Entropy and Complexity

Devavrat Shah[1], Mayank Sharma[2]
Basic Research Institute in the Mathematical Science
HP Laboratories Bristol
HPL-BRIMS-2000-32
December 20th, 2000*

E-mail devavrat@cs.stanford.edu, msharma@standford.edu

information theory;
algorithms

Computational complexity of algorithms for solving problems has been at the heart of theoretical computer science. Traditionally, the computational cost of an algorithm is estimated by "counting" operations combinatorially depending on the algorithm. We present a very different method for estimating cost of solving problem, incorporating ideas from information theory. Algorithms can be viewed as "search" procedures on the input (output) space. This naturally makes computational cost of algorithm as function of "entropy" of input (output) distribution. The relation of computational complexity and "entropy" of distribution depends on the particular "operations" used by algorithm to solve problem. This particular mapping between computational scale and "entropy" scale is independent of problem. We demonstrate the use of this method in classical "searching" and "sorting" problems. We work out many different searching and sorting algorithms' complexity using this method. In process, we also come up with new algorithms to solve some cases of searching and sorting that are aware of input distribution.

[1] Computer Science Department, Stanford University, Stanford, CA
[2] Department of Electrical Engineering, Stanford University, Stanford, CA

Entropy and Complexity

Devavrat Shah
Computer Science Department,
Stanford University.
devavrat@cs.stanford.edu

Mayank Sharma
Department of Electrical Engineering,
Stanford University.
msharma@stanford.edu

Abstract

Computational complexity of algorithms for solving problems has been at the heart of theoretical computer science. Traditionally, the computational cost of an algorithm is estimated by "counting" operations combinatorially depending on the algorithm. We present a very different method for estimating cost of solving problem, incorporating ideas from information theory. Algorithms can be viewed as "search" procedures on the input(output) space. This naturally makes computational cost of algorithm as function of "entropy" of input(output) distribution. The relation of computational complexity and "entropy" of distribution depends on the particular "operations" used by algorithm to solve problem. This particular mapping between computational scale and "entropy" scale is independent of problem. We demonstrate the use of this method in classical "searching" and "sorting" problems. We work out many different searching and sorting algorithms' complexity using this method. In process, we also come up with new algorithms to solve some cases of searching and sorting that are aware of input distribution.

1 Introduction

An algorithm takes some input, processes the input depending on the problem structure and finds output corresponding to this input. Thus an algorithm can be viewed as a function that takes input and maps it to output. Let us suppose that we can actually store the whole table of this function that captures the relation between inputs and outputs in the memory. Even in such simplified scenario, algorithm need to find the positions corresponding to each input. Or equivalently, we need to find the "code" corresponding to input that resembles to "address" in the table of (input,output). To do so, an algorithm uses some operators depending on the problem structure as well as knowledge of algorithm designer. For example, in case of sorting, "comparing two elements" is an operation. From above it can be deduced that, the complexity of problem is at least as much as "searching" for proper output for given input, using available operations.

If distribution on input elements and hence on output elements is known, then it can be incorporated in design of efficient algorithm. The distribution implies the "uncertainty" in output generation. Operations of algorithm help in removing this uncertainty by identifying subspace of the whole output space, incrementally. When an algorithm gets over, the exact output is known. Hence there is no uncertainty left. Generally uncertainty is measured by "entropy" of distribution. In that sense, algorithm starts with entropy of input(output) distribution and ends with zero entropy. So using the amount of entropy reduction obtained by operations used in algorithm, it is possible to predict approximately how many times an operator needs to be used, and hence predict approximate running time. Formally, let X be a random input, let A be operator applied on it and let Y be the processed input by operator A. Since operator is deterministic, to formulate it as a random variable, consider the "random" action it will take, induced by the randomness of X. Hence, if we consider A as such a "random" variable, then information of Y and A together can help us recovering X. Hence,

$$\begin{aligned} & H(Y,A) = H(X) \\ \Rightarrow\ & H(Y) + H(A|Y) = H(X) \\ \Rightarrow\ & H(A|Y) = H(X) - H(Y) \\ & \qquad = \text{Entropy Reduction} \end{aligned}$$

By property of $H()$, we know,

$$\begin{aligned} H(Y) + H(A|Y) &= H(A) + H(Y|A) \\ \Rightarrow H(A|Y) &= H(A) + (H(Y|A) - H(Y)) \end{aligned}$$

Hence, if Y is not dependent on A, then, we can say that, $H(A) \approx H(A|Y)$, and hence entropy reduction is like $H(A)$, the entropy reduction that an operator can obtain. Further, if algorithm starts with distribution of entropy h bits, and if $H(A)$ is say amount of entropy operator can reduce, then, $h/H(A)$ is the minimum number of times operations needs to be performed in order to find the output.

Putting it together, we can get a sort of "bijection" between computational "time scale" of algorithm and "entropy scale" of input(output) distribution. So if we can estimate the entropy reduction of operators, we can get estimate of time complexity of algorithm. Extending this further, by knowing upper bound on "entropy reduction" of operators, we get lower bound on time complexity.

From above discussion, reader can realize that, searching is canonical case where the method should be applicable. In the next section, this method is applied to analyze searching algorithms. In the section 3, this method is extended for the case of sorting. We finally discuss further work and some directions.

2 Searching

By definition, the problem in searching is : given n elements one need to find an element with particular property. Depending on property and available structure in the collection of points, "search" can be performed quickly. Searching is canonical example where distributional uncertainty or entropy of input query distribution should determine the complexity of searching that particular point out of n points. We consider searching under different properties and see how known algorithms behave from the point of view of removing distributional uncertainty.

2.1 Searching among ordered collection of points

Consider that, n ordered elements, $1, \cdots, n$ are given, and they are sorted in that order. Say they are positive real numbers, $a_1 < \ldots < a_n$. As an input a query q, some real number, is given. The task is to find out which element in these n elements is nearest to query value q. So for example, let $n = 4$, and ordered numbers are (1.2, 2.3, 4.6, 9.1) and say $q = 4.4$, then output should be 3, since 4.6 is nearest in value to q. Let us assume, for simplicity, that with almost zero probability q's value will be "exactly" equal to one of the a_ks. Note that, here allowed operation is picking any of n positions, and comparing it with query q.

Let us consider the distribution of input queries. Let $p_k = \Pr(\text{query's answer will be } k)$. Thus $(p_k)_{k=1}^n$ gives out the whole distribution of the outputs that will be generated. Hence the uncertainty in output generated, or, formally entropy of this discrete distribution is $H(p) = -\sum_{k=1}^n p_k \ln p_k$. Even if we manage to code these numbers by optimal coding schemes like huffman coding, $H(p)$ is lower bound on expected number of comparisons required for searching. To see the same thing other way, note that operation of querying and comparing k^{th} elements gives information whether $q \leq a_k$ or $q > a_k$. This will reduce the search space from n elements to k or $n - k$ elements. The probability mass on left of k^{th} element, including it, is $\sum_{j=1}^k p_j = l_k$, and hence in the space of first k elements, now the probability of each element becomes : $p_j/l_k, 1 \leq j \leq k$. Thus at the end of querying k^{th} element, with probability l_k, the remaining entropy of distribution is :

$$
\begin{aligned}
& -\sum_{j=1}^{k}(p_j/l_k)\ln(p_j/l_k) \\
& = -(1/l_k)\sum_{j=1}^{k} p_j \ln p_j - (\ln l_k/l_k)(\sum_{j=1}^{k} p_j) \\
& = -(1/l_k)\sum_{j=1}^{k} p_j \ln p_j - (\ln l_k)
\end{aligned}
$$

and with probability $r_k = (1 - l_k)$, the remaining entropy is :

$$\begin{aligned}
&- \sum_{j=k+1}^{n} (p_j/r_k) \ln(p_j/r_k) \\
&= -(1/r_k) \sum_{j=k+1}^{n} p_j \ln p_j - (\ln r_k/r_k)(\sum_{j=k+1}^{n} p_j) \\
&= -(1/r_k) \sum_{j=1}^{k} p_j \ln p_j - (\ln r_k)
\end{aligned}$$

Thus expected remaining entropy is :

$$\begin{aligned}
&-\sum_{j=1}^{n} p_j \ln p_j - l_k \ln l_k - r_k \ln r_k \\
&= H(p) - l_k \ln l_k - (1 - l_k) \ln(1 - l_k)
\end{aligned}$$

This shows that, the expected entropy reduction is

$$-l_k \ln l_k - (1 - l_k) \ln(1 - l_k)$$

which is maximized when $l_k = 1 - l_k = 1/2$, and then its 1 bit. So if one can achieve 1 bit reduction every time given the distribution, then that is the *optimal* strategy for searching using comparison. Also, since every operation reduces entropy only by one bit at best, at least $H(p)$ number of steps are required.

Suppose that, the distribution given is uniform, that is, all points are likely to be output with probability $1/n$. In that case, the entropy is $\ln n$ bits. The standard algorithm for search is *binary search* : *Every time, query the "midpoint" of the active interval of elements and depending on answer, either makes left or right half as active interval. Continue till interval size reduces to 1.*
Note that, due to uniform distribution, this exactly gives entropy reduction of 1 bit at every comparison, the maximum possible reduction. Hence, this is an optimal algorithm for searching under uniform distribution.

In general, say any distribution p is given. In that case, what should be an optimal algorithm ? In literature, this is known as problem of *alphabetic coding.* Known algorithm for alphabetic coding, for example the one by yeung [8], achieves expected time for search lesser or equal to $H(p) + 2$. This is quite near to optimal search time of $H(p)$, that can possibly be achieved. Note that, under special cases when each of the probability is power of $1/2$, exact bounds are achieved by the very same algorithm.

From over above observations, we can suggest a simple greedy heuristic. Every time, compare with the point in the interval, such that it divides the probability mass into half or as near to half as possible in both sides. Equivalently, query the point by which the entropy reduction is as near to 1 bit as possible. Note that, if sequence of probabilities are such that this is always possible, then this heuristic matches the optimal algorithm.

Next we look at another variation of searching, where different search operations have different prices.

2.2 Searching with "priced" queries

Consider the model where quering different elements have different and let us suppose input distribution is say uniform, and goal is to minimize the expected cost of the algorithm. The issue here is minizing total expected cost under heterogeneous query-cost model. One might be tempted to go for greedy algorithm that picks up the query which has maximum entropy reduction per unit cost. But that is not the optimal algorithm, as one can easily come up with counter example. To solve this optimally, a dynamic programming algorithm running in $O(n^2)$ can be used. Let us denote, $C(i,\ldots,j)$ as optimal expected cost for searching an element that comes uniformly from one of the $\{i,\ldots,j\}$ elements with query-cost c_k for querying k^{th} element. Also for notation, $C(i,i)=0$. Then we can write, recursively,

$$C(i+1,\ldots,i+m)=\min_{k=1}^{m}\{c_{i+k}+(k/m)C(i+1,\ldots,i+k)+(1-k/m)C(i+k,\ldots,m)\}$$

Instead of uniform if the probability distribution on querying were different, and say, $p=(p_k)_{k=1}^{n}$ is the probability distribution. Let $l_k=\sum_{j=1}^{k}p_j$. Then we could re-write the above recursive equation as,

$$C(i+1,\ldots,i+m)=\min_{k=1}^{m}\{c_{i+k}+l_kC(i+1,\ldots,i+k)+(1-l_k)C(i+k,\ldots,m)\}$$

Note that, the probability distribution while computing, $C(a,b), 1\le a\le b\le n$, gets normalized in order to make the probability on (a,b) adding up to 1. Note that, $C(1,\ldots,n)$ gives the minimum expected cost and finding exactly the minimizer values for each of the recursive stage, we get the complete tree of the queries to be made. Note that, this evaluation requires $\theta(n^2)$ operations.

The interesting thing to note here is : optimal algorithm becomes of second order in n, due to new dimension of price along with probability distribution.

2.3 Near Optimal search

In [7], Charikar et. al. present near-optimal algorithm. We can view that algorithm as generalized version of greedy heuristic that tries to query point having maximum entropy-reduction/cost. The algorithm presented by them assumes uniform distribution on the available query points. We notice that, the same algorithm can be extended with little change to case where any distribution over query points is possible. We present the description of the algorithm here.

Let r and c be some constant parameters. Let $p = (p_k)_{k=1}^{n}$ be probability distribution of q, the query point, with $p_k = \Pr(q = k^{th}\text{point})$. Initially costs of query points are grouped by considering all points with costs in $[r^{j-1}, r^j]$ in group j, with rounded-cost r^j. The costs can be normalized so that minimum cost is 1. The algorithm maintains a search interval I, which is the set of possible (contiguous) locations where q could lie, and split I into three (contiguous) intervals L, M, R where the left and right intervals L and R respectively dont not contain any element of (the current) group j and the middle interval M, referred to as the *effective interval*, which begins and ends with an element of group j. The algorithm maintains the property that I does not contain any elements of groups $j-1$ or lower. We repeatedly compare q with the group j element that divides M such that, on either side the total probability mass of M is divided closest to $1/2$. Such comparisons are called *regular* comparisons and each such comparison is guaranteed to halve the probability mass of the effective interval. This certainly makes progress as long as the element q lies within the effective interval. However, if q does not belong to the current group j, at some point q could fall outside the effective interval for group j. In such a case, we do not want to spend too much on querying group j elements. To handle this possibility, after every c regular comparisons an extra check is performed with boundary of the effective interval. If current search interval I does not contain anyelement from group j, we move to group $j+1$.

The above description is almost complete. Without formally stating the complete algorithm, we go onto analysis of this algorithm and stating some facts about the algorithm. Consider the following :

Claim 1 : Every regular comparison, reduces the probability mass of the effective interval at least by half.

Claim 2 : Total number of comparisons done of cost order j, c_j, is bounded above by $(1 + 1/c)\ln(p_j/p_{j+1}) + c + 2$.

From this, we state the following lemma.

Lemma 1. *For r, c, the competitive ratio of the algorithm is bounded above by,* $(1 + 1/c)r\ln(1/p) + (c+2)r^2/(r-1)$, *where p is say minimum nonzero probability of any query point.*

Proof. Let group m be the last group examined by the algorithm. Then the cost of the algorithm is at most,

$$\sum_{j=0}^{m} r^j c_j \quad \leq \quad \sum_{j=0}^{m} r^j((1+1/c)\ln(\frac{n_j}{n_{j+1}} + c + 2)$$

$$\begin{aligned}
&= (1+1/c)\sum_{j=0}^{m} r^j \ln(\frac{p_j}{p_{j+1}}) + (c+2)\sum_{j=0}^{m} r^j \\
&\leq (1+1/c) r^m \ln(1/p) + (c+2)\frac{r^{m+1}}{r-1}
\end{aligned}$$

Where, p is the probability of the last significant effective interval. The optimal proof has to be of cost r^{m-1}. Hence the competitive ratio of the algorithm is bounded above by

$$(1+(1/c))r\ln(1/p) + (c+2)r^2/(r-1)$$

□

By choosing different values of r and c we get different competitive ratio and one can optimize over that.

We now briefly look at searching problem in general and identify possible relation between time complexity and entropy of element to be searched.

2.4 Unordered Search

In general, there is no order available between elements among which we would like to search. Let, $p = (p_k)_{k=1}^{n}$ be probability distribution for output. For simplicity assume that, query matches exactly to one of the n point's value. Since by comparing with any of the n points, the information obtained is only regarding that point, the entropy reduction is just the difference of entropy before and the entropy conditioned on the information known after comparison. So if the distribution is uniform, then expected entropy reducion will be $(1/n)[k\ln k - (k-1)\ln(k-1)]$, at the k^{th} stage, that is, when the output point can be any of the k points, or equivalently, at the end of $(n-k)$ stages, the expected entropy is $(k/n)\ln k$. This shows that, expected time of searching is $\theta(n)$ as known.

3 Sorting and Entropy

Sorting involves the following : As input n elements are given which are from completely ordered class in arbitrary order. Goal is to order them and output the ordered list of these elements. This is equivalent to identifying the permutation of the original order in which input was presented, and then putting elements into right position once identification is done. For example, (1.2 4.2 0.4 5.1) corresponds to permutation (2 3 1 4), and sorting outputs (0.4 1.2 4.2 5.1).

We consider the algorithms for sorting that uses only : (a) comparison of two elements, and (b) swapping them if required. In identification of permutation, only comparison is useful, so we only concetrate on comparison operation.

Let us represent input as permutation π for simplicity over n elements. Let us consider, what information does comparison gives or what uncertainty does comparison remove. Elements of positions i and $j \neq i$ could be relatively of any order, that is, either, element of position i could be greater or smaller compared to j^{th} element. But after comparision, this uncertainty is reduced since it is exactly known that what the relative order is. Let α be probability that $\Pr(\pi(i) > \pi(j))$ for the input distribution over the permutation space. Then the uncertainty regarding these two positions is $H(\alpha) = -\alpha \ln \alpha - (1-\alpha)\ln(1-\alpha)$. At the end of comparison it is lost and hence reduction of uncertainty is $H(\alpha)$. The maximum amount can be 1 when $\alpha = 1/2$. This says that, if the input distribution over the space of permutations has entropy H bits, then the expected number of comparisons are at least H to determine the input. If input is n numbers long, then there are effectively $n!$ possible permutations. If all permutations are equally, then initially the uncertainty or entropy of space is : $\ln n!$ bits. Hence, from above discussion, $\ln n! \approx n \ln n$(stirling), is lower bound on the number of comparisons to be done, by any algorithm.

We use this formalization to actually analyze some of the known algorithms of sorting.

4 Entropy change for Sorting Algorithms

In this section, we analyze how the entropy changes in some of the known comparison based sorting algortihms. Classical analysis of these algorithms can be found in [4].

4.1 Merge Sort

Merge Sort : Algorithm

- Divide the list exactly in the middle, creating two sublists.
- If list is of size ≤ 2, trivially sort using a comparison
- If not, sort left list and right list recursively. After they are sorted, merge these two sorting lists. This merging is done by just comparing heads of the "non-merged" part of lists, and putting smaller (larger) element in the sorted list. Note that, this take number of comparisons between $[l, 2l]$, where l is size of the two equal sublist.

Next, we would like to see how the entropy is reduced as the algorithm proceeds. We assume, uniform distribution on input. Note that, the comparisons are done only when the list size is either ≤ 2 or when there is merging taking place. When list size is ≤ 2, the both elements of that particular position could have been equally likely in any relative order. So by comparison of that kind, the entropy is certainly reduced by 1 bit. For comparisons done during merging, again, since two sorted sublists that are merged, are completely "independent" in the sense that, elements from two different sublists are not compared before till they are merged. During merging, at every stage, both elements are equally likely to have any relative order. Hence such comparisons reduce entropy by 1 bit. Thus in merge-sort, all comparisons done, reduce the entropy by 1 bit. So from discussion for comparison based sorting algorithms in previous section, this is an optimal sorting algorithm. Since this algorithm exactly reduces 1 bit, it takes exactly, $\theta(n \ln n)$ comparisons irrespective to permutation.

4.2 Insertion sort

Insertion sort : Algorithm

- It sorts elements incrementally. At some stage, say first k elements of inputs are relatively sorted. Now consider $k+1$st element.
- Place the $k+1$st element, in this sorted list of first k elements. This gives first $k+1$ elements sorted.
- Proceed till all n elements are sorted.

Note that, by stage k, first k elements are sorted with respect to each other. Under uniform distribution on inputs, these first k elements are likely to have any of the $k!$ relative order equally likely. Thus by end of stage k, $\ln k!$ entropy is reduced. At $k+1$st stage, the $k+1$ element is equally likely to be at any of the position in sorted list of k elements. Thus the uncertainty of $k+1$st element's position is $\ln(k+1)$. At the end of the insertion, it is reduced, and hence reduction at the end of stage $k+1$ is $\ln k! + \ln(k+1) = \ln(k+1)!$. Note that, this involves, searching for position of $k+1$st element in sorted list of k elements. As noted before, this involves $\ln(k+1)$ comparisons. But only finding positions is not sufficient since we have to insert it at that position. Since this is done via linked list, on average it takes $\theta(k)$ comparisons. At the end of stage n, we get entropy reduced by $\ln n!$ and that involves $\theta(\sum_{k=1}^{n} k) = \theta(n^2)$ comparisons.

4.3 Few Other

Before going to some more interesting case, we note that, using the similar arguments, we can analyse other classically known algorithms like, *bubble sort,heap-sort* etc. In interest of not making article boring, we skip their analysis here. We would like to note that, analysis of these algorithms is not complicated.

4.4 Randomized Quick Sort

Randomized quick sort is a very interesting algorithm. Though worst case performance of this algorithm is $O(n^2)$, with high probability it works well, that is performance is $O(n \ln n)$. This is classically known and well studied [2] [1][3]. We analyze randomized quick sort using our approach and get the same results for expected running time as in [3] and for *concentration* around mean as in [1] with little analysis; which is much cleaner compared to classically known proof.

RandQS : Algorithm

- Choose a position i uniformly at random from all elements, say n. Let y be the element at position i.
- Let L and S are two lists which are empty intially. Start comparing elements starting from position 1 to n, in that order, with y. For any k, when it is compared with y, if it is smaller than y, it is appended at the end of the S else it is appended at the end of L.
- When all n are compared, sort the L and S recursively and merge them by placing y in middle of L and S.

The element y chosen, call "pivot", is likely to be of relative order between $\{1, \ldots, n\}$ with equal probability $1/n$. If y is, k^{th} element then there are $k-1$ elements in S-list and $n-k$ elements in L-list, and there is any relative order possible among the elements of S and L. Thus if input is distributed uniformly over all permutations, then initial entropy is : $\ln n!$, while, after choosing k^{th} element, the entropy left is $\ln k! + \ln(n-k)!$. Thus entropy reduction is : $\ln n! - \ln k! - \ln(n-k)!$, with probability $1/n$. So expected probability reduction is :

$$\begin{aligned}
\text{Entropy Reduction} &= \sum_{k=1}^{n} \frac{1}{n}[\ln n! - \ln k! - \ln(n-k)!] \\
&= \frac{1}{n}\sum_{k=1}^{n}[n \ln n - k \ln k - (n-k)\ln(n-k)] \\
&= \sum_{k=1}^{n}(k/n + (n-k)/n)\ln n - k/n \ln k - (n-k)/n \ln(n-k) \\
&= \sum_{k=1}^{n}[-(k/n)\ln(k/n) - (1-k/n)\ln(1-k/n)] \\
&= n\{\sum_{k=1}^{n}\frac{1}{n}[H(k/n)]\} \\
&\approx n\{\int_0^1 H(x)dx\} \\
&= n(1/2)
\end{aligned}$$

Hence the expected decrease in the entropy is $n/2$ bits. At the next stage, we have two RandQS running, one on size k and other on size $n-k$. Hence the expected reduction of entropy for that case is $k/2+(n-k)/2 = n/2$. This stage is completely independent of the previous stage. Thus, the expected entropy removed of every stage is $n/2$ bits while the number of comparisons done is n. Since net entropy is $n \ln n$, it takes around $2 \ln n$ stages, with each costing n comparisons. Hence we have expected number of comparisons : $2n \ln n$.

Further, we can use the above formulation to prove that the cost of RandQS is *concentrated* around $2n \ln n$. By $X_j, j \geq 1$, we denote the entropy reduced in stage k, by RandQS. Let $S_k = \sum_{j=1}^{k} X_j$. Let RandQS takes more than $2n \ln n + cn$, comparisons. That is, it takes, c more stages compared to $2 \ln n$ stages, or rather, $2 \ln n + c$ stages end up removing entropy $n \ln n$ bits. Or putting it other way, say some $k = 2 \ln n + c$, i.i.d. random varibles, add up to value which is $cn/2$ away from their aggregate mean, say $k\mu$. We would like to find the probability of this event, which can be written as :

$$\begin{aligned} \Pr(|\text{Comparisons in RandQS} - 2n \ln n| \geq cn) &= \Pr(|S_k - k\mu| \geq cn/2) \\ &= \Pr(|\frac{1}{k} S_k - \mu| \geq cn/2k) \end{aligned}$$

These random variables are nice and well-behaved. So they have large deviation rate function, say $I()$, which is convex, lower semicontious and it's fretchet-legendre transform of the log-moment generating fuction of i.i.d. random variable, which have mean $\mu = n/2$. Applying Cramer's theorem (see [6] for reference) for $\tilde{S}_k \overset{=}{\triangle} S_k/k$ (or to get only upper bound for all n, use Chernoff's bound), and we get,

$$\begin{aligned} \Pr(|\tilde{S}_k - \mu| \geq cn/2k) &\approx exp(-I(cn/2k)k) \\ &= n^{-I(cn/2k)}, \text{Since, } k = O(\ln n) \end{aligned}$$

We dont compute rate function $I()$ here, but it is easy to compute since we explicitly know the distribution of random variable. The purpose of this is to show that, using this method we can very simply show that RandQS has concentration in general, (no conditions assumed !) as proved in [1]. We would like to note that, the known proof of this fact [1], uses quite complicated argument involving bounded-martingale inequality and other things.

5 Lower bound of Sorting Algorithms

In this section, we demonstrate how the lower bound of running time of algorithms can be obtained using this method. In particular, we get non-trivial lower bounds for p shell sort algorithms' running time using this method.

5.1 Lower bound on p-shell sort

We would like to note that, exactly same lower bound on running time of $p-$shell sort algorithm are obtained by Jiang et. al. in [9] using ideas of incompressibility of problem instance and kolmogorov complexity. We note that, the proof we will present here turns out to be simpler and more intuitive.

We first describe p-shell sort algorithm.

p-Shell Sort : Algorithm

- Algorithm runs for p stages. At any stage $t \leq p$, n elements are divided into disjoint $\ell(t) \overset{\triangle}{=} \lceil n/k_t \rceil$ sublists, each of size k_t, where, $(k_t)_{t=1}^{p}$ are predetermined parameter.
- Sort the $\ell(t)$ sublists seperately using insertion sort on them, and replace the sorted elements back into n positions.
- Do this for all stages one after other. The way algorithm selects parameters, at the end of stage p, for any permutation, all elements are sorted.

For example, consider that size of input is 4. We consider $3-$stage shell sort algorithm. Let $k_1 = 2, k_2 = 2, k_3 = 2$. Let permuation is (4 2 1 3). In first stage, we have, first list contains first two elements (4 2) and second list (1 3). After sorting them, we get (2 4 1 3). Next stage we have sublists : (2 1) (4 3). Then we get, (1 3 2 4). Finally we have, (1 4) and (3 2), which yields completely sorted : (1 2 3 4).

Naturally question arises, what is the lower bound on the comparisons done by such $p-$shell sort algorithm. This question was known since early 1960s, and there was no non-trivial lower bound known till [9] by Jiang et. al. We will obtain similar bounds using our method.

Let p stages have list lengths $k_1, \cdots, k_p$. At any stage, since it uses insertion sort over every sublist, if sublist size is k, then $\theta(k^2)$ comparisons performed in expected sense and entropy is decreased by, $\ln(k\!) \approx k \ln k$ bits. Furthermore, insertion sort has property that, if it reduces entropy by $k \ln c$ bits, for k sized input, then it must have spent at least kc comparisons. Since at a stage, when the sublist size is k, there are n/k such lists. The net entropy reduced is $k \ln k * n/k = n \ln k$ and comparisons done are $n/k * k^2 = nk$, on average. So if $k_1, \cdots, k_p$ are each stage's sublist lengths, then the entropy reduction is :

$$\begin{aligned}
\text{Entropy Reduction of p stages} &= n(\ln k_1 + \ldots + \ln k_p) \\
&= n(\ln(\prod_{j=1}^{p} k_j))
\end{aligned}$$

and the number of comparisons done on average,

$$\text{Average Comparions} = n(k_1 + \ldots + k_p)$$

At the end of p stages, since all inputs are sorted, the entropy reduction should be net entropy reduction of input, which is $\ln n! \approx n \ln n$. Hence we get,

$$n(\ln(\prod_{j=1}^{p} k_j)) = n \ln n$$

$$\Rightarrow (\prod_{j=1}^{p} k_j) = n$$

By well-known arithmetic-geometric inequality,

$$(\sum_{j=1}^{p} k_j) \geq p(\prod_{j=1}^{p} k_j)^{1/p}$$

$$\Rightarrow (\sum_{j=1}^{p} k_j) \geq p(n)^{1/p}$$

$$\Rightarrow n(\sum_{j=1}^{p} k_j) \geq p(n)^{1+1/p}$$

Hence, we conclude that,

$$\text{Average Comparisons Required} \geq pn^{1+1/p}$$

Thus, we get lower bound of $pn^{1+1/p}$ for the number of comparisons done by $p-$shell sort.

Note that, more careful reader might come up with question of using entropy reduction of $k \ln k$ for a sublist at any stage, since at any subsequent stage, uniform distribution wont be present. But as noted before, for insertion sort, if kc are operations done, the maximum entropy removed is $k \ln c$, and hence the lower bound, follows exactly as above.

6 Distribution aware sorting algorithms

Till now, we considered that distribution is uniform for input over permutations. In this section, we consider any general distribution. For such situation, we try to obtain algorithms optimized with respect to given input distribution.

6.1 Quicker Quick Sort, given Information

In this section, we consider the "optimal" version of quick sort that will be aware of input distribution.

Let us consider, what does quick sort do at each stage, and that will itself explain how should it be in case of awareness of input distribution. At each stage, quick sort picks up an element as a "pivot", which creates two sublist, one of smaller elements and the other of larger elements. The choice of "pivot" governs how much of entropy is reduced at the cost of those n comparisons. The optimal situation would be the one where entropy reduction is maximized in expected sense at each step.

We can give the following *optimal* quick sort algorithm, optimized with respect to expected number of comparisons made by algorithm given the input distribution. The order in which algorithm should pick up "pivot" element of n unknown elements can be determined by a dynamic programing type algorithm. Note that, once given this order based on input distribution, its usual quick sort.

Let I be input distribution given on S_n, the permutation group. Let us denote $H(i, i+1, \cdot, j|I(i, \ldots, j)), i \leq j$, the minimum number of expected comparisons required given input distribution I restricted to only relative permutations of elements $(i, \cdots, j)$. Let q_{ij} be probability that element at position i is j^{th} element in relative order. (eg. in (213), element at position 1 is 2). Now denote $V_i(1, \cdots, n) = \sum_{j=1}^{n} q_{ij}[H(1, \cdots, j-1|I(1, \ldots, j-1)) + H(j+1, \cdots, n|I(j+1, \ldots, n))]$. This is expected number of comparisons other than n needed to be done if element at position i is chosen. Define $S^* \stackrel{\triangle}{=} \{k : V_k \leq V_j, \text{for} j = 1, \cdots, n\}$. If $|S^*| > 1$, pick any element of S^* uniformly at random and call this as i^*, the optimizing choice. Note that, in uniform distribution case, $V_1 = \cdots = V_n$, and hence either of them can be chosen uniformly at random at that stage. This holds recursively for all permuations of $(i, \cdots, j), i \leq j$, given input distribution, and hence this problem is particular instance of dynamic programing, where at each stage, decision takes $O(n)$ time and there are in all, $O(n^2)$ quantities to be computed, hence its $O(n^2)$ algorithm. Once the sequence of selection of dividor is done, its just execution of those sequences, which needs $O(n^2)$ storage. The expected time complexity is determined by the minimum value : $V_{i^*}(1, \cdots, n|I)$.

Thus this was optimal quick sort given input distribution.

6.2 General Optimal Sorting

In general, given input distribution, we would like to come up with comparison based algorithm, without any extra constraints like quick sort, to do sorting optimally. We note that, using similar idea of dynamic programing, one can derive the sequence-tree of comparisons one should make, to achieve optimal expected comparisons given input distribution. This dynamic program runs for $O(n^3)$ times, but it gets the *optimal* sequence tree, using which we get the optimal expected running time, given the input distribution. The amount of storage it requires is high though, but since worry is time complexity, one can ignore that.

Before we end, we would like to note that, most of the algorithmic concepts can be found in [5] with classical analysis of quite a few of the above presented algorithms.

7 Conclusion and Future-work

We presented a new method based on information theoretic ideas, to estimate computational complexity of some problems. We demonstrated the power of the method by showing its application to various searching-sorting problems. The method is very simple and intuitive. In the process, we obtained some new algorithms for some class of searching-sorting problems. The method described naturally gives rise to technique to obtain lower bound. In particular, we showed the use of this method to obtain lower bound for $p-$shell sort algorithm. We strongly believe that, this method can be applied to more general class of problems to estimate the time complexity of algorithms.

Furthermore, if some entropy is left in output distribution, then it indicates that, any randomly chosen answer from this remaining output space will give some sort of approximation to exact answer. Hence, this method can be extended for studying the tradeoff between running time and approximation of solution. It is easy to see how this approximation method can be used for cases of searching/sorting. It will be interesting to apply it for some other cases.

Acknowledgement

We acknowledge Balaji Prabhakar and Neil O'Connell for their very useful discussion, guidance and suggestions.

References

[1] C. McDiarmid and R. Hayward. Strong Concentration for Quicksort. In Proceedings of *The Symposium of Discrete Algorithms*, 1992.

[2] R. Motwani and P. Raghavan. Randomized Algorithms. Chapter 1, pp 4-6, Cambridge University Press ,1995.

[3] P. Hennequin. Combinatorial analysis of quicksort algorithm. *Theoretical Informatics and Applications*, 23(3), pp 317-333 ,1989.

[4] D. Knuth. Sorting and Searching, Vol 3, The Art of Computer Programming. Addison-Wesley, 1973, revised, 1996.

[5] T. H. Cormen, C. E. Leiserson and R. L. Rivest. Introduction to Algorithms. MIT Press, Cambridge.

[6] A. Dembo and O. Zeitouni. Large Deviations Techinques and Applications. Springer-Verlag,1998.

[7] M. Charikar, R. Fagin, V. Guruswami, J. Kleinberg, P. Raghavan and A. Sahai. Query Strategies for Priced Information. In Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (2000).

[8] R. W. Yeung. Alphabetic Codes Revisited. IEEE Transactions on Information Theory, 1991.

[9] Tao Jiang, Ming Li and Paul Vitanyi. The average-case complexity of Shellsort. In Proceedings of ICALP, 1999.