Georgia Tech S'22 CS 6550/8803: Advanced Algorithms & Uncertainty

Lecture 9: Random-Order Model

Lecturer: *Sahil Singla* Last updated: February 5, 2022

For many online problems it's impossible to obtain any non-trivial performance guarantees for worst-case arrivals. Today, we will see how one could go "beyond the worst-case" and obtain much better performance guarantees assuming a random the arrival order.

1 Motivation: Picking a Large Element

Suppose we want to pick the maximum of a collection of n numbers. At the beginning, we know this cardinality n, but nothing about the range of numbers to arrive. We are then presented distinct non-negative real numbers $v_1, v_2, \ldots, v_n$ one by one; upon seeing a number v_i, we must either immediately pick it, or discard it forever. We can pick at most one number. The goal is to maximize the expected value of the number we pick, where the expectation is over any randomness in our algorithm. We want this expected value to be close to the maximum value $v_{\max} := \max_{i \in \{1,2,\ldots,n\}} v_i$. Formally, we want to minimize the *competitive ratio*, which is defined as the ratio of $v_{\max}$ to our expected value. Note that this maximum $v_{\max}$ is independent of the order in which the elements are presented, and is unknown to our algorithm until all the numbers have been revealed.

If we use a deterministic algorithm, our value can be arbitrarily smaller than $v_{\max}$, even for $n = 2$. Say the first number $v_1 = 1$. If our deterministic algorithm picks v_1, the adversary can present $v_2 = M \gg 1$; if it does not, the adversary can present $v_2 = {}^1\!/\!{}_M \ll 1$. Either way, the adversary can make the competitive ratio as bad as it wants by making M large.

Using a randomized algorithm helps only a little: a naïve randomized strategy is to select a uniformly random position $i \in \{1, \ldots, n\}$ up-front and pick the i^{th} number v_i. Since we pick each number with probability $1/n$, the expected value is $\sum_i v_i/n \geq v_{\max}/n$. This turns out to be the best we can do, as long the input sequence is controlled by an adversary and the maximum value is much larger than the others. Indeed, one strategy for the adversary is to choose a uniformly random index j, and present the request sequence $1, M, M^2, \ldots, M^j, 0, 0, \ldots, 0$—a rapidly ascending chain of j numbers followed by worthless numbers. If M is very large, any good algorithm must pick the last number in the ascending chain upon seeing it. But this is tantamount to guessing j, and random guessing is the best an algorithm can do. (This can be made formal using Yao's lemma from the last lecture.)

These bad examples show that the problem is hard for two reasons: the first reason being the large range of the numbers involved, and the second being the adversary's ability to carefully design these difficult sequences. Consider the following way to mitigate the latter effect: what if the adversary chooses the n numbers, but then the numbers are shuffled and presented to the algorithm in a uniformly random order? This random-order version of the problem above is commonly known as the *secretary problem*: the goal is to hire the best secretary (or at least a fairly good one) if the candidates for the job appear in a random order.

Somewhat surprisingly, randomly shuffling the numbers changes the complexity of the problem drastically. Here is the elegant 50%-algorithm:

1. Reject the first $n/2$ numbers, and then
2. Pick the first number after that which is bigger than all the previous numbers (if any).

Theorem 1. *The 50%-algorithm gets an expected value of at least* $v_{\max}/4$.

Proof. Assume for simplicity all numbers are distinct. The algorithm definitely picks $v_{\max}$ if the highest number is in the second half of the random order (which happens with probability $1/2$), and also the second-highest number is in the first half (which, conditioned on the first event, happens with probability at least $1/2$, the two events being positively correlated). Hence, we get an expected value of at least $v_{\max}/4$. (We get a stronger guarantee: we pick the highest number $v_{\max}$ itself with probability at least $1/4$, but we will not explore this expected-value-versus-probability direction any further.) $\square$

1.1 The Model and a Discussion

The secretary problem, with the lower bounds in the worst-case setting and an elegant algorithm for the random-order model, highlights the fact that sequential decision-making problems are often hard in the worst-case not merely because the underlying *set* of requests is hard, but also because these requests are carefully woven into a difficult-to-solve *sequence*. In many situations where there is no adversary, it may be reasonable to assume that the ordering of the requests is benign, which leads us to the random-order model. Indeed, one can view this as a *semi-random* model from Chapter 9, where the input is first chosen by an adversary and then randomly perturbed before being given to the algorithm.

Let us review the *competitive analysis* model for worst-case analysis of online algorithms (also discussed in Chapter 24). Here, the adversary chooses a sequence of requests and present them to the algorithm one by one. The algorithm must take actions to serve a request before seeing the next request, and it cannot change past decisions. The actions have rewards, say, and the *competitive ratio* is the optimal reward for the sequence (in hindsight) divided by the algorithm's reward. (For problems where we seek to minimize costs instead of maximize rewards, the competitive ratio is the algorithm's cost divided by the optimal cost.) Since the algorithm can never out-perform the optimal choice, the competitive ratio is always at least 1.

Now given any online problem, the *random-order model* (henceforth the *RO model*) considers the setting where the adversary first chooses a *set* S of requests (and not a sequence). The elements of this set are then presented to the algorithm in a uniformly random order. Formally, given a set $S = \{r_1, r_2, \ldots, r_n\}$ of $n = |S|$ requests, we imagine nature drawing a uniformly random permutation π of $\{1, \ldots, n\}$, and then defining the input sequence to be

$r_{\pi(1)}, r_{\pi(2)}, \cdots, r_{\pi(n)}$. As before, the online algorithm sees these requests one by one, and has to perform its irrevocable actions for $r_{\pi(i)}$ before seeing $r_{\pi(i+1)}$. The length n of the input sequence may also be revealed to the algorithm at the beginning, depending on the problem. The competitive ratio (for maximization problems) is defined as the ratio between the optimum value for S and the expected value of the algorithm, where the expectation is now taken over both the randomness of the reshuffle π and that of the algorithm. (Again, we use the convention that the competitive ratio is at least one, and hence have to flip the ratio for minimization problems.)

A strength of the RO model is its simplicity, and that it captures other commonly considered stochastic input models. Indeed, since the RO model does not assume the algorithm has any prior knowledge of the underlying set of requests (except perhaps the cardinality n), it captures situations where the input sequence consists of independent and identically distributed (i.i.d.) random draws from some fixed and unknown distribution. Reasoning about the RO model avoids over-fitting the algorithm to any particular properties of the distribution, and makes the algorithms more general and robust by design.

Another motivation for the RO model is aesthetic and pragmatic: the simplicity of the model makes it a good starting point for designing algorithms. If we want to develop an online algorithm (or even an offline one) for some algorithmic task, a good step is to first solve it in the RO model, and then extend the result to the worst-case setting. This can be useful either way: in the best case, we may succeed in getting an algorithm for the worst-case setting using the insights developed in the RO model. Else the extension may be difficult, but still we know a good algorithm under the (mild?) assumption of random-order arrivals.

Of course, the assumption of uniform random orderings may be unreasonable in some settings, especially if the algorithm performs poorly when the random-order assumption is violated. There have been attempts to refine the model to require less randomness from the input stream, while still getting better-than-worst-case performance. See [1] for further discussion.

2 The Secretary Problem

We saw the 50% algorithm based on the idea of using the first half of the random order sequence to compute a threshold that weeds out "low" values. This idea of choosing a good threshold will be a recurring one in this chapter. The choice of waiting for half of the sequence was for simplicity: a right choice is to wait for $1/e \approx 37\%$ fraction, which gives us the 37%-algorithm:

1. Reject the first n/e numbers, and then
2. Pick the first number after that (if any) which is bigger than all the previous numbers.

(Although n/e is not an integer, rounding it to the nearest integer does not impact the guarantees substantively.) Call a number a *prefix-maximum* if it is the largest among the numbers revealed before it. Notice being the maximum is a property of just the set of numbers, whereas being a prefix-maximum is a property of the random sequence and the current position. A *wait-and-pick* algorithm is one that rejects the first m numbers, and then picks the first prefix-maximum number.

Theorem 2. *As $n \to \infty$, the 37%-algorithm picks the highest number with probability at least $1/e$. Hence, it gets expected value at least $v_{\max}/e$. Moreover, n/e is the optimal choice of m among all wait-and-pick algorithms.*

Proof. If we pick the first prefix-maximum after rejecting the first m numbers, the probability we pick the maximum is

$$\sum_{t=m+1}^{n} \Pr[v_t \text{ is max}] \cdot \Pr[\text{max among first } t-1 \text{ numbers falls in first } m \text{ positions}]$$
$$\overset{(\star)}{=} \sum_{t=m+1}^{n} \frac{1}{n} \cdot \frac{m}{t-1} \quad = \quad \frac{m}{n}\big(H_{n-1} - H_{m-1}\big),$$

where $H_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k}$ is the k^{th} harmonic number. The equality $(\star)$ uses the uniform random order. Now using the approximation $H_k \approx \ln k + 0.57$ for large k, we get the probability of picking the maximum is about $\frac{m}{n} \ln \frac{n-1}{m-1}$ when m, n are large. This quantity has a maximum value of $1/e$ if we choose $m = n/e$. $\square$

Next we show we can replace any strategy (in a comparison-based model) with a wait-and-pick strategy without decreasing the probability of picking the maximum.

Theorem 3. *The strategy that maximizes the probability of picking the highest number can be assumed to be a wait-and-pick strategy.*

Proof. Think of yourself as a player trying to maximize the probability of picking the maximum number. Clearly, you should reject the next number v_i if it is not prefix-maximum. Otherwise, you should pick v_i only if it is prefix-maximum and the probability of v_i being the maximum is more than the probability of you picking the maximum in the remaining sequence. Let us calculate these probabilities.

We use Pmax to abbreviate "prefix-maximum". For position $i \in \{1, \ldots, n\}$, define

$$f(i) = \Pr[v_i \text{ is max} \mid v_i \text{ is Pmax}] \overset{(\star)}{=} \frac{\Pr[v_i \text{ is max}]}{\Pr[v_i \text{ is Pmax}]} \overset{(\star\star)}{=} \frac{1/n}{1/i} = \frac{i}{n},$$

where equality $(\star)$ uses that the maximum is also a prefix-maximum, and $(\star\star)$ uses the uniform random ordering. Note that $f(i)$ increases with i.

Now consider a problem where the numbers are again being revealed in a random order but we must reject the first i numbers. The goal is to still maximize the probability of picking the highest of the n numbers. Let $g(i)$ denote the probability that the optimal strategy for this problem picks the global maximum.

The function $g(i)$ must be a non-increasing function of i, else we could just ignore the $(i+1)^{st}$ number and set $g(i)$ to mimic the strategy for $g(i+1)$. Moreover, $f(i)$ is increasing. So from the discussion above, you should not pick a prefix-maximum number at any position i where $f(i) < g(i)$ since you can do better on the suffix. Moreover, when $f(i) \geq g(i)$, you should pick v_i if it is prefix-maximum, since it is worse to wait. Therefore, the approach of waiting until f becomes greater than g and thereafter picking the first prefix-maximum is an optimal strategy. $\square$

Theorems 2 and 3 imply for $n \to \infty$ that no algorithm can pick the maximum with probability more than $1/e$. Since we placed no bounds on the number magnitudes, this can also be used to show that for any $\epsilon > 0$, there exist an n and numbers $\{v_i\}_{i\in\{1,\dots,n\}}$ where every algorithm has expected value at most $(1/e + \epsilon) \cdot \max_i v_i$.

3 Maximum-Weight Forest

Suppose the items arriving in a random order are the n edges of a (multi-)graph $G = (V, E)$, with edge e having a value/weight v_e. The algorithm knows the graph at the beginning, but not the weights. When the edge e arrives, its weight v_e is revealed, and we decide whether to pick the edge or not. Our goal is to pick a subset of edges with large total weight that form a forest (i.e., do not contain a cycle). The target $V^\star$ is the total weight of a maximum-weight forest of the graph: offline, we can solve this problem using, e.g., Kruskal's greedy algorithm. This *graphical secretary* problem generalizes the secretary problem: Imagine a graph with two vertices and n parallel edges between them. Since any two edges form a cycle, we can pick at most one edge, which models the single-item problem.

As a first step towards an algorithm, suppose all the edge values are either 0 or v (but we don't know in advance which edges have what value). A greedy algorithm is to pick the next weight-v edge whenever possible, i.e., when it does not create cycles with previously picked edges. This returns a max-weight forest, because the optimal solution is a maximal forest among the subset of weight-v edges, and every maximal forest in a graph has the same number of edges. This suggests the following algorithm for general values: if we know some value v for which there is a subset of acyclic edges, each of value v, with total weight $\geq \frac{1}{\alpha} \cdot V^\star$, then we can get an α-competitive solution by greedily picking value-v edges whenever possible.

How do we find such a value v that gives a good approximation? The Random-Threshold algorithm below uses two techniques: *bucketing* the values and *(randomly) mixing* a collection of algorithms. We assume that all values are powers of 2; indeed, rounding values down to the closest power of 2 loses at most a factor of 2 in the final guarantee.

1. Ignore the first $n/2$ items and let $\hat{v}$ be their highest value.
2. Select a uniformly random $r \in \{0, \dots, \log n\}$, and set threshold $\tau := \hat{v}/2^r$.
3. For the second $n/2$ items, greedily pick any item of value at least τ that does not create a cycle.

Theorem 4. *The order-oblivious Random-Threshold algorithm for the graphical secretary problem gets an expected value* $\Omega\big(\frac{V^\star}{\log n}\big)$.

Proof. What does the optimum solution $S^\star$ (with value $V^\star$) look like? Let $v_{\max}$ be the maximum value of any edge/item in $S^\star$. If $v_{\max} \geq V^\star/4$, we could run the single-item order-oblivious algorithm to get an expected value of $v_{\max}/4 = \Omega(V^\star)$.

Else, define "bucket" B_i to be edges of value exactly $v_{\max}/2^i$. We claim that the items in $S^\star$ that lie within buckets $B_0, B_1, \ldots, B_{\log_2 n}$ have total value at least $V^\star/2$. Indeed, if not, at least $V^\star/2$ of $S^\star$'s value must come from items of value less than $v_{\max}/2^{\log_2 n} = v_{\max}/n$. But there are at most $n-1$ edges in the forest $S^\star$. Even if each of those edges contributes $v_{\max}/n < V^\star/(4n)$, we cannot get $V^\star/2$ value from them, a contradiction. Hence, we get that

$$\textstyle\sum_{i=0}^{\log n}(v_{\max}/2^i)\cdot|S^\star \cap B_i| \quad \geq \quad V^\star/2. \tag{1}$$

Moreover, for any of these buckets i, we could run the algorithm that picks items of value exactly $v_{\max}/2^i$, and get a value of $v_{\max}/2^i \cdot |S^\star \cap B_i|$.

The difficulty is that we know neither $v_{\max}$, nor whether $v_{\max} \geq V^\star/4$, nor which of these buckets to choose. Hence the idea is to "mix" all the above algorithms by randomly choosing between them.

It is easy to see that if $v_{\max} \geq V^\star/4$, the single-item algorithm does well. Else, condition on $v_{\max}$ is in the first half (which happens with probability $1/2$), so $\hat{v} = v_{\max}$. Moreover, for any choice of $r = i \in \{0, \ldots, \log n\}$ (which happens with probability $\Theta(1/\log n)$), the second half contains in expectation half the elements of $S^\star \cap B_i$; now using (1), our algorithm gets value $\Omega(V^\star/\log n)$ in expectation. □

3.1 An Improved Algorithm for Max-Weight Forests

The Random-Threshold algorithm above used relatively few properties of the max-weight forest. Indeed, it extends to downward-closed set systems with the property that if all values are 0 or v then picking the next value-v element whenever possible gives a near-optimal solution. However, we can do better using properties of the underlying graph. Here is a constant-competitive algorithm for graphical secretary where the main idea is to decompose the problem into several disjoint single-item secretary problems.

1. Choose a uniformly random permutation $\widehat{\pi}$ of the vertices of the graph.
2. For each edge $\{u, v\}$, direct it from u to v if $\widehat{\pi}(u) < \widehat{\pi}(v)$.
3. Independently for each vertex u, consider the edges directed *towards* u and run the order-oblivious 50%-algorithm on these edges.

Theorem 5. *The algorithm above for the graphical secretary problem is order-oblivious and gets an expected value at least* $V^\star/8$.

Proof. The algorithm picks a forest, i.e., there are no cycles (in the undirected sense) among the picked edges. Indeed, the highest numbered vertex (w.r.t. $\widehat{\pi}$) on any such cycle would have two or more incoming edge picked, which is not possible.

However, since we restrict to picking only one incoming edge per vertex, the optimal max-weight forest $S^\star$ may no longer be feasible. Despite this, we claim there is a forest with the one-incoming-edge-per-vertex restriction, and expected value $V^\star/2$. (The randomness here is over the choice of the permutation $\widehat{\pi}$, but not of the random order.) Since the 50%-algorithm gets a quarter of this value (in expectation over the random ordering), we get the desired bound of $V^\star/8$.

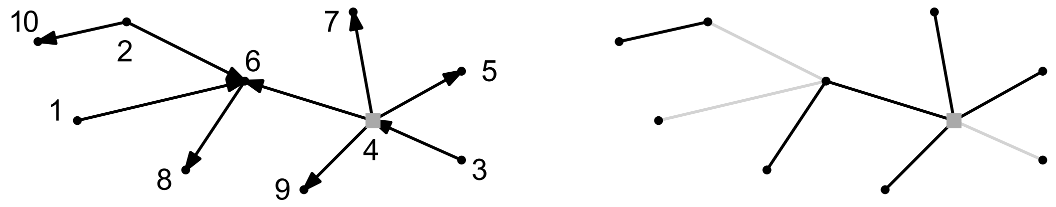

Figure 1: The optimal tree: the numbers on the left are those given by $\widehat{\pi}$. The grey box numbered 4 is the root. The edges in the right are those retained in the claimed solution.

To prove the claim, root each component of $S^\star$ at an arbitrary node, and associate each non-root vertex u with the unique edge $e(u)$ of the undirected graph on the path towards the root. The proposed solution chooses for each vertex u, the edge $e(u) = \{u, v\}$ if $\widehat{\pi}(v) < \widehat{\pi}(u)$, i.e., if it is directed *into* u (Figure 1). Since this event happens with probability $1/2$, the proof follows by linearity of expectation. ☐

This algorithm is order-oblivious because the 50%-algorithm has the property. If we don't care about order-obliviousness, we can instead use the 37%-algorithm and get expected value at least $^{V^\star}/_{2e}$.

3.2 The Matroid Secretary Problem

One of the most tantalizing generalizations of the secretary problem is to *matroids*. (A matroid defines a notion of *independence* for subsets of elements, generalizing linear-independence of a collection of vectors in a vector space. E.g., if we define a subset of edges to be independent if they are acyclic, these form a "graphic" matroid.) Suppose the n items form the ground set elements of a known matroid, and we can pick only subsets of items that are independent in this matroid. The weight/value $V^\star$ of the max-weight independent set can be computed offline by the obvious generalization of Kruskal's greedy algorithm. The open question is to get an expected value of $\Omega(V^\star)$ online in the RO model. The approach from Theorem 4 gives expected value $\Omega(V^\star/\log k)$, where k is the largest size of an independent set (its *rank*). The current best algorithms (which are also order-oblivious) achieve an expected value of $\Omega(V^\star/\log\log k)$. Moreover, we can obtain $\Omega(V^\star)$ for many special classes of matroids by exploiting their special properties, like we did for the graphic matroid above; see the notes for references.

4 Facility Location

A slightly different algorithmic intuition is used for the *online facility location* problem, which is related to the k-means and k-median clustering problems. In this problem, we are given a metric space (V, d) with point set V, and distances $d : V \times V \to \mathbb{R}_{\geq 0}$ satisfying the triangle inequality. Let $f \geq 0$ be the cost of opening a facility; the algorithm can be extended to cases where different locations have different facility costs. Each request is specified by a point in the metric space, and let $R_t = \{r_1, \ldots, r_t\}$ be the (multi)-set of request points that arrive by time t. A solution at time t is a set $F_t \subseteq V$ of "facilities" whose opening cost is $f \cdot |F_t|$, and whose connection cost is the sum of distances from every request to its closest facility in F_t, i.e., $\sum_{j \in R_t} \min_{i \in F_t} d(j, i)$. An open facility remains open forever, so we require $F_{t-1} \subseteq F_t$. We want the algorithm's total cost (i.e., the opening plus connection costs) at time t to be at most a constant times the optimal total cost for R_t in the RO model. Such a result is impossible in the adversarial arrival model, where a tight $\Theta(\frac{\log n}{\log \log n})$ worst-case competitiveness is known.

There is a tension between the two components of the cost: opening more facilities increases the opening cost, but reduces the connection cost. Also, when request r_t arrives, if its distance to its closest facility in F_{t-1} is more than f, it is definitely better (in a greedy sense) to open a new facility at r_t and pay the opening cost of f, than to pay the connection cost more than f. This suggests the following algorithm:

```
When a request $r_t$ arrives:
  1. Let $d_t := \min_{i \in F_{t-1}} d(r_t, i)$ be its distance to the closest facility in $F_{t-1}$.
  2. Set $F_t \leftarrow F_{t-1} \cup \{r_t\}$ with probability $p_t := \min\{1, d_t/f\}$, and $F_t \leftarrow F_{t-1}$
     otherwise.
```

Observe that the choice of p_t approximately balances the expected opening cost $p_t \cdot f \leq d_t$ with the expected connection cost $(1 - p_t)d_t \leq d_t$. Moreover, since the set of facilities increases over time, a request may be reassigned to a closer facility later in the algorithm; however, the analysis works even assuming the request r_t is permanently assigned to its closest facility in F_t.

Theorem 6. *The above algorithm is $O(1)$-competitive in the RO model.*

The insight behind the proof is a charging argument that first classifies each request as "easy" (if they are close to a facility in the optimal solution, and hence cheap) or "difficult" (if they are far from their facility). There are an equal number of each type, and the random permutation ensures that easy and difficult requests are roughly interleaved. This way, each difficult request can be paired with its preceding easy one, and this pairing can be used to bound their cost.

Proof. (Sketch) For simplicity assume that $f = 1$, and all distances are at most 1, so that $\min\{1, d_t/f\} = d_t$. Consider some facility i^* in the optimal solution, and let S be the set of request indices served by it, i.e., those for which the closest facility is i^*. Let

$d^* := \frac{1}{|S|}\sum_{j\in S} d(r_j, i^*)$ be the distance from an average request point in S to the facility i^*. The optimal cost to open facility i^* and serve these requests is

$$OPT_S := f + \textstyle\sum_{j\in S} d(r_j, i^*) = 1 + |S|d^*.$$

Observe that in our algorithm, each request point r_t pays $f = 1$ with probability $\min(1, d_t/f) = d_t$, and d_t otherwise. Summing these two, the expected cost for request r_t is at most $2d_t$, and hence it suffices to show that the algorithm's cost for requests in S, which is $\sum_{j\in S} d_j$, is at most $O(1)\cdot OPT_S$.

Sort all requests in S in order of their distance from the facility i^*, and call the first half the "close" requests C, and the second half the "distant" requests D. The close requests are at distance at most $2d^*$ from the facility by Markov's inequality. Note that once we open a facility j in C, each of the subsequent requests j' pays at most $d_{j'} \le d(j', j) \le d(j', i^*) + d(j, i^*) \le d(j', i^*) + 2d^*$, which sum to at most $OPT_S + 2d^*|S| \le 3OPT_S$. Hence, we need to bound the cost incurred by requests that arrive before any facility is opened inside S.

We first "charge" the distant requests to the close requests. Indeed, consider the (random) arrival ordering: each distant request $j \in D$ charges to its preceding close request $j' \in C$ in this ordering (if any). Since j can use the same facility as j', the distance $d_j \le d_{j'} + d(r_j, r_{j'}) \le d_{j'} + d(r_j, i^*) + d(r_{j'}, i^*)$. Now by the random ordering and the fact that $|C| = |D|$, each close request $j' \in C$ only has a single distant request $j \in D$ charging to it in expectation, so summing gives that $\mathbb{E}[\sum_{j\in D} d_j] \le \mathbb{E}[\sum_{j\in C} d_j] + O(OPT_S)$. (The distant requests that come before all close requests have to be charged separately as below; we skip the details.)

Finally, it remains to bound the expected cost for the close requests before we open any close facilities: for each request j, we pay d_j and the process stops (since we open a facility) with probability d_j, else we continue. A simple calculation shows that the total expected cost is at most 1. $\square$

To summarize: we used the random ordering to create a matching between the distant ("difficult") and the close ("easy") requests, and charged the former to the latter. The latter we bounded by the triangle inequality, and the fact that we open facilities with probability proportional to how much we pay at each step.

Notes

The lecture is partly based on [1].

References

[1] Gupta, Anupam, and Sahil Singla. "Random-Order Models." (2020), Chapter 11 in Beyond the Worst-Case Analysis of Algorithms.