1 Introduction to Online Algorithms

An algorithm is said to be online if it has to make irreversible decisions for processing an input which is not entirely revealed to it. In traditional offline algorithms, we are given an input and we want to find a solution which is optimal or nearly optimal for that input. When designing offline algorithms, we often focus on achieving algorithms which are optimal (or near-optimal) with respect to their time complexity. We know some problems are NP-hard and hence, to solve them optimally, one needs computing machines which are stronger that common computers modeled as Turing machines (assuming $P \neq NP$). One way to approach NP-hard problems is to devise approximation algorithms with polynomial time complexity which guarantee that their output is a good estimate of the optimal solution. As an example we know that the edge coloring problem (where we want to color edges of a given graph using a minimum number of colors so that edges which share an endpoint have different colors) is an NP-hard problem, i.e., we cannot achieve a coloring with minimum number of colors in polynomial time. However, there are polynomial algorithms which color edges so that the number of colors for any input is at most twice the required number, e.g., a simple greedy algorithm gives such guarantee. We say that such algorithm has an approximation ratio of at most 2.

If we have unbounded computational power, all offline algorithms can be solved optimally. In contrast, an online algorithm cannot achieve an optimal solution, because it does not have the whole input in hand. This lack of knowledge about future is often costly for the online algorithm. To measure this constraint, we compare online algorithms with an optimal offline algorithm. Basically, we assume that both online and offline algorithms have unbounded computational power, and try to figure how much advantage the offline algorithm has. To understand this, consider a minimization problem where we want to minimize a cost (e.g., number of colors in a coloring problem). The competitive ratio of an online algorithm $A$ is defined as the maximum ratio between the cost of the
algorithm and that of the optimal offline algorithm for any input \( \sigma \), i.e.,

\[
c.r.(A) = \max_{\text{input}} \left\{ \frac{\text{cost}_{\sigma}(A)}{\text{cost}_{\sigma}(\text{Opt})} \right\}
\]  

(1)

The above definition implies that the competitive ratio is a worst-case measure, i.e., it only considers the worst-case input to judge performance of an algorithm. This is consistent with similar notions that you have previously seen in analysis of algorithms, e.g., time complexity of an algorithm is the time that it takes to process a worst-case input (e.g., bubble sort might run in \( O(n) \) for some input while it runs in \( O(n^2) \) for the worst-case input).

In order to minimize competitive ratio, the main challenge is to design online algorithms that can perform well in the worst-case input. This often seems like a game between an online algorithm and an adversary. Here, by adversary we mean an enemy who wants to generate the worst-case input for the algorithm. The adversary knows the code of the algorithm and based on that generates a worst-case input. The competitive ratio of the algorithm is the ratio between its cost and that of \( \text{Opt} \) for serving such input.

### 1.1 Ski-rental problem

Let’s start our journey to analysis of online algorithms with the classic ski-rental problem. Assume it is the beginning of the winter, and you plan to have a joyful skiing season. You do not know how many skiing trips you will eventually go to (online component). Also, you do not own any skiing equipment. You can buy the equipment at a one-time cost of \( b > 1 \). Alternatively, you can rent the equipment for a unit-cost of 1 per day. What do you do?

First, let’s see how \( \text{Opt} \) approaches this problem. Since it is offline, it knows the number of skiing trips. Let’s assume it is \( x \). Intuitively, \( \text{Opt} \) either buys the equipment just before the first trip or rents it for the whole season. More precisely, \( \text{Opt} \) compares \( x \) and \( b \): if \( x \leq b \), the number of skiing trips is less than the cost of the equipment. In this case, \( \text{Opt} \) always rents and incurs a total cost of \( x \). In case \( x > b \), \( \text{Opt} \) buys the equipment at the beginning and pays a one-time cost of \( b \). Regardless, for the cost of \( \text{Opt} \), we have

\[
\text{cost}(\text{Opt}) = \min\{x, b\}
\]  

(2)

In reality, you need an online strategy which does not know \( x \). What that strategy can be? First, assume an online algorithm \( A_1 \) which buys the equipment on day 1. The adversary, who wants to generate worst-case input, asks for an input (season) where there is no further skiing trip, i.e., \( x = 1 \). In that case, from Equation 2, the cost of \( \text{Opt} \) is \( \min\{1, b\} = 1 \). The cost of the algorithm is \( b \). The competitive ratio would be:

\[
c.r.(A_1) = \frac{b}{1} = b
\]  

(3)

Next, assume a second online strategy \( A_2 \) which never buys the equipment. In this case, the adversary creates a season with \( x = nb \) trip, where \( n \) is an arbitrary large integer. The cost of the adversary is \( \min\{nb, b\} = b \). The algorithm, however, keeps renting and incurs a cost of \( nb \). The competitive ratio would be:

\[
c.r.(A_2) = \frac{nb}{b} = n
\]  

(4)

\(^{1}\)We always use \( \text{Opt} \) to denote the optimal offline algorithm, and \( \sigma \) to denote an arbitrary input sequence.
Next, consider a third strategy which start with renting for the first $b-1$ days and then buys the equipment at the beginning of the $b$'th day. Note that the algorithm is oblivious to the value of $x$. The worst-case scenario that adversary can come up to is when to create a season where the season ends just after when $A_3$ buys the equipment, i.e., when $x = b$. The cost of $\text{Opt}$ would be $\min\{x, b\} = b$. The cost of the algorithm, however, is

$$\frac{1 + 1 + \ldots + 1 + b}{b-1\text{times}} = 2b - 1$$

So, the competitive ratio is $\frac{2b-1}{b} = 2 - 1/b$, which takes its maximum value of 2 for large values of $b$. Consequently, for any value of $b$ and for any sequence, the competitive ratio of $A_3$ is no more than 2. We call such algorithm a competitive algorithm, i.e., the competitive ratio of the algorithm is constant and does not depend of the input. Note that $A_1$ and $A_2$ are not competitive in this example.

## 2 Path-cow Problem & Doubling Technique

In practice, we often with situations where you search for a target without knowing on which ‘direction’ you should search. The easiest form of this problem is modeled with path-cow problem.

A cow faces a fence, infinite in both left and right directions. She wants to find a hole in order to get to the green pasture on the other side. The cow’s online strategy specifies the path traveled in search of the hole. The cow’s objective is to minimize the distance traveled.

Assume $u$ indicates the distance of the cow and the hole, i.e., the hole is located at distance $u$ on the left or right direction. Note that the cow does not know the value of $u$ neither the direction that it is located. The optimal offline algorithm just needs to know the direction: it moves the cow in the right direction, and it finds the hole at optimal distance $u$. So, the cost of $\text{Opt}$ is $u$.

An online algorithm can gamble on one direction and just takes one direction, e.g., it moves all the way to the right. However, in case the hole is on the left, the hole cannot be found using this strategy. Consequently, a smart cow should gradually extend the explored interval of the fence. Basically, it has to alternate between left and right intervals. Such algorithm can be formulated by the following general approach:

- Starting at the origin, go right for a distance of $d_0$.
- Go back to the origin, go left for distance $d_1$.
- Go back to the origin, go right for distance $d_2$.
- Continue accordingly for $d_3, \ldots, d_k$ until the hole is found.

Assume the hole is found when the cow moves from the origin and intends to take a move of length $d_k$. This implies that it has previously gone a distance of $d_i$ to a direction and moved back to the origin for $i < k$, i.e., a total distance of $2d_0 + 2d_1 + \ldots + 2d_{k-1}$. On its last move from the origin for a distance of $d_k$, the cow finds the hole and stops, i.e., the last move has a cost of $u$. So, for the cost of any smart-cow algorithm $A$ we have

$$\text{cost}(A) = 2d_0 + 2d_1 + \ldots + 2d_k + u$$

Recall that the cost of $\text{Opt}$ is $u$. The competitive ratio of $A$ would be:
Figure 1: A smart cow algorithm with $d_i = 2^i$. The worst-case happens when $u = d_{k-2} + \epsilon$.

$$\text{c.r.}(A) = \frac{2d_0 + 2d_1 + \ldots + 2d_{k-1} + u}{u} = 1 + \frac{2(d_0 + d_1 + \ldots + d_{k-1})}{u}$$

If you are the adversary, where you place the hole? W.l.o.g. assume in the last three moves the cow moves a distance of $d_{k-2}$ to the right, then back to the origin, then a distance of $d_{k-1}$ to the left, then back to the origin, and finally a distance of $u$ to the right, where it finds the whole. The worst case for the algorithm happens when the hole was just at distance $d_{k-2} + \epsilon$ on the right (see Figure 1). Note that if we make $\epsilon$ larger, the cost both algorithm and OPT increases by the same amount, and hence the ratio between them decreases. So, the adversary chooses $\epsilon$ to be as small as possible. So, in the worst case, we have $\text{OPT}(\sigma) = u = d_{k-2} + \epsilon$ for small $\epsilon$. The competitive ratio would be:

$$\text{c.r.}(A) = 1 + \frac{2(d_0 + d_1 + \ldots + d_{k-1})}{d_{k-2} + \epsilon}$$

Note that the above framework does not specify an online algorithm. For that, we should indicate what the values of $d_i$’s should be. A natural approach is to double the explored area in each step, i.e., go right for 1 unit, back to origin, 2 units to the left, back to origin, etc. This implies that the value of $d_i$ is $2^i$. From Equation 8, the competitive ratio of the algorithm becomes:

$$\text{c.r.}(A) = 1 + \frac{2(1 + 2 + \ldots + 2^{k-1})}{2^{k-2} + \epsilon} = 1 + \frac{2^k - 1}{2^{k-2} + \epsilon} = 1 + \frac{4 \cdot (2^{k-2} + \epsilon) - 1 - 4\epsilon}{2^{k-2} + \epsilon} = 9 - \frac{1 + 4\epsilon}{2^{k-2} + \epsilon}$$

The adversary chooses $k$ to be large so that the above ratio becomes larger; in this case, this ratio converges to 9. So, the doubling algorithm results in a 9-competitive algorithm for the path-cow problem. One might ask why we doubled the distance at each step? why not triple the distance? or increase the distance by a factor of 1.5 at each step? Each of these algorithms also result in a competitive algorithm; however, the competitive ratio becomes larger. In fact, the best deterministic algorithm for path-cow problem is the smart-cow with doubling steps.\(^2\)

### 2.1 Randomized Online Algorithms

In the analysis of online algorithms, we should always distinguish deterministic algorithms from randomized algorithms. In case of a deterministic algorithms, the adversary knows the exact decisions made by the algorithm and creates a worst-case input accordingly. For example, in the case of the doubling algorithm in the previous section, the adversary knows the direction of the first move.

\(^2\)The proof is a bit involved and we skip it in this course
made by the algorithm and also the exact path explored by the algorithm; hence, it can select the worst-case inputs by provided with this knowledge. One way for the algorithm to avoid these worst-case scenarios is to hide its decisions from the adversary by taking some random decisions. The adversary still knows how the algorithm works (i.e., it knows the code of the algorithm); however, it does not know the content of the random bits used by the algorithm (i.e., it does not know the outcome of the coin flips in the run-time).3

Let’s continue our exploration of the path-cow problem with a randomized algorithm. Recall that the doubling algorithm makes its first move to one direction, second to the opposite direction, etc. There, without loss of generality, we assumed that the first move was to the right. The particular choice of the first move does not make any difference for a deterministic doubling algorithm because the adversary knows the first move of the algorithm and creates the worst-case scenario accordingly. Now consider a randomized algorithm that flips a coin at the beginning to decide the direction of the first move, and applies the doubling technique after that. Basically, this algorithm randomly selects one of the deterministic algorithms that mirror each-other, that is, a blue algorithm whose first move to the right and a red algorithm whose first move is to the left (see Figure 2).

To analyze a randomized algorithm, we compare its expected cost with the cost of an optimal algorithm. In the case of the randomized path-cow algorithm described above, the algorithm acts as the blue algorithm with a chance of 1/2 and as the red algorithm, again with a chance of 1/2, i.e., the expected cost is (cost_{blue} + cost_{red})/2. In order to calculate the expected cost, we find the summation of the costs of both blue and red algorithm. Note that if the blue explores distance $2^i$ on a direction, the red algorithm explores a distance of $2^{i+1}$ on that direction. In other words, when calculating the sum of both algorithms, all powers of 2 are explored in each direction (each power by one algorithm). W.l.o.g. assume the hole is located on the right. The total distance moved by blue and red on the left is $2(1 + 2 + \ldots + 2^{k-1}) = 2 \cdot (2^k - 1)$. Similarly, the total distance moved by the algorithms on the right before the last move is $2(1 + 2 + \ldots + 2^{k-2}) = 2 \cdot (2^{k-1} - 1)$. In the last move, both algorithms moved a distance of $u$, which sums to $2u$, and stop after finding the hole. So, the expected cost of the algorithm is:

$$\text{Expected-cost} = \frac{\text{cost(blue)} + \text{cost(red)}}{2} = \frac{\text{total-cost-left} + \text{total-cost-right}}{2} = \frac{(2 \cdot (2^k - 1)) + (2 \cdot (2^{k-1} - 1) + 2u)}{2} = 2^k + 2^{k-1} + u - 2$$

The competitive ratio of the randomized algorithm would be:

$$\text{c.r.}(A) = \frac{2^k + 2^{k-1} + u - 2}{u} = 1 + \frac{2^k + 2^{k-1} - 2}{u}$$

As before, to maximize this ratio, it is best for the adversary to place the hole just a bit after the last distance moved by one of the algorithms, e.g., $u = 2^{k-2} + \epsilon$ for a small $\epsilon$. To see the rational for this choice of the adversary, assume $u$ is at distance $2^{k-2}$, i.e., $u = 2^{k-2} + \epsilon$. Increasing $\epsilon$ will increase the expected cost of the randomized algorithm and the cost of Opt by the same value, and hence the ratio between these two costs decreases. So, the adversary prefers $\epsilon$ be as small as possible, i.e., $2^{k-2} + \epsilon$. Consequently, the competitive ratio of the algorithm would become:

3This type of adversary which is oblivious to the random bits of the algorithm is called oblivious adversary. There are other types of adversaries for analysis of randomized algorithms which are beyond the scope of this course.
Figure 2: An illustration of the randomized doubling algorithm. The worst-case happens when $u = d_{k-2} + \epsilon$.

As before, to maximize this ratio, the adversary chooses large values of $k$, for which the ratio converges to 7. In summary, the competitive ratio of the doubling algorithm improves from 9 to 7 by applying a simple randomization on the first direction. In fact, the randomized algorithm that we studied is barely random because it uses a constant number of random bits (here only one). Later in the course, we learn more about barely random algorithms. More complex randomized algorithms achieve better competitive ratios. The best randomized algorithm for the path-cow problem has a competitive ratio of 4.591, and this is the best that a randomized algorithm can achieve.\footnote{The competitive ratio 4.591 is deduced as $\gamma + 1$ where $\gamma$ is the solution to $\gamma \ln \gamma = \gamma + 1$.}

2.2 Searching under Uncertainty

A typical cow is not necessarily smart, and she does not care about competitive analysis. However, the path-cow problem is a good way to model a variety of search problems. Assume a robot looking for a target. It faces a two-way intersection, and it has to decide which side it should take. This situation is similar to the path-cow problem. In fact, it can be thought as searching in an unbounded path. Now, assume, instead of a two-way intersection, the robot faces a $w$-way intersection. This is similar to searching on a star with unbounded rays (see Figure 3). One way to approach this problem is to used doubling technique as before: move a distance of 1 on the first ray, back to the origin, distance 2 on the second ray, back to origin, etc. One can show that the competitive ratio of this algorithm grows exponentially with $w$. Assume that, instead of doubling the explored distances, we grow the distances by ratio $w/(w-1)$, i.e., first we go distance 1 in the first ray, distance $w/(w-1)$ for the second ray, $(w/w-1)^2$ for the third ray, etc. This algorithm results in a competitive ratio of $1 + 2^{w/(w-1)}$. For large values of $w$, this ratio is roughly $1 + 2e(w-1)$, which implies that the competitive ratio grows linearly with $w$ ($e \approx 2.71$ is the base of natural logarithm). Indeed, this algorithm is the best possible randomized algorithm for searching on stars.

2.3 Advice Complexity of Online Algorithms

Randomization is one way to improve competitive ratio of online algorithms. Another way is to provide algorithm with some advice about the unknown input. The advice can encode any information that the algorithm desires to know about the input. We often assume the advice is generated
by a benevolent offline oracle with unbounded computational power. The advice is written on an 
advice tape which is accessible by the algorithm before it takes any decision. When we talk about 
one-line algorithms with advice, we need to indicate how the algorithm works and what information 
the advice bits carry. Clearly, more advice results in better algorithms. We are often interested in 
algorithms with a small number of bits of advice. The advice complexity of an online algorithm is 
the number of advice bits that it reads from the advice tapes.

As the first step in studying advice complexity of online algorithms, let’s think about path-cow 
problem. What kind of advice can improve the competitive ratio? Consider an online algorithm 
that receives only one bit of advice indicating the direction at which the hole is located. Such advice 
is sufficient to solve the problem optimally. You can observe that a single bit of advice can be 
dramatically useful. Unfortunately it is not the case for all problems.

As another example, consider the online search problem under stars. How advice can help us in 
approaching this problem? Assume that we are given 1 bit of advice. We can partition the $w$ rays 
into two groups of size $w/2$ and, using one bit, indicate in which group the target is located. Then, 
we use the algorithm that jumps with a factor $w/2$ on the indicated group. The competitive ratio 
of such algorithm would be $1 + 2e(w/2 - 1)$, which is an improvement over $1 + 2e(w - 1)$ of 
the algorithm that does not use advice. More generally, given $k$ bits of advice, we can partition rays 
into $2^k$ groups of size roughly $w/(2^k)$. The competitive ratio will be improved to $1 + 2e(w/2^k - 1)$.

Advice complexity is a new topic in design and analysis of online algorithms. In the remainder 
of the course, we learn about advice complexity of a variety of problems. There is, however, a lot of 
unstudied problems which provide interesting opportunities for selecting research projects.

## 3 Online Bidding

In this section, we learn about the online bidding problems which helps us dig further into random-
ization and lower-bound techniques for analysis of online algorithms.

Assume an online algorithm facing with an unknown target value $u$, and has to submit a sequence 
$d_0, d_1, \ldots, d_k$ of bids until one is larger than or equal to $u$. Clearly, the bids that the algorithms 
submits form an increasing sequence, and we have:

\[ d_0 < d_1 < \ldots < d_{k-1} < u \leq d_k \]

The cost of the algorithm is the total size of all bids that it submits, i.e., $d_0 + d_1 + \ldots + d_k$. This 
problem has similarities with the path-cow problem, and a natural algorithm is to double the bids
at each step, i.e., define $d_i = 2^i$. The cost of such doubling algorithm would be:

$$\text{cost}(A) = 1 + 2 + 4 + \ldots + 2^{k-1} + 2^k = 2^{k+1} - 1$$

The adversary selects the target to be just a bit more than the second to last bid, i.e., $2^{k-1} + \epsilon$ (why?). So, the competitive ratio of the doubling technique is:

$$\text{c.r.}(A) = \frac{2^{k+1} - 1}{2^{k-1} + \epsilon} = 4 - \frac{1}{2^{k-1} + \epsilon}$$

This ratio approaches 4 for large values of $k$, i.e., the competitive ratio of the doubling algorithm is at most 4 for the online bidding problem. Can we do better? In what follows, we show that we cannot. We prove the following:

**Theorem 1.** No deterministic algorithm can achieve a competitive ratio better than 4 for the online bidding problem.

**Proof.** Consider otherwise, i.e., assume there is a deterministic online algorithm $A$ with competitive ratio $\alpha < 4$. Let $d_0, d_1, \ldots$ denote the bids of $A$. Define $s_i \equiv d_0 + d_1 + \ldots + d_i$, i.e., $s_i$ is the cost of $A$ after making $i$ bids. Note that we have $d_n = s_n - s_{n-1}$. Moreover, define $y_i \equiv \frac{s_i}{d_i}$. Assume the adversary creates a target value of $d_n + \epsilon$, where $n$ is a large integer. The cost of $A$ for this input would be $d_0 + \ldots + d_n + d_{n+1} = s_{n+1}$. The cost of $\text{OPT}$ is $d_n + \epsilon$. The ratio between the two costs is $\frac{s_{n+1}}{d_{n+1}}$. Since $n$ is large, we can ignore the $\epsilon$ and write the above ratio as roughly $\frac{s_n + 1}{d_n}$. Since the competitive ratio of $A$ (the maximum ratio between the cost of $A$ and $\text{OPT}$ for any input) is $\alpha$, we should have $\frac{s_n + 1}{d_n} \leq \alpha$. We can write:

$$s_{n+1} \leq \alpha d_n \quad \text{divide by } d_n$$

$$\frac{s_{n+1}}{s_n} \leq \alpha$$

$$s_{n+1} \leq \alpha s_n - s_{n-1} \quad \text{by definitions of } s_n \text{ and } y_{n-1}$$

$$\frac{s_{n+1}}{s_n} \leq \alpha$$

$$y_n \leq \alpha(1 - \frac{1}{y_{n-1}})$$

For any value of $x$, we have $(x - 2)^2 = x^2 - 4x + 4 \geq 0$, which gives $1 - 1/x \leq x/4$. So for $x = y_{n-1}$, we get $1 - \frac{1}{y_{n-1}} \leq y_{n-1}/4$. Plugging this in the above inequality we get:

$$y_n \leq \alpha(y_{n-1}/4) = (\alpha/4)y_{n-1}$$

Writing the same inequality for $y_{n+1}$ and $y_{n+2}$ we get:

$$y_{n+1} \leq (\alpha/4)y_n \leq (\alpha/4)^2 y_{n-1}$$

$$y_{n+2} \leq (\alpha/4)y_{n+1} \leq (\alpha/4)^3 y_{n-1}$$

and similarly,

$$y_{n+i} \leq (\alpha/4)^{i+1} y_{n-1}$$

Since $\alpha < 4$, after each bid, the value of $y_{n+i}$ is decreased by the fraction $\alpha/4$. Consequently, for a sufficiently large $i$, we will have $y_{n+i} < 1$ which, by definition of $y_{n+i}$, implies $\frac{s_{n+i}}{s_{n+i-1}} < 1$, i.e., $s_{n+i} < s_{n+i-1}$. In other words, we get:

$$d_0 + d_1 + \ldots + d_{n+i-1} + d_{n+i} < d_0 + d_1 + \ldots + d_{n+i-1} \Rightarrow d_{n+i-1} < 0$$

However bids are supposed to be all positive values. As a result, $d_{n+i-1} < 0$ is a contradiction, i.e., the initial assumption that $\alpha < 4$ is wrong, and the proof is complete.
In the above proof, we used \(1 - \frac{1}{y_i} \leq y_i/4\), and the equality holds when \(y_i = 2\). One might wonder which algorithm has \(y_i = 2\). To see that, note \(y_i = \frac{2x_i}{x_i-1} = 1 + \frac{d_i}{d_1 + \ldots + d_{i-1}}\). So, in order to have \(y_i = 2\), we should have \(d_i = d_1 + \ldots + d_{i-1}\). The doubling algorithm satisfies this requirement since \(1 + 2 + \ldots + 2^k \approx 2^{k+1}\). This might give you an intuition on why the doubling algorithm achieves the same competitive ratio of 4 as given by the above lower bound.

**Corollary 1.** The doubling algorithm is the optimal deterministic algorithm for the online bidding problem.

### 3.1 Randomized Online Bidding

We learned that the doubling algorithm has a competitive ratio of 4, and it is the best that a deterministic algorithm can do for online bidding. In this section, we study a randomized algorithm which turns out to be the optimal randomized algorithm for the online bidding problem.

The first step in this algorithm is to select a value uniformly at random from the range \([0, 1]\). Before continuing, let’s see how we can ‘play’ with such random variable. Let \(X\) denote a random variable in \(U[0,1]\), i.e., \(X\) can uniformly take a value between 0 and 1. Let \(p\) be an arbitrary positive value. Assume we increase \(X\) by units of 1 until at some point its value becomes in the target range \([p, p+1]\).

For example, assume our initial value is randomly selected to be \(x = 0.4\), and we have \(p = 5.3\). After increasing \(x\) by \(k = \lfloor p \rfloor = 5\) units, its value becomes 5.4 \(\in [5.3, 6.3]\). If the initial value was selected as \(x = 0.2\), we had to increase \(x\) by \(k = \lceil p \rceil = 6\) units to get \(x = 6 = 6.2 \in [5.3, 6.3]\). More generally, let \(t = p - \lfloor p \rfloor\) (in the above example \(t = 0.3\)). For a random value \(x\) in the range \([0, t]\), after increasing \(k = \lfloor p \rfloor = p - t + 1\) units, we get \(x + k = \lfloor p - t + 1, p + 1 \rfloor\). If the value of \(x\) is in \([t, 1]\), after increasing \(k = \lfloor p \rfloor = p - t\) units, we get \(x + k = (p, p + 1)\). Regardless, after increasing \(x\) by \(k\) units, its value becomes uniformly distributed in the range \([p, p + 1]\). We get the following observation:

**Observation 1.** Let \(x\) be a random number in \(U[0,1]\) and \(p\) be an arbitrary positive value. Assume we increase \(x\) for \(k\) times by units of one until \(x + k\) becomes at least \(p\). The random variable \(x + k\) is uniformly distributed in the range \([p, p + 1]\).

Recall that the randomized algorithm starts with selecting a random number \(X\) in \([0, 1]\). The bids of the algorithm are \(d_0 = e^X, d_1 = e^{X+1}, d_2 = e^{X+3}, \ldots, d_{k-1} = e^{X+k-1}, d_k = e^{X+k}\). Basically, the algorithm starts with the first bid \(e^X\) and continues its bidding by multiplying its previous bid by a constant factor of \(e\). Recall that \(e \approx 2.71\) is the base of the natural logarithm. Note that, unlike the deterministic case, the adversary does not know the actual bids (all depends on the first random guess), and hence it cannot create a worst-case scenario by selecting a target that is just a bit more than the previous bid of the algorithm (again, it is oblivious to the previous bid).

In order to analyze the algorithm, we calculate the expected value of \(J = \frac{d_k}{d_{k-1}}\), i.e., the ratio between the expected value of the last bid of the algorithm and the target value. Basically, we want to know how much the last bid is larger than the target value. Define \(p \equiv \ln u\), i.e., \(u = e^p\). Assume the last bid of the algorithm is its the \(k\)’th guess. The value of such bid is \(d_k = e^{X+k}\). So, we have \(u = e^p \leq e^{X+k} < e^{p+1}\). The ratio of \(J = \frac{d_k}{d_{k-1}}\) becomes \(\frac{e^{X+k}}{e^{X+k-1}} = e^{X+k-p}\).

Note that the algorithm has been bidding \(e^X, e^{X+1}, \ldots\) until \(e^{X+k}\) is larger than \(e^p\). This implies that the algorithm has been using sequence \(X, X+1, \ldots\) until \(X + k\) becomes at least \(p\). By Observation 1, the value of \(X + k\) is uniformly distributed between \([p, p + 1]\), i.e., \(X + k - p\) is uniformly distributed in \([0, 1]\). Recall the the value of \(J\) is \(e^{X+k-p}\). We conclude that \(J\) is a random variable distributed as \(e^Y\) where \(Y = X + k - p\) is uniformly distributed in \([0, 1]\). The expected value
of \( J \) would be \( \int_0^1 e^Y dY = e \). So, the last guess of the algorithm is expected to be \( e \) times larger than the target value \( u \). Compare this with the last guess of the doubling algorithm, which is twice larger than \( u \) in the worst case. So, we see an improvement from 2 to \( e - 1 \) for the randomized algorithm.

Next, we calculate the competitive ratio of the randomized algorithm. Recall that the bids of the algorithm are \( d_0 = e^X, d_1 = e^{X+1}, d_2 = e^{X+3}, \ldots, d_{k-1} = e^{X+k-1}, d_k = e^{X+k} \) and we know the value of \( J = \frac{d_k}{u} \) is expected to be \( e \). Rewriting the cost of the algorithm, we get:

\[
\text{cost}(A) = e^X + e^{X+1} + \ldots + e^{X+k} = e^X (1 + e + \ldots + e^k) = e^X \cdot e^k (1/e^k + 1/e^{k-1} + \ldots + 1/e + 1) = e^{X+k} \cdot e = d_k \cdot e
\]

The last equality follows from definition of \( d_k \). For the competitive ratio of the algorithm we have:

\[
\text{c.r.}(A) = \frac{E[\text{cost}(A)]}{u} = \frac{E[d_k e/(e - 1)]}{u} = E[d_k u] \cdot e/(e - 1) = E[J] \cdot e/(e - 1) = (e - 1) \cdot e/(e - 1) = e
\]

So, the competitive ratio of the algorithm is \( e \), which is an improvement over the best competitive ratio of 4 that the best deterministic algorithm (the doubling algorithm) can achieve. This is another evidence that randomization can help improve competitive ratio of online algorithms.\(^5\)

### 3.2 Online Clustering Problem

Sometimes algorithmic solutions for one online problem can be used as black-box tools in designing algorithms for another problem. In this section, we see one such example, where we use online bidding algorithms as black-box components in designing online clustering algorithms.

In a clustering problem, we would like to partition a set of points in to a set of clusters so that points that appear in the same cluster are closer to each other. The diameter of a cluster is the maximum distance between any two points in the cluster. There are two flavors of clustering problems: we either have a fixed number of clusters and want to minimize the maximum diameter of clusters or we are given a fixed bound on the diameter and want to minimize the number of used clusters. In this section, we are mostly concerned with the first variant of clustering in the online setting, which is defined as follows.

**Definition 1.** In the online clustering problem, the goal is to partition an online sequence of points into \( k \) clusters. The goal is to achieve a clustering in which the maximum diameter is minimized. When a point arrives, the algorithm should assign it to an existing cluster. In doing so, is allowed to merge existing clusters but it cannot divide them.

In the above definition, it is critical to allow online algorithms to ‘merge’ existing clusters; otherwise, one can show that no algorithm can be competitive. For example, assume \( k = 3 \) and the input starts with three points of pairwise distance \( b \). If an online algorithm decides to place these points in 3 different clusters, the adversary sends next point at a minimum distance \( 2b \) from the first three points. Since no merging is allowed, the algorithm has to place the new point in one of the existing clusters, which results in diameter of at least \( 2b \). The optimal algorithm, however,

\(^5\)As promised in the class, there will not be a question from this section of the notes. However, students are encouraged to understand it for their projects and bonus questions.
Figure 4: If we do not allow merging, no online algorithm can be competitive. Assume the three black points appear at pairwise distance $b$. If they are placed in different clusters, the red point arrives at distance $2^k$ and the competitive ratio becomes $\frac{2^k}{b}$. If two of the black points are placed in the same cluster, instead of the red point, blue points arrive at distance $\log b$ of a black point and the competitive ratio becomes $\frac{2}{\log b}$.

places two of the first point in the same cluster and opens a new cluster for the fourth point. This results in diameter of $b$ for Opt. The competitive ratio becomes $\frac{2^k}{b}$, which is unbounded for large values of $b$. Next, assume the algorithm places two of the first two points in the same cluster, i.e., there is a cluster of diameter $b$ in the algorithm’s clustering. In this case, the adversary sends the remaining points in a way that each point is at distance $\log b$ of one of the first three points. In this case, the optimal algorithm opens a cluster for any of the first three points to achieve a diameter $\log b$. The competitive ratio becomes $\frac{b}{\log b}$, which is unbounded for large values of $b$. (See Figure 4 for an illustration). In the remainder of this section, we show that if the algorithm has the ‘power’ to merge some existing clusters, we can achieve a constant competitive ratio.

Consider the following algorithm ClusterBid which uses a sequence $d_0, d_1, \ldots$ as its parameter. Each value of $d_i$ is associated with a ‘phase’ of the algorithm. Recall that at each given point of time, a clustering should maintain at most $k$ clusters. ClusterBid algorithm defines a ‘center’ for each cluster so that at any given phase $i$, the mutual distance between any pair of centers is at least $d_i$ (see figure 5).

Next, we describe how ClusterBid works. Assume we are at phase $i$ and a new point $p$ arrives. The algorithm needs places $p$ into a new cluster while maintaining the ingredient that the mutual distance of centers is at least $d_i$ at phase $i$. For that, it uses the following rules:

- Rule I) If distance of $p$ to any center $c_x$ is at most $d_i$, add $p$ to the cluster of $c_x$. In this case, the number of clusters and the mutual distances of centers are not changed.

- Rule II) Else, if there are fewer than $k$ clusters, create a new cluster which only contains $p$ as a single point. In this case, the number of clusters is incremented but remains at most $k$. The mutual distance between centers is also at least $d_i$ (otherwise, $p$ should have been treated by Rule I).
- **Rule III** Else, we end phase $i$ and start phase $i + 1$. For that we create a temporary $(k + 1)$'th cluster which only contains $p$. At this point, we are in an invalid state (there are more than $k$ clusters). To fix this, we start phase $i + 1$ by merging some clusters. We process centers one by one; when processing center $c_x$, merge it with any cluster with center $c_y$ at distance at most $d_{i+1}$ from $c_x$. In this case, $c_y$ becomes the center of the new cluster (and we do not process $c_y$ any more). Note that after processing all centers, the mutual distance of all remaining centers is at least $d_{i+1}$ (otherwise, their clusters would have been merged). At the end of the above process, if a merger occurred, the number of clusters will be at most $k$ (it will be less than $k$ if more than one merger occurs). If no merger occurs, we end phase $i + 1$ and start phase $i + 2$, and this continues.

In what follows, we show that ClusterBid is a competitive algorithm. Assume the algorithm is at phase $k + 1$ when the sequence ends. We claim that in an optimal clustering, there is a cluster of diameter at least $d_k$. This is because, when ClusterBid is starting the $(k + 1)$'th phase, it has $k + 1$ clusters (including the temporary one) whose centers are at mutual distance of at least $d_k$. In other words, the input includes $k + 1$ points which are at mutual distance of $d_k$. Since at most $k$ clusters are permitted, Opt has to place two of these points into the same cluster, i.e., the diameter of a cluster in Opt is at least $d_k$.

Next, we analyze the cost of the ClusterBid solution. Instead of focusing on the 'diameter', let’s compute the ‘radius’ of the solutions created by ClusterBid. The radius of a cluster is the maximum distance between the center and any other point in the cluster. By triangle inequality, the diameter of a cluster is no more than twice its radius. So, in order to find an upper bound for the maximum diameter at phase $k + 1$, it suffices to find an upper bound for the maximum radius. Note that phase $k + 1$ starts with merging some clusters of phase $k$. Let $r_k$ denote the maximum radius of any cluster at the end of phase $k$; we express $r_{k+1}$, the same value for phase $k + 1$, in terms of $r_k$. Assume two clusters with centers $c_x$ and $c_y$ are merged, and $c_z$ is the center of the new cluster (see Figure 6). After the merger, the distance of $c_z$ to the old points in its cluster (green points in the figure) remain at most $r_k$. The distance of $c_z$ to any point $q$ that formerly belonged to cluster of $c_y$ (red points) is at most $d_{k+1} + r_k$. This follows from the triangle inequality for the triangle formed by $c_x$, $c_y$, and $q$, i.e., we have $d(c_y, q) \leq d(c_y, c_x) + d(c_x, q)$. Meanwhile, we have $d(c_y, x) \leq d_{k+1}$ (because the algorithm decides to merge the two cluster) and $d(x, q) \leq r_k$ (because
At the beginning of phase $k+1$, the maximum radius is increased by at most $d_{k+1}$. $r_k$ is the maximum radius at phase $k$). To summarize, the maximum radius at the beginning of phase $k+1$ is increased by at most $d_{k+1}$ compared to the maximum radius at phase $k$. The maximum radius does not change throughout phase $k+1$ since Rule I and Rule II do not change the maximum radius. In summary, we can write $r_{k+1} \leq d_{k+1} + r_k$. Using the same argument for the $k$'th phase, we can write $r_k \leq d_k + r_{k-1}$. So, we have $r_{k+1} \leq d_{k+1} + d_k + r_{k-1}$. More generally, applying this argument for all phases, we will get $r_{k+1} \leq d_{k+1} + d_k + \ldots + d_1 + d_0$. Recall that the maximum diameter is at most the maximum radius, i.e., the cost of the ClusterBid at phase $k+1$ is at most $2(d_{k+1} + d_k + \ldots + d_1 + d_0)$. As we saw earlier, the cost of the Opt at phase $k+1$ is at least $d_k$. Consecutive, the competitive ratio of the algorithm is at most

$$c.r.(\text{ClusterBid}) \leq \frac{2(d_{k+1} + d_k + \ldots + d_1 + d_0)}{d_k}$$

We can use a bidding algorithm to set the values of $d_0, d_1, \ldots, d_{k+1}$ for the ClusterBid algorithm. The cost of the bidding algorithm will be $d_0 + d_1 + \ldots + d_{k+1}$, which is half of the cost of ClusterBid. The cost of an optimal algorithm for the bidding problem is at least $d_k$, which is the same as the cost of an optimal algorithm for the clustering problem. We can conclude the following theorem:

**Theorem 2.** Given an algorithm with competitive ratio of $c$ for the online bidding problem, the ClusterBid algorithm that makes uses the same bids (for its sequence of $d_i$'s) achieves a competitive ratio of $2c$ for the clustering problem.

In particular, if we use the doubling algorithm to define the values of $d_i$'s, the competitive ratio of the ClusterBid becomes at most 8. Similarly, if we use the optimal randomized algorithm for bidding, the competitive ratio of the resulting randomized ClusterBid becomes at most $2c$.

### 3.2.1 Other Variants of Clustering Problem

Consider the online clustering problem where points arrive in a 3-dimensional space. As before we want to minimize the diameter (maximum distance between any two point in the 3-dimensional
space). As an exercise, think about the competitive ratio of the extension of ClusterBid in the 3-dimensional space for this problem.

Other variants of the clustering problems can also be studied in the online setting. For example, assume we are given a value $d$ and need to minimize the number of clusters for a sequence of online points when the diameter of each cluster has to be at most $d$. A simple variant of this problem is a 1-dimensional version when all points are guaranteed to be on a line. What is the competitive ratio of the best algorithm for this problem? How advice can help? What about the general 2-dimensional or 3-dimensional problems for this setting of clustering? These are all potential topics for your research projects!