On-line algorithms and Machine Learning

Najmieh Sadat Safarabadi

Spring 2023

On-line Algorithms

Many of the algorithmic problems that arise in practice are online. In these problems the input is only partially available because some relevant input data arrives in the future and is not accessible at present.

An online algorithm must generate output without knowledge of the entire input. Online problems arise in many areas of computer science .

Data structures

Consider a data structure such as a linear linked list or a tree. We wish to dynamically maintain this structure so that a sequence of accesses to elements can be served at low cost.

Future access patterns are unknown The quality of online algorithms is usually evaluated using competitive analysis. The idea of competitiveness is to compare the output generated by an online algorithm to the output produced by an optimal offline algorithm.

The better an online algorithm approximates the optimal solution, the more competitive this algorithm is.

Self-organizing data structures

For Example The list update problem is one of the first online problems that were studied with respect to competitiveness. The problem is to maintain a dictionary as an unsorted linear list.

Consider a set of items that is represented as a linear linked list. We receive a request sequence σ, where each request is one of the following operations.

(1) It can be an access to an item in the list,

(2) it can be an insertion of a new item into the list, or

(3) it can be a deletion of an item. To access an item, a list update algorithm starts at the front of the list and searches linearly through the items until the desired item is found.

Self-organizing data structures

In serving requests a list update algorithm incurs cost.

If a request is an access or a delete operation, then the incurred cost is i, where i is the position of the requested item in the list If the request is an insertion, then the cost is n + 1, where n is the number of items in the list before the insertion.

While processing a request sequence, a list update algorithm may rearrange the list.

Immediately after an access or insertion, the requested item may be moved at no extra cost to any position closer to the front of the list. These exchanges are called free exchanges.

Self-organizing data structures

Using free exchanges, the algorithm can lower the cost on subsequent requests. At any time two adjacent items in the list may be exchanged at a cost of 1.

These exchanges are called paid exchanges.

The goal is to serve the request sequence so that the total cost is as small as possible With respect to the list update problem, we require that a c-competitive online algorithm has a performance ratio of c for all size lists.

More precisely, a deterministic online algorithm for list update is called competitive if there is a constant a such that for all size lists and all request sequences.

Here should re order

Certainly, there are other data structures such as balanced search trees or hash tables that, depending on the given application, can maintain a set in a more efficient way. In general, linear lists are useful when the set is small and consists of only a few dozen items.

Recently, list update techniques have been applied very successfully in the development of data compression algorithms, There are three well-known deterministic online algorithms for the list update problem

1. **Move-To-Front**: Move the requested item to the fron of the list

2. **Transpose**: Exchange the requested item with the immediately preceding item in the list

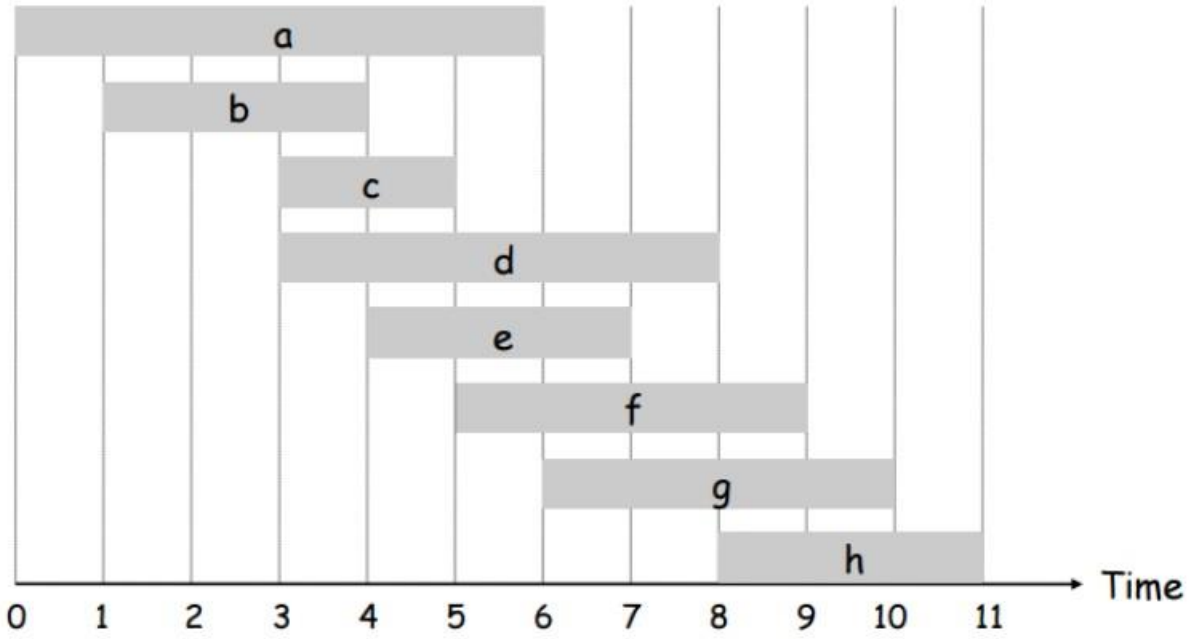
3. **Frequeucy Count**: Maintain a frequency count for each item in the list. Wherever an item is requested, increase it's count by 1. Maintain the list so that the items always occur in nonincreasing order of frequency count

Interval coloring is also known as the Interval Scheduling problem.

<u>The goal is to:</u>

Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



The interesting thing to realize about the interval scheduling problem is that it is only asking "**which non-overlapping intervals should we choose so that we can keep the highest number of intervals**".

The question has nothing to do with re-scheduling the appointments that these intervals might represent. They all start and end at fixed times and we accept that we can't keep all of them. The goal is only to choose the ones that let us keep the most possible.

Greedy algorithms are algorithms that, at every point in their execution, have some straightforward method of choosing the best thing to do next and just repeatedly apply that method to the remaining things to do until they are done. To be honest, I think Wikipedia says it better:

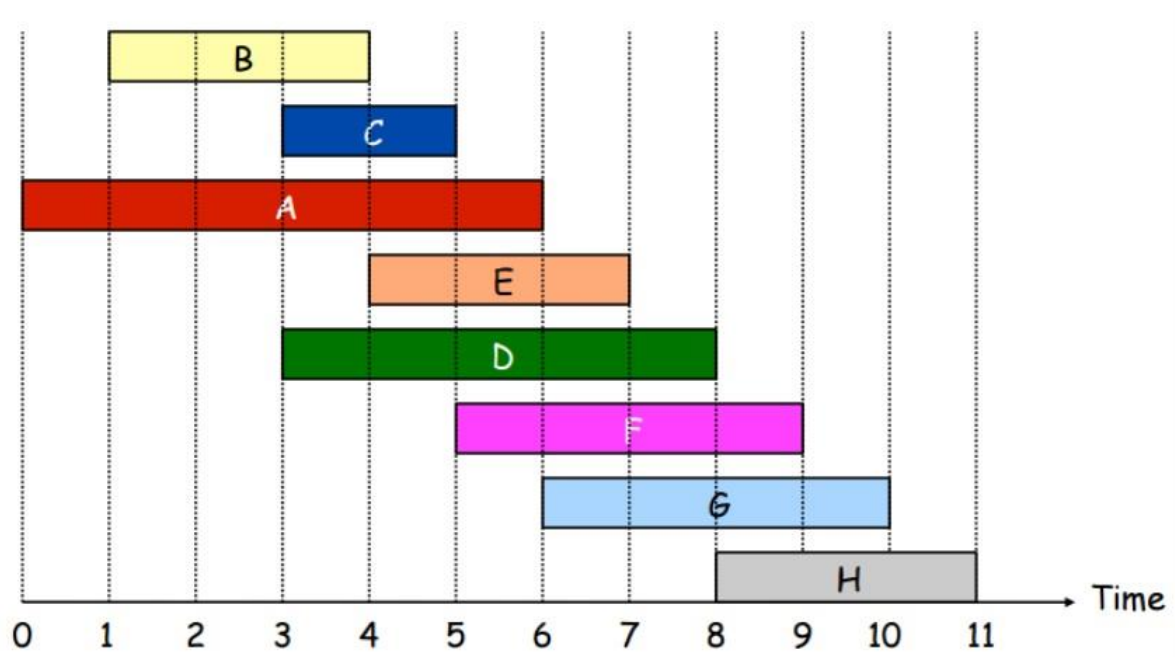
A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

You should know that there are many cases where greedy algorithms are, in principle alone, not capable of finding the global optimum. Not for the problem we're here to talk about though!

As it turns out, there exists a greedy algorithm to solve the interval scheduling problem that can be proven to always find the optimal solution.

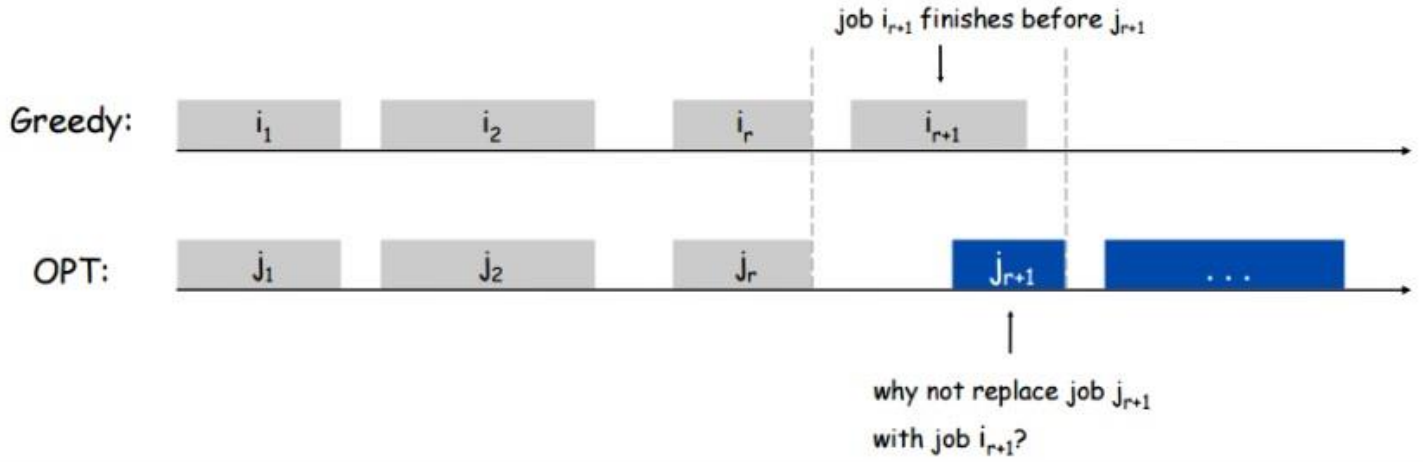
The optimal algorithm is 2 steps:

1. Sort the list of intervals (s_i, f_i) by finishing time f_i.
2. While elements remain in the list, add the soonest-ending interval compatible with the exisiting solution set to the solution set.



It's not obvious that this algorithm produces the optimal solution. It does, though. How do we prove this? By contradiction.

- Let $i_1, i_2, \ldots i_k$ denote a set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote a set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



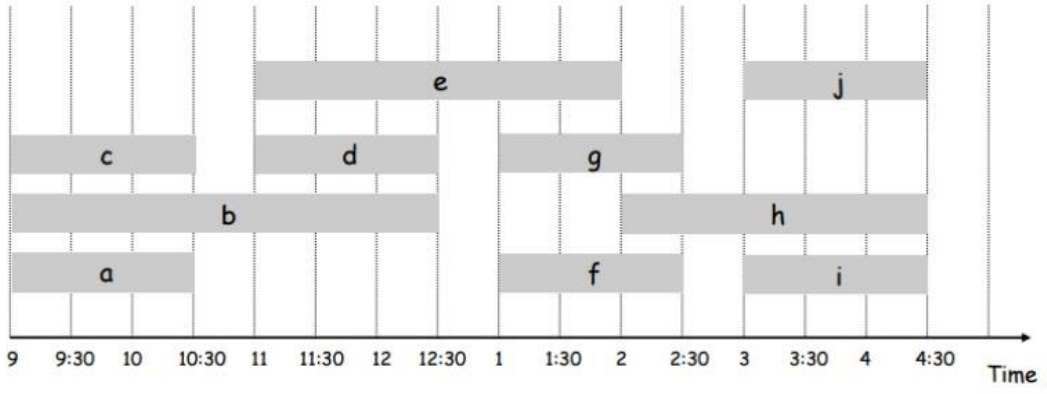
As we can see in the picture, there are two reasons why the greedy is the optimal. First, even though we chose the maximal value of r, there really is no reason to choose the optimal's $(r+1)^{th}$ choice over the greedy's $(r+1)^{th}$ choice. The greedy's choice must end soonest, which means that it makes more room for all future choices than any other choice does. So, between the greedy's $(r+1)^{th}$ choice and the optimal's $(r+1)^{th}$ choice, the greedy's $(r+1)^{th}$ choice is actually **more optimal**. So if this optimal was truly the optimal it would make the same $(r+1)^{th}$ choice as the greedy. Second, since the greedy's $(r+1)^{th}$ choice ends the soonest, both the greedy and the optimal have the same choices available for their $(r+2)^{th}$ choice, and once again the greedy's $(r+2)^{th}$ choice would be the most optimal. Thus we have shown that the greedy algorithm consistently makes the best decision and that even though we picked r to be as large as possible, it clearly could have been larger and that's a contradiction so our proof by contradiction is complete.

Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

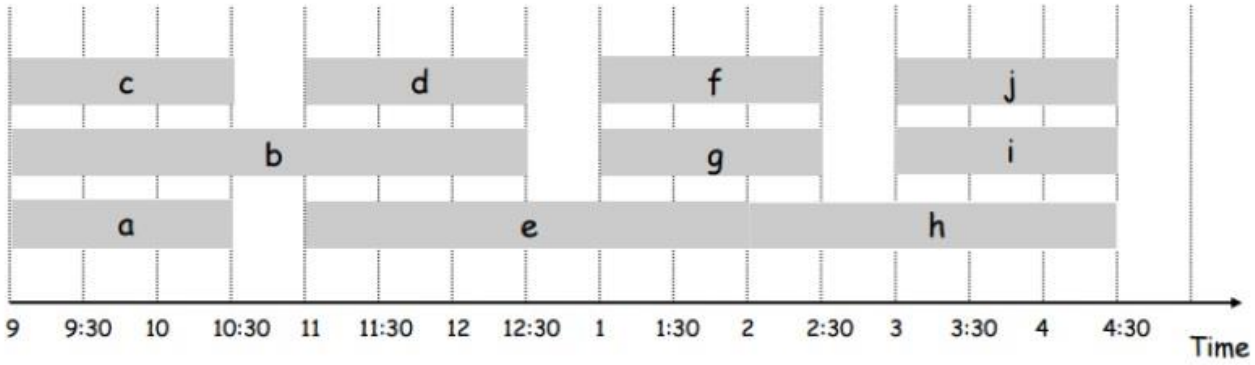
Ex: This schedule uses 4 classrooms to schedule 10 lectures.



Notice that in the image above, there are at most 3 lectures that are running at every moment in time. This is called the **depth** of the set of intervals.

Notice also that in the picture above, 4 classrooms are used, despite the depth of the set of intervals being 3. Below, a solution is shown that uses only 3 classrooms for the same set of intervals.

Ex: This schedule uses only 3.



This is a better way to partition the intervals since it uses only 3 classrooms. Notice that the number of classrooms used is equal to the depth of the set of intervals. A good question is whether or not this is always possible? It's obvious that the number of classrooms needed is at least the depth of the set, but does there always exists a partitioning that uses exactly the number of classrooms as the depth of the set **and no more**?

Not only is the answer yes, but – you guessed it – there's a greedy algorithm to solve this problem too!

Some details about the algorithm above:

- A good way to store the classrooms is in a priority queue. Priority is given to classrooms that have been free the longest – in other words, classrooms whose last scheduled interval had the earliest ending time.
- This algorithm runs in $O(n\ log(n))$ time. First, it's clearly $O(n\ log(n))$ for the sorting of the intervals. Then, for each of the n intervals, we must choose a classroom to add them to. This choice takes at worst $O(log(n))$ (assuming we implement the priority queue as a heap), since each time we add to a classroom, we need to remove it and re-instert it into the priority queue since its key for the priority queue (the ending time of its most recent interval) has been changed.

So it runs fast, but how do we know that it gives the optimal solution? With proof, of course. For the following proof, keep in mind that our algorithm never schedules two overlapping intervals in the same classroom.

Before we start the proof, just know that the way the proof works is to show that the algorithm would use D classes to partition the intervals only if at some point D intervals overlapped with each other. This is clearly equivalent to proving that D is the **depth** of the set of intervals.

Okay so let's say that the greedy partitioning algorithm terminates with D classrooms used. The D^{th} classroom was only used because it was trying to add some interval I that was incompatible with the other $D - 1$ classrooms. The algorithm decides I is incompatible on the basis that the other $D - 1$ classrooms has intervals ending after I's start time. And since the algorithm adds intervals in ascending order of start times, the other classrooms' last intervals has to start before I starts. So we have shown that the other classrooms' last intervals started before I and ended after I started – therefore they all overlap with I. Thus we have shown that if the algorithm uses D classrooms, it is only because there are D overlapping intervals. Proof is done.

Interval Coloring (An example)

1 - In the interval coloring problem, in which each request is an interval on the real line and each action assigns a color to the current request.

2 - No two overlapping intervals may receives the same color.

3 - The cost of a sequence of requests is the number of colors used. This problem can be interpreted a scheduling problem in which each interval represents the time span of some task, and the color represents the processor assigned to execute the task.

4 – give an online algorithm with optimal competitive ratio for this problem.

Interval Coloring (An example)

As each interval I arrives it is assigned a positive integer h(I) called it's height and a color as follows:

1) If I does not intersect any previous interval of height 1 then h(I) is set equal to 1; otherwise, h(I) is set equal to the least $j > 1$ such that I does not intersect more than two previous intervals of height j

2) One color is reserved for intervals of height 1, and, for each $j > 1$, three colors are reserved for each height j; interval I is assigned any color, among those reserved for it's height, that has not been assigned to any previous interval that intersects I

If K is the maximum height assigned to any interval then:

1) We described the Kiersteead-Trotter algorithm uses at most 3k – 2 colors;

2) There is a set of k mutually intersecting intervals; thus, any algorithm requires at least k colors.

It follows that, it the optimal number of colors is k, then the Kierstead-trotter on-line algorithm will requires at most 3k – 2 colors.

It can be shown by means of an adversary argument that, for any online algorithm, any K, there exists an instance for which the optimal number of colors required is k but algorithm requires 3k – 2 colors. Thus, the Kierstead-Trotter algorithm is an optimal on-line algorithm.

Interval Coloring (An example)

Problem Given n lectures, each with a start time and a finish time, find a minimum number of lecture halls to schedule all lectures so that no two occur at the same time in the same hall.

Example Given lectures, A(1, 3], B(1, 5], C(1, 2], D(4, 6], E(4, 8], F(7, 10], G(7, 11], H(9, 13], I(12, 14], J(12, 15], an optimal schedule uses 3 halls.

It schedules lectures C, D, F, J in the first hall, B, G, I in the second hall, and A, E, H in the third hall. We will number the halls by positive integers 1, 2, 3, ...

High-level Algorithm

A greedy method for solving this problem works as follows.

for each lecture ℓ in order of increasing start time **do**

assign to ℓ the smallest hall that has not been assigned to any previously assigned lectures that overlap ℓ

Low-level Algorithm

$d \leftarrow 0$ // d contains the largest hall ever used

$A \leftarrow \emptyset$ // A is an empty queue of available halls that are ever used

/* Priority queue Q contains all the endpoints (with references to their intervals). */

/* We break ties in favor of finish time; for equal finish times break the tie by start time.

$Q \leftarrow \{ S[i], F[i] : 1 \leq i \leq n \}$

while $Q \neq \emptyset$ **do** {

    $x \leftarrow$ extract-min(Q)

    if x is a start time **then** {

        if $A \neq \emptyset$ **then** // there is some hall that was used but is available right now

Low-level Algorithm

```
            c ← dequeue(A) // so reuse it
        else { // all halls that's ever used are being used
            d ← d + 1 // so need to get a new hall
            c ← d
        }
        assign hall c to the interval of x
    } else { // x is a finish time
        c ← hall of the interval of x
        enqueue(c, A) // release hall c and put it in the pool
    }
}
```

Running Time

The algorithm has an O(n log n) time bound because there are 2n endpoints, and we do an insert and an extract-min on each endpoint, with the work of priority queue operations dominating all other work.

Correctness

Let k be the maximum number of halls ever used at any point in time in our algorithm, i.e., k is the value of the variable d at the end of the algorithm. Consider the point in our algorithm when hall number k is assigned to a lecture.

At that time, every hall from hall 1 to hall k − 1 is assigned to some lecture. Our algorithm only lets a lecture occupy a hall from its start to finish time, but no more.

This means that the set of lectures in the input instance contains k mutually overlapping lectures. This means that k is a lower bound on the number of halls required by any algorithm. Since our algorithm uses exactly k halls, it uses the least number possible

Adversary model in the on-line Algorithms

In computer science, an online algorithm measures its competitiveness against different adversary models.

For deterministic algorithms, the adversary is the same as the adaptive offline adversary. For randomized online algorithms competitiveness can depend upon the adversary model used.

The three common adversaries are the oblivious adversary, the adaptive online adversary, and the adaptive offline adversary.

The **oblivious adversary** is sometimes referred to as the weak adversary. This adversary knows the algorithm's code, but does not get to know the randomized results of the algorithm.

The **adaptive online adversary** is sometimes called the medium adversary. This adversary must make its own decision before it is allowed to know the decision of the algorithm.

The **adaptive offline adversary** is sometimes called the strong adversary. This adversary knows everything, even the random number generator. This adversary is so strong that randomization does not help against it.

The formulations of list update algorithms generally assume that a request sequence consists of accesses only. It is obvious how to extend the algorithms so that they can also handle insertions and deletions.

The Move-To-Front algorithm is 2-competitive. The algorithms Transpose and Frequency-Count are not c-competitive, for any constant c.

For the Move-To-Front algorithm: a lower bound on the competitiveness that can be achieved by deterministic online algorithms. This lower bound implies that Move To-Front has an optimal competitive ratio

The Move-To-Front algorithm is 2-competitive. The algorithms Transpose and Frequency-Count are not c-competitive, for any constant c.

For the Move-To-Front algorithm: a lower bound on the competitiveness that can be achieved by deterministic online algorithms. This lower bound implies that Move To-Front has an optimal competitive ratio

Against adaptive adversaries, no randomized online algorithm for list update can be better than 2-competitive Bit: Each item in the list maintains a bit that is complemented whenever the item is accessed.

If an access causes a bit to change to 1, then the requested item is moved to the front of the list. Otherwise the list remains unchanged. The bits of the items are initialized independently and uniformly at random

For example a generalized the Bit algorithm has been suggested and proved an upper bound of $\sqrt{3} \approx 1.73$ against oblivious adversaries.
The best randomized algorithm currently known is a combination of the Bit algorithm and a deterministic 2-competitive online algorithm called Timestamp proposed

Timestamp (TS): Insert the requested item, say x, in front of the first item in the list that precedes x and that has been requested at most once since the last request to x.
If there is no such item or if x has not been requested so far, then leave the position of x unchanged.
As an example, consider a list of six items being in the order

$$L : x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4 \rightarrow x_5 \rightarrow x_6$$

Suppose that algorithm TS (Timestamp) has to serve the second request to x5 in the request sequence $\sigma = \ldots x_5, x_2, x_2, x_3, x_1, x_1, x_5$. Items x_3 and x_4 were requested at most once since the last request to x_5 whereas x_1 and x_2 were both requested twice.

Thus, TS will insert x_5 immediately in front of x_3 in the list. Therefore a combination of Bit and TS was propsed.

The classic on-line problem first analysed with competitive analysis is the list update problem: Given a list of items and a sequence of requests for the various items, minimize the cost of accessing the list where the elements closer to the front of the list cost less to access. (Typically, the cost of accessing an item is equal to its position in the list.) After an access, the list may be rearranged. Most rearrangements have a cost. The MoveTo-Front algorithm simply moves the requested item to the front after the access, at no cost

The Transpose algorithm swaps the accessed item with the item immediately before it, also at no cost. Classical methods of analysis showed that Transpose is optimal in certain contexts. In practice, Move-To-Front performed much better.

Competitive analysis was used to show that an adversary can make Transpose perform arbitrarily badly compared to an optimal algorithm, whereas Move-ToFront can never be made to incur more than twice the cost of an optimal algorithm

In the case of online requests from a server, competitive algorithms are used to overcome uncertainties about the future.

That is, the algorithm does not"know" the future, while the imaginary adversary (the "competitor")"knows".

Similarly, competitive algorithms were developed for distributed systems, where the algorithm has to react to a request arriving at one location, without"knowing" what has just happened in a remote location

Competitive analysis is a method invented for analyzing online algorithms, in which the performance of an online algorithm (which must satisfy an unpredictable sequence of requests, completing each request without being able to see the future) is compared to the performance of an optimal offline algorithm that can view the sequence of requests in advance.

An algorithm is competitive if its competitive ratio—the ratio between its performance and the offline algorithm's performance—is bounded.

Unlike traditional worst-case analysis, where the performance of an algorithm is measured only for "hard"inputs, competitive analysis requires that an algorithm perform well both on hard and easy inputs, where "hard" and "easy" are defined by the performance of the optimal offline

For many algorithms, performance is dependent not only on the size of the inputs, but also on their values.

For example, sorting an array of elements varies in difficulty depending on the initial order. Such data-dependent algorithms are analysed for average-case and worst-case data. Competitive analysis is a way of doing worst case analysis for on-line and randomized algorithms, which are typically data dependent.

In competitive analysis, one imagines an "adversary" which deliberately chooses difficult data, to maximize the ratio of the cost of the algorithm being studied and some optimal algorithm.

When considering a randomized algorithm, one must further distinguish between an oblivious adversary, which has no knowledge of the random choices made by the algorithm pitted against it, and an adaptive adversary which has full knowledge of the algorithm's internal state at any point during its execution.

(For a deterministic algorithm, there is no difference; either adversary can simply compute what state that algorithm must have at any time in the future, and choose difficult data accordingly.)

The classic on-line problem first analysed with competitive analysis (Sleator & Tarjan 1985) is the list update problem: Given a list of items and a sequence of requests for the various items, minimize the cost of accessing the list where the elements closer to the front of the list cost less to access. (Typically, the cost of accessing an item is equal to its position in the list.) After an access, the list may be rearranged

Most rearrangements have a cost. The Move-To-Front algorithm simply moves the requested item to the front after the access, at no cost. The Transpose algorithm swaps the accessed item with the item immediately before it, also at no cost.

Classical methods of analysis showed that Transpose is optimal in certain contexts. In practice, Move-To-Front performed much better.

Competitive analysis was used to show that an adversary can make Transpose perform arbitrarily badly compared to an optimal algorithm, whereas Move-To-Front can never be made to incur more than twice the cost of an optimal algorithm.

In the case of online requests from a server, competitive algorithms are used to overcome uncertainties about the future. That is, the algorithm does not"know" the future, while the imaginary adversary (the "competitor")"knows".

Similarly, competitive algorithms were developed for distributed systems, where the algorithm has to react to a request arriving at one location, without "knowing" what has just happened in a remote location

Splay Tree

Splay trees are the self-balancing or self-adjusted binary search trees. In other words, we can say that the splay trees are the variants of the binary search trees.

The prerequisite for the splay trees that we should know about the binary search trees. A splay tree contains the same operations as a Binary search tree, i.e., Insertion, deletion and searching, but it also contains one more operation, i.e., splaying. So. all the operations in the splay tree are followed by splaying.

To search any element in the splay tree, first, we will perform the standard binary search tree operation. After performing the search operation, we need to perform splaying. Here splaying means that the operation that we are performing on any element should become the root node after performing some rearrangements. The rearrangement of the tree will be done through the rotations

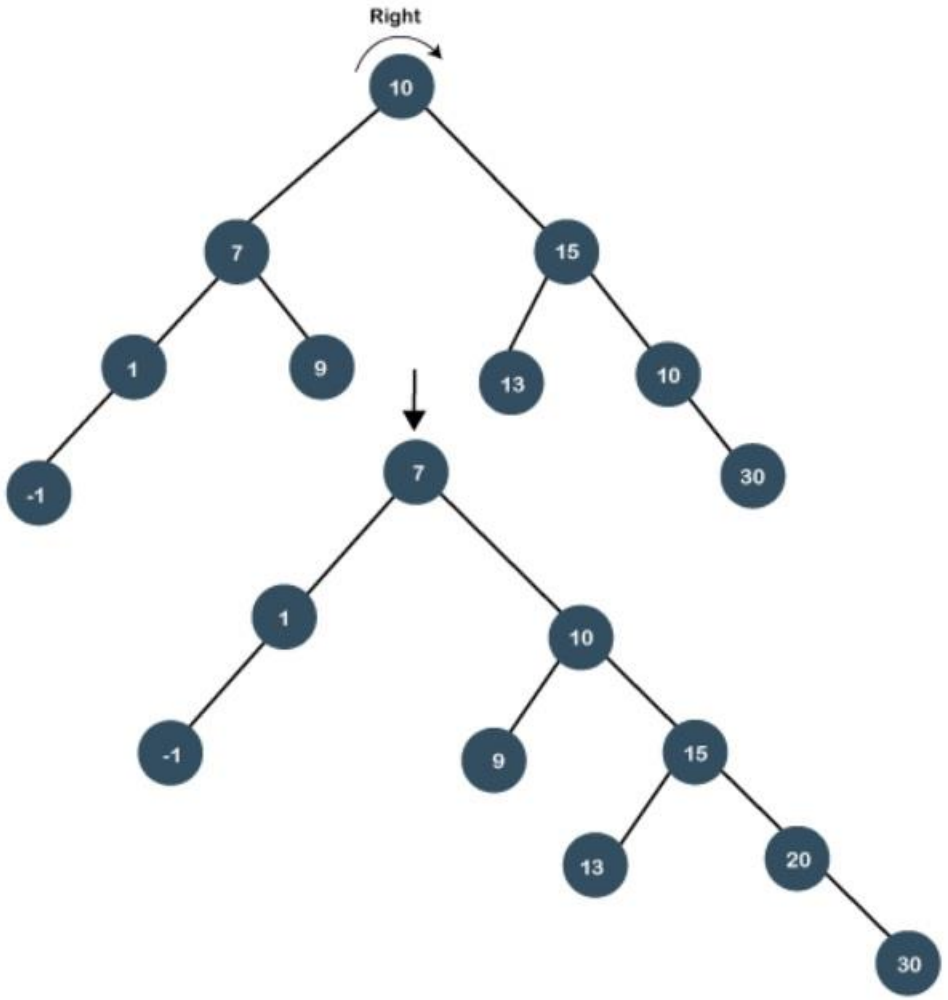
Example of rotations are

1. Zig rotation (Right rotation) In the above example, we have to search 7 element in the tree.

We will follow the below steps:

Step 1: First, we compare 7 with a root node. As 7 is less than 10, so it is a left child of the root node.

Step 2: Once the element is found, we will perform splaying. The right rotation is performed so that 7 becomes the root node of the tree, as shown below:


Right
10
7
15
1
9
13
10
-1
30
7
1
10
-1
9
15
13
20
30

Combination: With probability 4/5 the algorithm serves a request sequence using Bit, and with probability 1/5 it serves a request sequence using TS.

The algorithm Combination is 1.6-competitive against any oblivious adversary.

Many of the concepts shown for self-organizing linear lists can be extended to binary search trees.

The most popular version of self-organizing binary search trees are the splay trees

In a splay tree, after each access to an element x in the tree, the node storing x is moved to the root of the tree using a special sequence of rotations that depends on the structure of the access path.

This reorganization of the tree is called splaying. Splay trees are O(1)-competitive against optimal static search trees On the other hand, the famous splay tree conjecture is still open: It is conjectured that on any sequence of accesses splay trees are as efficient as any dynamic binary search tree

They showed that the amortized asymptotic time of access and update operations is as good as the corresponding time of balanced trees. More formally, in an n-node splay tree, the amortized time of each operation is O(log n)

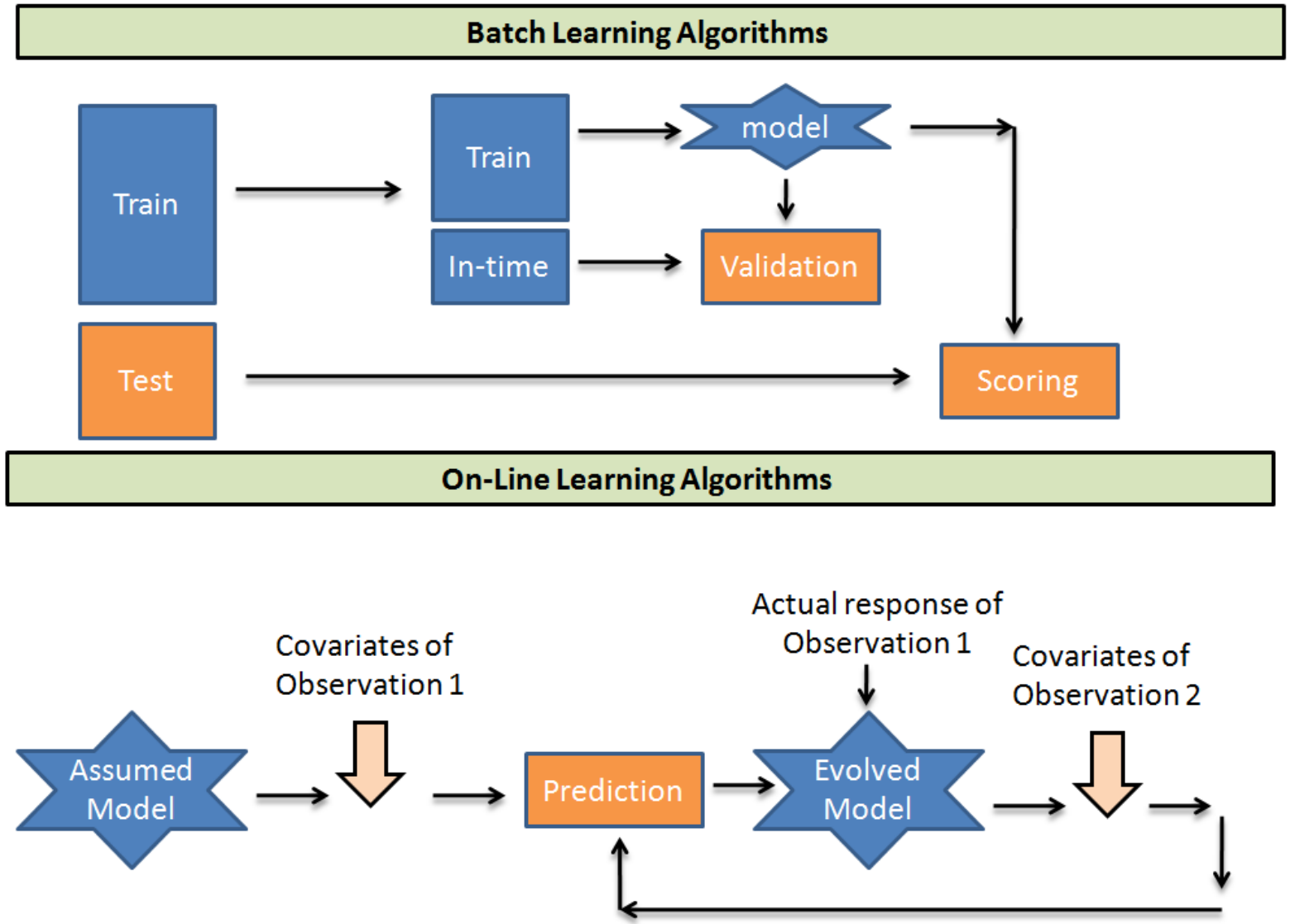
Adversary model

In computer science, an online algorithm measures its competitiveness against different adversary models. For deterministic algorithms, the adversary is the same as the adaptive offline adversary. For randomized online algorithms competitiveness can depend upon the adversary model used.

The three common adversaries are the oblivious adversary, the adaptive online adversary, and the adaptive offline adversary. The oblivious adversary is sometimes referred to as the weak adversary. This adversary knows the algorithm's code, but does not get to know the randomized results of the algorithm.

The adaptive online adversary is sometimes called the medium adversary. This adversary must make its own decision before it is allowed to know the decision of the algorithm. The adaptive offline adversary is sometimes called the strong adversary. This adversary knows everything, even the random number generator. This adversary is so strong that randomization does not help against it.

In contrast to batch learning algorithms, online learning is a method of machine learning for data arriving in a sequential order, where a learner aims to learn and update the best predictor for future data at every step. Online learning is able to overcome the drawbacks of batch learning in that the predictive model can be updated instantly for any new data instances. Thus, online learning algorithms are far more efficient and scalable for large-scale machine learning tasks in real-world data analytics applications where data are not only large in size, but also arriving at a high velocity.


Batch Learning Algorithms
Train
Train
In-time
model
Validation
Test
Scoring
On-Line Learning Algorithms
Covariates of Observation 1
Actual response of Observation 1
Covariates of Observation 2
Assumed Model
Prediction
Evolved Model

there’s an almost limitless number of ways it can go wrong. You might get lots of feedback (examples) from one thing but not another, leading to a skewed classes problem. You might’ve set your learning rate too high, causing your model to forget everything that happened more than a second ago. You might overfit, or underfit. Someone might DDoS your system, screwing up learning in the process. Online learning is prone to catastrophic interference — more so than most other techniques.

In machine learning, the conventional approach is to process data in batches or chunks. Batch learning models assume that all the data is available at once. When a new batch of data is available, these models have to be retrained from scratch. The assumption of data availability is a hard constraint for the application of machine learning in multiple real world applications where data is continuously generated.

Additionally, keeping historical data requires dedicated storage and processing resources, which in some cases might be impractical, e.g. storing the network logs from a data center. A different approach is to treat data as a stream, in other words, as an infinite sequence of items; data is not stored and models continuously learn one data sample at a time.

to process a very large dataset by chunks. For example here: The way we proceed is that we load an image at a time and extract randomly 50 patches from this image. Once we have accumulated 500 of these patches (using 10 images), we run the partial fit method of the online unsupervised algortihms

```
# The online learning part: cycle over the whole dataset 4 times
index = 0
for _ in range(6):
    for img in faces.images:
        data = extract_patches_2d(img, patch_size, max_patches=50,
                                  random_state=rng)
        data = np.reshape(data, (len(data), -1))
        buffer.append(data)
        index += 1
        if index % 10 == 0:
            data = np.concatenate(buffer, axis=0)
            data -= np.mean(data, axis=0)
            data /= np.std(data, axis=0)
            kmeans.partial_fit(data)
            buffer = []
        if index % 100 == 0:
            print('Partial fit of %4i out of %i'
                  % (index, 6 * len(faces.images)))

dt = time.time() - t0
print('done in %.2fs.' % dt)


################################# the results

plt.figure(figsize=(4.2, 4))
for i, patch in enumerate(kmeans.cluster_centers_):
    plt.subplot(9, 9, i + 1)
    plt.imshow(patch.reshape(patch_size), cmap=plt.cm.gray,
               interpolation='nearest')
    plt.xticks(())
    plt.yticks(())


plt.suptitle('Patches of faces\nTrain time %.1fs on %d patches' %
             (dt, 8 * len(faces.images)), fontsize=16)
plt.subplots_adjust(0.08, 0.02, 0.92, 0.85, 0.08, 0.23)

plt.show()
```

Thank You for your Attention!