, we'll need $lg\binom{n+m}{m}$ comparisons to correctly merge the two lists.
From the first quiz, we have the result $\binom{a}{b} \geq (\frac{a}{b})^b$; so it follows that

$$lg\binom{n+m}{n} \geq lg\left(\frac{n+m}{m}\right)^m = mlg\left(\frac{n+m}{m}\right) = \Omega\left(mlog\left(\frac{n+m}{m}\right)\right)$$

Thus, any deterministic comparison-based algorithm for MERGE must use $\Omega\left(mlog\left(\frac{n+m}{m}\right)\right)$ comparisons in the worst case.

3: Preprocessing for Query Day

(b)
Consider a set S of n distinct elements and let the preprocessing uses $P(n) = 0$ comparison. Suppose there exists a deterministic algorithm that has a query time $Q(n) = n - c$ comparisons for some integer $c < n$ and can still output the correct answer. Then after $n - c$ comparisons, there are still c elements that have not been compared to q. Suppose an adversary chooses q to be one of these c elements. Then the algorithm would output that $q \notin S$ when in fact $q \in S$. Thus, any deterministic algorithm must have $Q(n) \geq n$ if $P(n) = 0$ to ensure correctness.

(c)
The Dart Lemma says that given some partition of the interval $[0, 1]$ using k randomly chosen pivots, the expected distance from a point $p \in [0, 1]$ to its nearest pivot is $\frac{1}{1+k}$, making the expected length of each sub-interval to be $\frac{2}{1+k}$. Extending this to a $[0, n]$ interval, the expected length of each sub-interval is $\frac{2n}{k+1}$. To achieve a preprocessing time $P(n) = \frac{1}{2}nlg(n)$, the desired bucket length is $\sqrt{n}$, and thus, $k = 2\sqrt{n} - 1$ pivots.

For the preprocessing algorithm, first randomly pick $2\sqrt{n} - 1$ elements from the input set S and sort them, then partition the rest of the elements using the pivots. Since it takes $\Omega(nlog(n))$ to sort n elements (proven in class), it takes at least $(2\sqrt{n} - 1)lg(2\sqrt{n} - 1)$ comparisons to sort the pivots. Using binary search, each of the $(n - 2\sqrt{n} + 1)$ remaining elements gets compared to the pivots at most $lg(2\sqrt{n} - 1)$. So the total number of comparisons is

$$\begin{aligned} P(n) &= (2\sqrt{n} - 1)lg(2\sqrt{n} - 1) + (n - 2\sqrt{n} + 1)lg(2\sqrt{n} - 1) && (1)\\ &= nlg(2\sqrt{n} - 1) && (2)\\ &\approx nlg(2\sqrt{n}) && (3)\\ &= nlg(\sqrt{n}) + nlg(2) && (4)\\ &= \frac{1}{2}nlg(n) + O(n) && (5) \end{aligned}$$

For the query time, using the fact that k pivots create $k+1$ sub-intervals on the real line, the $2\sqrt{n}-1$ pivots give $2\sqrt{n}$ buckets; each one is expected to have $\sqrt{n}$ elements. So searching for an element q requires doing binary search through the $2\sqrt{n}$ buckets first, which takes an expected $O(lg(2\sqrt{n}))$ comparisons, and then linear search within the correct bucket, another $\sqrt{n}$ comparisons. Thus, the total number of comparisons required to complete query q is

$$\begin{aligned} Q(n) &= lg(2\sqrt{n}) + \sqrt{n} && (6)\\ &= lg(2) + \frac{1}{2}lg(n) + \sqrt{n} && (7)\\ &= \frac{1}{2}lg(n) + \sqrt{n} + O(1) && (8) \end{aligned}$$