COS597D: Information Theory in Computer Science	October 24 and 26, 2011
Lecture 11-12	
Lecturer: Mark Braverman	*Scribe: Max Rabinovich*

1. Tournaments

Definition. A *tournament* on n vertices is a directed graph such that for each pair of vertices $u \neq v$, exactly one of the edges $u \to v$ and $v \to u$ is present. A graph of this form is said to be a *transitive tournament* if for every triple of vertices u, v, w such that $u \to v$ and $v \to w$ are edges in the graph, $u \to w$ is also an edge.

The idea is that such a graph represents a competition in which every player plays every other. The presence of the edge $u \to v$ indicates that u defeated v. If the tournament is transitive, it can be completely specified by giving its vertices in topological order: intuitively, this corresponds to listing the players in rank from highest to lowest. Note that in this case, there is a unique such ordering.

We will use Kolmogorov complexity to prove that not every tournament contains a large transitive tournament as a subgraph. More precisely, define

$$v(N) = \max\{v \ : \text{ any tournament on } n \text{ vertices contains a transitive tournament on } v \text{ vertices}\}.$$

We then have the following

Theorem 1. *If $N = 2^n$, then $v(N) \leq 2n + 1$.*

Proof First note that there are exactly $2^{\binom{N}{2}}$ tournaments on N vertices and that these tournaments can be specified as bitstrings by giving the direction of each of the arrows. Now consider the following encoding scheme P for tournaments T :

Encoding: Find a transitive tournament $S \subseteq T$ of size $v(N)$ and specify the topological order of S. Then for each pair of vertices u, v with at least one outside of S specify the direction of the arrow between them.

Decoding: For vertices u, v in S, the arrow between them is $u \to v$ if u comes before v in the encoded list and $v \to u$ otherwise. For other pairs, the direction is explicitly specified, so the tournament can simply be read off.

This gives an alternate encoding of tournaments as bitstrings. We now write $K(T \mid P, N)$ for the Kolmogorov complexity of T given N and a program (which we also call P) that performs the above encoding and decoding. As we would have with ordinary Kolmogorov complexity, we have

$$K(T_0 \mid P, N) \geq \binom{N}{2}.$$

Since T_0 can be output given its P-encoding $E(T_0)$, this implies

$$|E(T_0)| = v(N) \lg N + \binom{N}{2} - \binom{v(N)}{2} \geq \binom{N}{2}.$$

Rewriting $\lg N = n$ and rearranging, we deduce

$$\frac{1}{2} v(N)(v(N) - 1) \leq v(N)n.$$

This gives

$$v(N) \leq 2n + 1.$$

■

2. Sorting Complexity

2.1. **Complexity of Heapsort.** Heapsort is a well-known $O(n \lg n)$ sorting algorithm which uses repeated removal and insertion into a shrinking binary heap to sort a given array. In this section, we present a precise estimate of its expected running time. Before we begin, we recall the definition of a binary heap.

Definition. A *binary heap* is a binary tree each of whose nodes x stores an integer $n(x)$ and such that if y is a child of x, then $n(y) \leq n(x)$. (This definition can be extended to include any totally ordered set of values.)
By convention, a binary heap is represented as an array A such that the children of node i (which stores the value $A[i]$) are $2i$ and $2i + 1$.
The algorithm takes an array A as input and proceeds in two stages:

(1) *Heapify.* Rearrange the input array so that it forms a binary heap. This can be accomplished efficiently using recursion: scan the array from right to left, moving each entry down to its proper place as it is scanned. The cost of this operation is $O(n)$.
(2) *Sort by repeated reheapification.* On a high level, this part of the sort proceeds as follows:
for $i = n \to 1$
switch $A[i]$ with $A[1]$
rearrange $A[1 \cdots (i-1)]$ into a binary heap
end for

The rearrangement step is responsible for most of the cost, and the key issue in it is to reinsert $A[1]$ (formerly stored at index i) into the heap. There are two principal methods of accomplishing this:

Williams' method: insert from the top down, maintaining a pointer to the node currently being examined. First check if either of the children at the root is larger than the value to be inserted. If not, store the new value at the root. Otherwise, update the pointer to point to the child node holding the larger value and use the same insertion procedure into the binary tree rooted at that child. Repeat until neither of the child values is larger than the new value. At the end, move each of the values along the insertion path up into its parent node and store the new value in the end node.

Floyd's method: insert from the bottom up, maintaing a pointer to the node currently being examined. First move down the heap from the root, always following the link to the child with the larger value among the two. Upon reaching a leaf, move back up this path until the parent of the current node stores a value larger than the new value. Then move all the values along the path from the root to this final node up into their parents (as in Williams' method) and store the new value at the current node.

A more detailed description of both methods, including Java code for each, can be found in Sedgewick and Wayne, *Algorithms*.

If the new value is ultimately stored at depth d, Williams' method requires $2d$ comparisons, while Floyd's method requires $2 \lg n - d$. Both methods make $2d$ data movements. On average, we expect d to be close

to $\lg n$, simply because most of a binary tree is near the bottom, so Floyd's method should, in principle, be more efficient. This is indeed the case, as the following theorem shows.

Theorem 2. *With probability $p_n \to 1$ and also on average, heapsort makes $n \lg n + O(1)$ data movements using either Williams' or Floyd's method and makes $2n \lg n - O(n)$ comparisons using Williams' method and $n \lg n + O(1)$ comparisons using Floyd's method.*

Proof (Sketch.) We can completely ignore the initial heapification step, as it is $O(n)$. Now let d_i be the depth of i^{th}insertion. By what we have already said above, heapsort makes $\sum d_i$ data movements using either insertion method. On the other hand, using Williams' method, $\sum(2d_i) = 2\sum d_i$ comparisons are made, while $2n \lg n - \sum d_i$ are made using Floyd's method. It therefore suffices to show $\sum d_i \geq n \lg n - O(n)$.

For this, as a matter of notation, let $\delta_i = \lg n - d_i$. Since there are $n!$ permutations on n elements, all but an exponentially small proportion of them satisfy

$$K(\pi \mid n,\, A,\, P) \geq n \lg n - 2n,$$

where P denotes a decoding program which takes as input the new location of $A[i]$ after reinsertion and outputs the initial heap and A is the sorted array. Now, since the initial heapification procedure can be specified in $O(n)$ bits, if h denotes the initial heap, we must have

$$K(h \mid n,\, A,\, P) \geq n \lg n - O(n).$$

On the other hand, the input to P for the i^{th} step can be given by writing down the path from the root to the storage node, using say 0 to denote a left move and 1 to denote a right one, and terminating that piece of the input by writing a prefix-free binary represenation of δ_i. This can be done in $d_i + 2\lg \delta_i = \lg n - \delta_i + 2 \lg \delta_i$ bits. Since this input is sufficient to obtain h from n, A, P, we must have

$$n \lg n - O(n) < n \lg n - \sum(\delta_i - 2 \lg \delta_i)$$

so that $\sum \delta_i - 2 \lg \delta_i = O(n)$. A simple calculation implies the same for $\sum \delta_i$, which shows $\sum d_i = n \lg n - O(n)$, as required. ■

A detailed proof of Theorem 2 can be found in (Li, Vitányi, "An introduction to Kolmogorov complexity", 3rd ed., Section 6.6).

2.2. **Complexity of Shellsort.** Shellsort is another well-known sorting method, which is, however, less efficient than heapsort. We begin by recalling its definition.

Definition. An *h-chain* in an array A is the subarray of all entries at indices $i = qh + r$ where q is allowed to vary and $0 \leq r < h$ is fixed.

Shellsort with p passes using gaps $(h_1, \ldots, h_p)$ proceeds by iteratively sorting each of the h_j-chains for each $1 \leq j \leq p$, using any sorting method (say insertion sort). We require $h_p = 1$ so that the final pass completely sorts the array. (A fairly detailed discussion can be found in Sedgewick and Wayne, *Algorithms*.) The idea is that each pass makes the array more sorted so that the next pass takes less time than it would in the worst case. Although the worst case running time is still $O(n^2)$, we have the following facts.

Fact ([Pratt]). *If each gap h_j is of the form $2^a 3^b$, Shellsort has time complexity $O(n \lg^2 n)$.*

And in the other direction:

Fact ([Knuth]). *If $p = 2$, the optimal choice of gaps gives an average running time of $\Theta(n^{5/3})$.*

As with heapsort, Kolmogorov complexity provides a way to obtain a general estimate on the average running time of Shellsort.

Theorem 3 ([Li, Vitányi]). *With probability $p_n \to 1$ and also on average, any p-pass Shellsort takes $\Omega(pn^{1+\frac{1}{p}})$ steps.*

Proof If $p = \Omega(\lg n)$, the bound reduces to $\Omega(n \lg n)$, so we may take $p = o(\lg n)$. Now, all but an exponentially small set of permutations π on n letters satisfy

$$K(\pi \,|\, n,\, A,\, P) \geq n \lg n - 2n,$$

where A is the sorted array and P is the decoding program we describe below.

For each $1 \leq i \leq n$, $1 \leq k \leq p$, let $m_{i,k}$ denote the signed distance the key at position i moves in the h_k-chain containing i. Then define,

$$M \;:=\; \sum_{i,k} |m_{i,k}|$$

and note that this is exactly the number of data movements made by the sort. Observe further that there is a program P reconstructing the original permuted array from the sorted array and the sequence of $m_{i,k}$ in lexicographical order (by indices). A few minutes' thought is sufficient to see that the number of possible signed sequences of length pn with sum (in absolute value) bounded by M is

$$D(M) = \binom{M + pn - 1}{pn - 1} \cdot 2^{pn}.$$

We use the asymptotic estimate $\binom{a}{b} < \frac{a^b}{b!} \lesssim (\frac{a}{b})^b e^b$ to find

$$\lg D(M) \lesssim pn \, \lg(1 + \frac{M}{pn-1}) + O(pn).$$

Since all the information in the sequence can be specified using $\lg D(M)$ bits, we deduce finally that if M is large enough that more than an exponentially small fraction of permutations can be sorted in fewer than M steps, we must have

$$\lg(1 + \frac{M}{pn-1}) \gtrsim \frac{n \lg n - 2n - O(pn)}{pn} = \frac{\lg n}{p} - O(1).$$

Unraveling this gives

$$M \;\gtrsim\; pn^{1+\frac{1}{p}} - O(pn),$$

which proves the claim. ■

3. Remark: Prime Number Theorem via Kolmogorov Complexity

For $n \geq 1$, let $\pi(n)$ denote the number of positive primes $\leq n$. The following asymptotic characterization of this function is a celebrated result of analytic number theory.

Theorem 4 (Prime Number Theorem). $\pi(n) \sim \frac{n}{\ln n}$.

This result is usually proven by involved analytic arguments. However, we can get quite close to it by fairly elementary Kolmogorov complexity considerations. More precisely, consider a recursive encoding E of integers n defined by

$$E(n) = (m, E(\frac{n}{p_m}))$$

where m is the index of the largest prime dividing n. Roughly speaking, the idea is that we should not be able to save much this way, and with this sort of argument we can find

$$\pi(n) \gtrsim \frac{n}{\ln n \cdot L(n)}$$

where L is $(\log \log n)^2$ (and can be further reduced slightly). This is quite close to the actual Prime Number Theorem and significantly easier to prove.