Advanced Algorithms (CS 1.406) Spring 22.

Tentative topics

1. Introduction to randomized algorithms: RandQS, Min-Cut, Las vegas & Monte Carlo algorithms
2. Analysis of randomized algorithms: Yao's Min-max principle
3. Concentration inequalities: Coupon collector's problem, Balls and bins, Hashing, Set balancing
4. Probabilistic method, Lovasz Local Lemma, Method of Conditional probabilities
5. Markov chains and Random walks, 2-SAT, 3-SAT, $s \rightsquigarrow t$ connectivity
6. Polynomial Identity testing, Isolation lemma, Mulmuley-Vazirani-Vazirani perfect matching
7. Approximate counting
8. Derandomization techniques & Pseudorandom numbers
9. Number theoretic algorithms: Integer multiplication, Integer factorization, Primality, etc.
10. Algorithms for polynomials: Root finding, Polynomial multiplication, Polynomial division, GCD, Polynomial factorization, etc.

* Problem set 1 posted on teams
* Grading scheme and schedule of test are also posted on teams.

1. Introduction to randomized algorithms

Please revise the basics of probability!

Definition 1.1: *A* probability space *has three components:*

1. *a* sample space Ω, *which is the set of all possible outcomes of the random process modeled by the probability space;*
2. *a family of sets* $\mathcal{F}$ *representing the allowable* events, *where each set in* $\mathcal{F}$ *is a subset of the sample space* Ω; *and*
3. *a* probability function $\Pr : \mathcal{F} \rightarrow \mathbf{R}$ *satisfying Definition 1.2.*

Definition 1.2: *A* probability function *is any function* $\Pr: \mathcal{F} \to \mathbf{R}$ *that satisfies the following conditions:*

1. *for any event* E, $0 \leq \Pr(E) \leq 1$;
2. $\Pr(\Omega) = 1$; *and*
3. *for any finite or countably infinite sequence of pairwise mutually disjoint events* $E_1, E_2, E_3, \ldots,$

$$\Pr\left(\bigcup E_i\right) = \sum \Pr(E_i).$$

Question: Why study randomized algorithms?

Recall that P was the class of all problems solvable over Deterministic Turing machines in polynomial time.

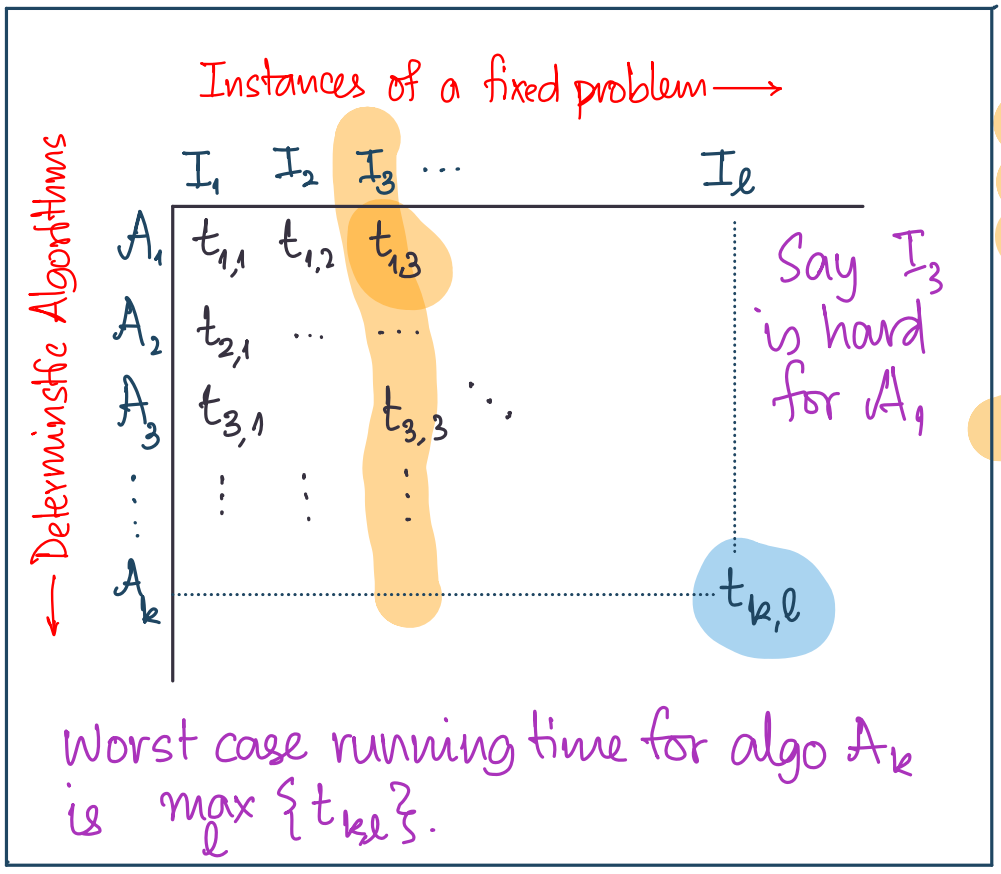
→ Unfolding this defn, we understand that a problem is in P if all its instances can be solved in polynomial time. That is, even for a "worst case" input, there is a poly time algorithm.

Recall that a problem is "not efficiently computable" if for every deterministic polynomial time algorithm A, there exists an instance of the problem I s.t A cannot solve I efficiently.

In other words, I is a worst case instance for algorithm A. Note that the "worst case" instances for different algorithms may be different.

Question: For a fixed instance I such that I is hard for A, will it still be hard for other algorithms? Or for what fraction of algorithms is the instance I hard for?

One way to answer the latter question is by setting up a distribution of algorithms and obtain the fraction as a prob.



Worst case running time for algo A_k is $\max_l \{t_{kl}\}$.

Grand wish:

Want the fraction of algorithms that are inefficient for any instance of a problem of interest, to be small.

What if we consider a probability distribution over inputs by fixing an algorithm?

Formally, for a problem B, we would like to

1. define a probability distribution over a set of deterministic algorithms for the problem B, and
2. show that the "expected" running time is "good".

Question: How does one define a probability distribution over deterministic algorithms?

Randomized algorithm: $A(x, r)$

input instance, random string

Algorithm A uses randomness by taking a random string r as input along with x, the instance of a problem.

Let us illustrate it via an example: Rand QS } Randomized Quick Sort

Recall that the "performance" of the deterministic Quick Sort depends on the selection of the pivot (see step 1 in algo below).

1.1 Randomized Quick Sort

Algorithm (QuickSort (S))
Input: a set of numbers S
Output: elements of S sorted in increasing order

1. Pick an element y from S.
2. By comparing each element of S with y, determine the set S_1 of elements smaller than y, and the set S_2 of elements larger than y.
3. Recursively sort S_1 and S_2. Then return the list that is obtained by concatenating QuickSort(S_1), y, QuickSort(S_2).

$S = \{1,3,5,7,4,12,2\} \rightarrow \{5,7,12\}$

From above, $T(S) = T(S_1) + T(S_2) + O(|S|)$. In worst case, $T(S)$ can be $O(|S|^2)$. But can we improve this perf by carefully choosing the pivot y?

Want: $T(S) \leq 2 \cdot T(\frac{S}{2}) + O(|S|)$
$\hookrightarrow O(|S| \log |S|)$

Algorithm (RandQS (S))

- Input: a set of numbers S. (given as a list)
- Output: elements of S sorted in an increasing order (output as a list). $S = \{1,3,5,7,4,12,2\}$

1. Pick an element y "uniformly at random" from S.
2. By comparing each element of S with y, determine the set S_1 of elements smaller than y, and the set S_2 the set of elements larger than y.

$7 \rightarrow 1,3,5,4,2$ and 12

3. Recursively sort S_1 and S_2. Then return the list concatenated as follows, in order.

$$RandQS(S_1) + [y] + RandQS(S_2).$$

Note that, RandQS has randomness implicitly. The input random string to the algorithm can be thought of as a concatenation of all random strings used in the above "procedure".

Qn: What is the "expected running time"?

Claim 1: The total running time is "roughly" equal to the no. of comparisons.

(There are other costs involved in picking the element y, and concatenating lists in step 3.)

Expected running time is "roughly" equal to the expected number of comparisons made. Most importantly, the expectation is over the random choices made in step 1.

Analysis of RandQS:

$\{1,3,5,7,4,12,2\} \leftarrow \{x_1,\dots,x_n\}$
$\{1,2,3,4,5,7,12\} \leftarrow \{s_1,\dots,s_n\}$

Let $S = \{x_1,\dots,x_n\}$ and let $s_1,\dots,s_n$ be the sorted list of elements of S in increasing order. For $i<j$, let X_{ij} be a 0-1 random variable such that

$$X_{ij} = \begin{cases} 1 & \text{if } s_i \text{ and } s_j \text{ are compared at some point over the run of the algorithm, and} \\ 0 & \text{otherwise} \end{cases}$$

Thus the total no. of comparisons (denoted by X) is $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$.

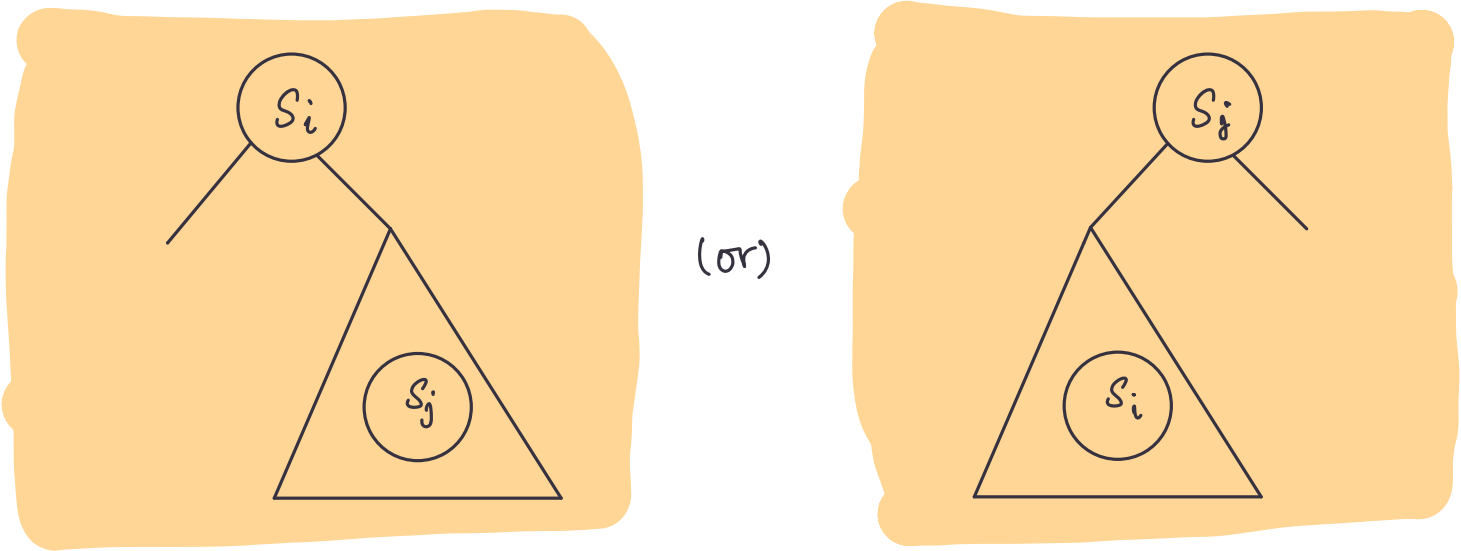
By taking expectation, we get that $\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right]$

$\mathbb{E}[X_{ij}] = 1 \cdot \Pr[X_{ij}=1] + 0 \cdot \Pr[X_{ij}=0]$

$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}]$.

Further, $E[X_{ij}]$ = probability that s_i and s_j are compared at some point during the run of the algorithm.

Observation: Elements s_i and s_j are compared if and only if either s_i or s_j is the first pivot selected by RandQS(S) from the set $\{s_i, \ldots, s_{j-1}, s_j\}$. $(i < j)$



One of these two situations will occur.

Recall that every element gets to be a pivot at some point in the course of the algorithm. The above observation states that either of s_i or s_j are picked as a pivot before $\{s_{i+1}, \ldots, s_{j-1}\}$. If s_i or s_j is not picked as a pivot first and say s_k (for some $k \in [i+1, j-1]$) is picked then s_i would go into the "left child" of s_k and s_j would go into the "right child" of s_k. In such a situation s_i and s_j would never be compared.

Putting all of this together, we get that the probability of comparing s_i and s_j is equal to choosing either s_i or s_j from the set $\{s_i, s_{i+1}, \ldots, s_{j-1}, s_j\}$ first as a pivot.

Thus, $\mathbb{E}[X_{ij}]$ = prob. of choosing s_i or s_j from $\{s_i, \ldots, s_{j-1}, s_j\}$

$$= \frac{2}{j-i+1}$$

and

$$\mathbb{E}[X] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}]$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k}$$

$$\leq 2 \cdot \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k}$$

$$= 2n\left(\sum_{k=1}^{n} \frac{1}{k}\right) = 2n H_n \approx 2n \ln n + \Theta(n).$$

(H_n: n^{th} Harmonic number)

Total no. of comparisons is at most $2nH_n$ in expectation over the random choices made during the run of the algorithm.

Thus, the expected running time of RandQS is $O(n \log n)$. ∎

Question: Suppose we modify QuickSort to always pick the first element as the pivot in step 1 but preprocess the input S by permuting its elements by uniformly chosen random permutation, before feeding it to QuickSort, what would be the expected running time?