Part V

Online Algorithms and Competitive Analysis

Chapter 14

Warm up: Ski rental

We now study the class of online problems where one has to commit to provably good decisions as data arrive in an online fashion. To measure the effectiveness of online algorithms, we compare the quality of the produced solution against the solution from an optimal offline algorithm that knows the whole sequence of information a priori. The tool we will use for doing such a comparison is *competitive analysis*.

Remark We do not assume that the optimal offline algorithm has to be computationally efficient. Under the competitive analysis framework, only the *quality of the best possible solution* matters.

Definition 14.1 (α-competitive online algorithm)**.** *Let σ be an input sequence, c be a cost function, $\mathcal{A}$ be the online algorithm and OPT be the optimal offline algorithm. Then, denote $c_{\mathcal{A}}(\sigma)$ as the cost incurred by $\mathcal{A}$ on σ and $c_{OPT}(\sigma)$ as the cost incurred by OPT on the same sequence. We say that an online algorithm is α-competitive if for any input sequence σ, $c_{\mathcal{A}}(\sigma) \leq \alpha \cdot c_{OPT}(\sigma)$.*

Definition 14.2 (Ski rental problem)**.** *Suppose we wish to ski every day but we do not have any skiing equipment initially. On each day, we can choose between:*

- *Rent the equipment for a day at CHF* 1
- *Buying the equipment (once and for all) for CHF B*

In the toy setting where we may break our leg on each day (and cannot ski thereafter), let d be the (unknown) total number of days we ski. What is the best online strategy for renting/buying?

Claim 14.3. $\mathcal{A}$ = *"Rent for B days, then buy on day B+1" is a 2-competitive algorithm.*

Proof. If $d \leq B$, the optimal offline strategy is to rent everyday, incurring a cost of $c_{OPT}(d) = d$. $\mathcal{A}$ will rent for d days and also incur a loss of $c_A(d) = d = c_{OPT}(d)$. If $d > B$, the optimal offline strategy is to buy the equipment immediately, incurring a loss of $c_{OPT}(d) = B$. $\mathcal{A}$ will rent for B days and then buy the equipment for CHF B, incurring a cost of $c_A(d) = 2B \leq 2c_{OPT}(d)$. Thus, for any d, $c_{\mathcal{A}}(d) \leq 2 \cdot c_{OPT}(d)$. □

Chapter 15

Linear search

Definition 15.1 (Linear search problem)**.** *We have a stack of n papers on the desk. Given a query, we do a linear search from the top of the stack. Suppose the i-th paper in the stack is queried. Since we have to go through i papers to reach the queried paper, we incur a cost of i doing so. We have the option to perform two types of swaps in order to change the stack:*

Free swap *Move the queried paper from position i to the top of the stack for 0 cost.*

Paid swap *For any consecutive pair of items (a, b) before i, swap their relative order to (b, a) for 1 cost.*

What is the best online strategy for manipulating the stack to minimize total cost on a sequence of queries?

Remark One can reason that the free swap costs 0 because we already incurred a cost of i to reach the queried paper.

15.1 Amortized analysis

Amortized analysis[1] is a way to analyze the complexity of an algorithm on a sequence of operations. Instead of looking the worst case performance on a single operation, it measures the total cost for a *batch* of operations.

The dynamic resizing process of hash tables is a classical example of amortized analysis. An insertion or deletion operation will typically cost

[1] See https://en.wikipedia.org/wiki/Amortized_analysis

$\mathcal{O}(1)$ unless the hash table is almost full or almost empty, in which case we double or halve the hash table of size m, incurring a runtime of $\mathcal{O}(m)$.

Worst case analysis tells us that dynamic resizing will incur $\mathcal{O}(m)$ run time per operation. However, resizing only occurs after $\mathcal{O}(m)$ insertion/deletion operations, each costing $\mathcal{O}(1)$. Amortized analysis allows us to conclude that this dynamic resizing runs in amortized $\mathcal{O}(1)$ time. There are two equivalent ways to see it:

- Split the $\mathcal{O}(m)$ resizing overhead and "charge" $\mathcal{O}(1)$ to each of the earlier $\mathcal{O}(m)$ operations.
- The *total* run time for every sequential chunk of m operations is $\mathcal{O}(m)$. Hence, each step takes $\mathcal{O}(m)/m = \mathcal{O}(1)$ amortized run time.

15.2 Move-to-Front

Move-to-Front (MTF) [ST85] is an online algorithm for the linear search problem where we move the queried item to the top of the stack (and do no other swaps). We will show that MTF is a 2-competitive algorithm for linear search. Before we analyze MTF, let us first define a potential function Φ and look at examples to gain some intuition.

Let Φ_t be the number of pairs of papers (i, j) that are ordered differently in MTF's stack and OPT's stack at time step t. By definition, $\Phi_t \geq 0$ for any t. We also know that $\Phi_0 = 0$ since MTF and OPT operate on the same initial stack sequence.

Example One way to interpret Φ is to count the number of inversions between MTF's stack and OPT's stack. Suppose we have the following stacks (visualized horizontally) with $n = 6$:

	1	2	3	4	5	6
MTF's stack	a	b	c	d	e	f
OPT's stack	a	b	e	d	c	f

We have the inversions (c, d), (c, e) and (d, e), so $\Phi = 3$.

Scenario 1 We swap (b, e) in OPT's stack — A new inversion (b, e) was created due to the swap.

	1	2	3	4	5	6
MTF's stack	a	b	c	d	e	f
OPT's stack	a	e	b	d	c	f

Now, we have the inversions (b, e), (c, d), (c, e) and (d, e), so $\Phi = 4$.

Scenario 2 We swap (e, d) in OPT's stack — The inversion (d, e) was destroyed due to the swap.

	1	2	3	4	5	6
MTF's stack	a	b	c	d	e	f
OPT's stack	a	b	d	e	c	f

Now, we have the inversions (c, d) and (c, e), so $\Phi = 2$.

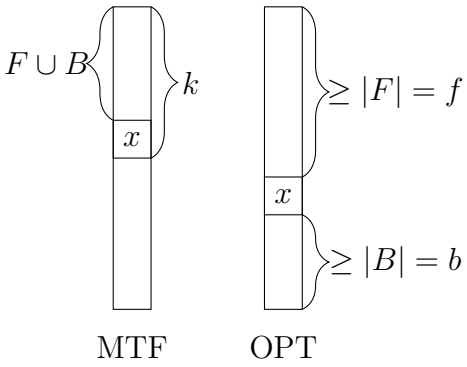
In either case, we see that any paid swap results in ± 1 inversions, which changes Φ by ± 1.

Claim 15.2. *MTF is 2-competitive.*

Proof. We will consider the potential function Φ as before and perform amortized analysis on any given input sequence σ. Let $a_t = c_{MTF}(t) + (\Phi_t - \Phi_{t-1})$ be the amortized cost of MTF at time step t, where $c_{MTF}(t)$ is the cost MTF incurs at time t. Suppose the queried item x at time step t is at position k in MTF's stack. Denote:

$F = \{\text{Items } \textit{on top} \text{ of } x \text{ in MTF's stack and } \textit{on top} \text{ of } x \text{ in OPT's stack}\}$
$B = \{\text{Items } \textit{on top} \text{ of } x \text{ in MTF's stack and } \textit{underneath } x \text{ in OPT's stack}\}$

Let $|F| = f$ and $|B| = b$. There are $k-1$ items in front x, so $f+b = k-1$.



Since x is the k-th item, MTF will incur $c_{MTF}(t) = k = f + b + 1$ to reach item x, then move it to the top. On the other hand, OPT needs to

spend at least $f + 1$ to reach x. Suppose OPT does p paid swaps, then $c_{OPT}(t) \geq f + 1 + p$.

To measure the change in potential, we first look at the swaps done by MTF and how OPT's swaps can affect them. Let $\Delta_{MTF}(\Phi_t)$ be the change in Φ due to MTF and $\Delta_{OPT}(\Phi_t)$ be the change in Φ_t due to OPT. Thus, $\Delta(\Phi_t) = \Delta_{MTF}(\Phi_t) + \Delta_{OPT}(\Phi_t)$. In MTF, moving x to the top destroys b inversions and creates f inversions, so the change in Φ due to MTF is $\Delta_{MTF}(\Phi_t) = f - b$. If OPT chooses to do a free swap, Φ does not increase as both stacks now have x before any element in F. For every paid swap that OPT performs, Φ changes by one since inversions only locally affect the swapped pair and thus, $\Delta_{OPT}(\Phi_t) \leq p$.

Therefore, the effect on Φ from both processes is: $\Delta(\Phi_t) = \Delta_{MTF}(\Phi_t) + \Delta_{OPT}(\Phi_t) \leq (f - b) + p$. Putting together, we have $c_{OPT}(t) \geq f + 1 + p$ and $a_t = c_{MTF}(t) + (\Phi_t - \Phi_{t-1}) = k + \Delta(\Phi_t) \leq 2f + 1 + p \leq 2 \cdot c_{OPT}(t)$. Summing up over all queries in the sequence yields:

$$2 \cdot c_{OPT}(\sigma) = \sum_{t=1}^{|\sigma|} 2 \cdot c_{OPT}(t) \geq \sum_{t=1}^{|\sigma|} a_t$$

With $a_t = c_{MTF}(t) + (\Phi_t - \Phi_{t-1})$ and using the fact that the sum over the change in potential is telescoping, we get:

$$\begin{aligned} \sum_{t=1}^{|\sigma|} a_t &= \sum_{t=1}^{|\sigma|} c_{MTF}(t) + (\Phi_t - \Phi_{t-1}) \\ &= \sum_{t=1}^{|\sigma|} c_{MTF}(t) + (\Phi_{|\sigma|} - \Phi_0) \end{aligned}$$

Since $\Phi_t \geq 0 = \Phi_0$ and $c_{MTF}(\sigma) = \sum_{t=1}^{|\sigma|} c_{MTF}(t)$:

$$\sum_{t=1}^{|\sigma|} c_{MTF}(t) + (\Phi_{|\sigma|} - \Phi_0) \geq \sum_{t=1}^{|\sigma|} c_{MTF}(t) = c_{MTF}(\sigma)$$

We have shown that $c_{MTF}(\sigma) \leq 2 \cdot c_{OPT}(\sigma)$ which completes the proof. □

Chapter 16

Paging

Definition 16.1 (Paging problem [ST85])**.** *Suppose we have a fast memory (cache) that can fit k pages and an unbounded sized slow memory. Accessing items in the cache costs 0 units of time while accessing items in the slow memory costs 1 unit of time. After accessing an item in the slow memory, we can bring it into the cache by evicting an incumbent item if the cache was full. What is the best online strategy for maintaining items in the cache to minimize the total access cost on a sequence of queries?*

Denote *cache miss* as accessing an item that is not in the cache. Any sensible strategy should aim to reduce the number of cache misses. For example, if $k = 3$ and $\sigma = \{1, 2, 3, 4, \ldots, 2, 3, 4\}$, keeping item 1 in the cache will incur several cache misses. Instead, the strategy should aim to keep items $\{2, 3, 4\}$ in the cache. We formalize this notion in the following definition of *conservative strategy*.

Definition 16.2 (Conservative strategy)**.** *A strategy is conservative if on any consecutive subsequence that includes only k distinct pages, there are at most k cache misses.*

Remark Some natural paging strategies such as "Least Recently Used (LRU)" and "First In First Out (FIFO)" are conservative.

Claim 16.3. *If $\mathcal{A}$ is a deterministic online algorithm that is α-competitive, then $\alpha \geq k$.*

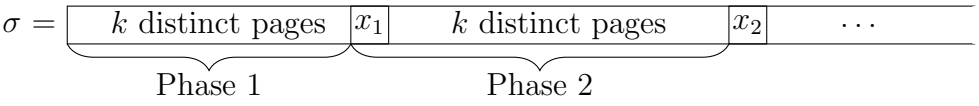
Proof. Consider the following input sequence σ on $k + 1$ pages: since the cache has size k, at least one item is not in the cache at any point in time. Iteratively pick $\sigma(t + 1)$ as the item not in the cache after time step t.

Since $\mathcal{A}$ is deterministic, the adversary can simulate $\mathcal{A}$ for $|\sigma|$ steps and build σ accordingly. By construction, $c_{\mathcal{A}}(\sigma) = |\sigma|$.

On the other hand, since OPT can see the entire sequence σ, OPT can choose to evict the page i that is requested furthest in the future. The next request for page i has to be at least k requests ahead in the future, since by definition of i all other pages $j \neq i \in \{1, ..., k+1\}$ have to be requested before i. Thus, in every k steps, OPT has ≤ 1 cache miss. Therefore, $c_{OPT} \leq \frac{|\sigma|}{k}$ which implies: $k \cdot c_{OPT} \leq |\sigma| = c_{\mathcal{A}}(\sigma)$. □

Claim 16.4. *Any conservative online algorithm $\mathcal{A}$ is k-competitive.*

Proof. For any given input sequence σ, partition σ into m *maximal phases* — $P_1, P_2, \ldots, P_m$ — where each phase has k distinct pages, and a new phase is created only if the next element is different from the ones in the current phase. Let x_i be the first item that does not belong in Phase i.



By construction, OPT has to pay ≥ 1 to handle the elements in $P_i \cup \{x_i\}$, for any i; so $c_{OPT} \geq m$. On the other hand, since $\mathcal{A}$ is conservative, $\mathcal{A}$ has $\leq k$ cache misses per phase. Hence, $c_{\mathcal{A}}(\sigma) \leq k \cdot m \leq k \cdot c_{OPT}(\sigma)$. □

Remark A randomized algorithm can achieve $\mathcal{O}(\log k)$-competitiveness. This will be covered in the next lecture.

16.1 Types of adversaries

Since online algorithms are analyzed on all possible input sequences, it helps to consider adversarial inputs that may induce the worst case performance for a given online algorithm $\mathcal{A}$. To this end, one may wish to classify the classes of adversaries designing the input sequences (in increasing power):

Oblivious The adversary designs the input sequence σ at the beginning. It does not know any randomness used by algorithm $\mathcal{A}$.

Adaptive At each time step t, the adversary knows all randomness used by algorithm $\mathcal{A}$ thus far. In particular, it knows the exact state of the algorithm. With these in mind, it then picks the $(t+1)$-th element in the input sequence.

Fully adaptive The adversary knows all possible randomness that will be used by the algorithm $\mathcal{A}$ when running on the full input sequence σ. For

instance, assume the adversary has access to the same pseudorandom number generator used by $\mathcal{A}$ and can invoke it arbitrarily many times while designing the adversarial input sequence σ.

Remark If $\mathcal{A}$ is deterministic, then all three classes of adversaries have the same power.

16.2 Random Marking Algorithm (RMA)

Consider the Random Marking Algorithm (RMA), a $\mathcal{O}(\log k)$-competitive algorithm for paging against oblivious adversaries:

- Initialize all pages as marked
- Upon request of a page p
  - If p is not in cache,
    - If all pages in cache are marked, unmark all
    - Evict a random unmarked page
  - Mark page p

Example Suppose $k = 3$, $\sigma = (2, 5, 2, 1, 3)$.

Suppose the cache is initially:

Cache	1	3	4
Marked?	✓	✓	✓

When $\sigma(1) = 2$ arrives, all pages were unmarked. Suppose the random eviction chose page '3'. The newly added page '2' is then marked.

Cache	1	2	4
Marked?	✗	✓	✗

When $\sigma(2) = 5$ arrives, suppose random eviction chose page '4' (between pages '1' and '4'). The newly added page '5' is then marked.

Cache	1	2	5
Marked?	✗	✓	✓

When $\sigma(3) = 2$ arrives, page '2' in the cache is marked (no change).

Cache	1	2	5
Marked?	✗	✓	✓

When $\sigma(4) = 1$ arrives, page '1' in the cache is marked. At this point, any page request that is not from $\{1, 2, 5\}$ will cause a full unmarking of all pages in the cache.

Cache	1	2	5
Marked?	✓	✓	✓

When $\sigma(5) = 3$ arrives, all pages were unmarked. Suppose the random eviction chose page '5'. The newly added page '3' is then marked.

Cache	1	2	3
Marked?	✗	✗	✓

We denote a phase as the time period between 2 consecutive full unmarking steps. That is, each phase is a maximal run where we access k distinct pages. In the above example, $\{2, 5, 2, 1\}$ is such a phase for $k = 3$.

Observation As pages are only unmarked at the beginning of a new phase, the number of unmarked pages is monotonically decreasing within a phase.

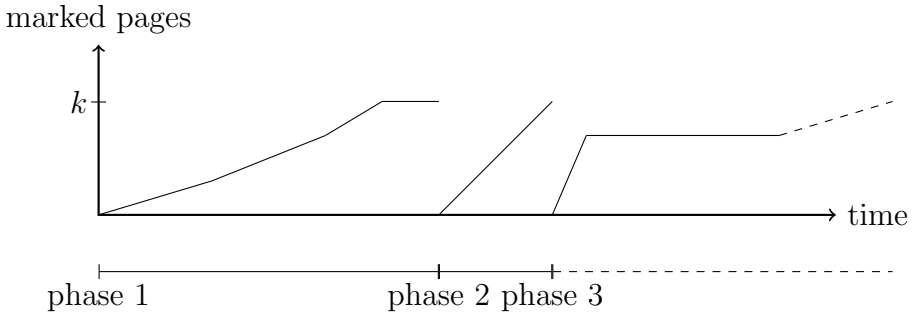


Figure 16.1: The number of marked pages within a phase is monotonically increasing.

Theorem 16.5. *RMA is $\mathcal{O}(\log k)$-competitive against any oblivious adversary.*

Proof. Let P_i be the set of pages at the *start* of phase i. Since requesting a marked page does not incur any cost, it suffices to analyze the *first time* any request occurs within the phase.

Let m_i be the number of *unique new requests* (pages that are not in P_i) and o_i as the number of *unique old requests* (pages that are in P_i). By definition, $o_i \leq k$ and $m_i + o_i = k$.

We have $c_{RMA}(\text{Phase } i) = (\text{Cost due to new requests}) + (\text{Cost due to old requests})$. We first focus on the extra cost incurred from the old requests,

that is when an old page is requested that has already been kicked out upon the arrival of a new request.

Order the old requests in the order which they appear in the phase and let x_j be the j^{th} old request, for $j \in \{1, \ldots, o_i\}$. Define l_j as the number of *distinct new* requests before x_j.

For $j \in \{1, \ldots, o_i\}$, consider the first time the j^{th} old request x_j occurs. Since the adversary is oblivious, x_j is equally likely to be in any position in the cache at the start of the phase. After seeing $(j-1)$ old requests and marking their cache positions, there are $k-(j-1)$ initial positions in the cache that x_j could be in. Since we have only seen l_j new requests and $(j-1)$ old requests, there are at least[1] $k - l_j - (j-1)$ old pages remaining in the cache. So, the probability that x_j is in the cache when requested is at least $\frac{k-l_j-(j-1)}{k-(j-1)}$. Then,

$$\begin{aligned}
\text{Cost due to old requests} &= \sum_{j=1}^{o_i} \Pr[x_j \text{ is } \textit{not} \text{ in cache when requested}] && \text{Sum over old requests} \\
&\leq \sum_{j=1}^{o_i} \frac{l_j}{k-(j-1)} && \text{From above} \\
&\leq \sum_{j=1}^{o_i} \frac{m_i}{k-(j-1)} && \text{Since } l_j \leq m_i = |\mathcal{N}| \\
&\leq m_i \cdot \sum_{j=1}^{k} \frac{1}{k-(j-1)} && \text{Since } o_i \leq k \\
&= m_i \cdot \sum_{j=1}^{k} \frac{1}{j} && \text{Rewriting} \\
&= m_i \cdot H_k && \text{Since } \sum_{i=1}^{n} \frac{1}{i} = H_n
\end{aligned}$$

Since every new request incurs a unit cost, the cost due to these requests is m_i.

Together for new and old requests, we get $c_{RMA}(\text{Phase } i) \leq m_i + m_i \cdot H_k$.

We now analyze OPT's performance. By definition of phases, among all requests between two consecutive phases (say, $i-1$ and i), a total of $k + m_i$ distinct pages are requested. So, OPT has to incur at least $\geq m_i$ to bring in

[1] We get an equality if *all* these requests kicked out an old page.

these new pages. To avoid double counting, we lower bound $c_{OPT}(\sigma)$ for both odd and even i: $c_{OPT}(\sigma) \geq \sum_{\text{odd } i} m_i$ and $c_{OPT}(\sigma) \geq \sum_{\text{even } i} m_i$. Together,

$$2 \cdot c_{OPT}(\sigma) \geq \sum_{\text{odd } i} m_i + \sum_{\text{even } i} m_i \geq \sum_i m_i$$

Therefore, we have:

$$c_{RMA}(\sigma) \leq \sum_i (m_i + m_i \cdot H_k) = \mathcal{O}(\log k) \sum_i m_i \leq \mathcal{O}(\log k) \cdot c_{OPT}(\sigma)$$

□

Remark In the above example, $k = 3$, phase $1 = (2, 5, 2, 1)$, $P_1 = \{1, 3, 4\}$, new requests $= \{2, 5\}$, old requests $= \{1\}$. Although '2' appeared twice, we only care about analyzing the first time it appeared.

16.3 Lower Bound for Paging via Yao's Principle

Yao's Principle Often, it is considerably easier to obtain (distributional) lower bounds against deterministic algorithms, than to (directly) obtain deterministic lower bound instances against randomized algorithms. We use Yao's principle to bridge this gap. Informally, this principle tells us that if *no* deterministic algorithm can do well on a given distribution of random inputs (D), then for any randomized algorithm, there is a deterministic bad input so that the cost of the randomized algorithm on this particular input will be high (C). We next state and prove Yao's principle.

Before getting to the principle, let us observe that given the sequence of random bits used, a randomized algorithm behaves deterministically. Hence, one may view a randomized algorithm as a random choice from a distribution of deterministic algorithms.

Let $\mathcal{X}$ be the space of problem inputs and $\mathcal{A}$ be the space of all possible deterministic algorithms. Denote probability distributions over $\mathcal{A}$ and $\mathcal{X}$ by $p_a = \Pr[A = a]$ and $q_x = \Pr[X = x]$, where X and A are random variables for input and deterministic algorithm, respectively. Define $c(a, x)$ as the cost of algorithm $a \in A$ on input $x \in X$.

Theorem 16.6 ([Yao77])**.**

$$C = \max_{x \in X} \mathbb{E}_p[c(a, x)] \geq \min_{a \in A} \mathbb{E}_q[c(a, x)] = D$$

Proof.

$$
\begin{aligned}
C &= \sum_x q_x \cdot C && \text{Sum over all possible inputs } x \\
&\geq \sum_x q_x \mathbb{E}_p[c(A,x)] && \text{Since } C = \max_{x \in X} \mathbb{E}_p[c(A,x)] \\
&= \sum_x q_x \sum_p p_a c(a,x) && \text{Definition of } \mathbb{E}_p[c(A,x)] \\
&= \sum_a p_a \sum_q q_x c(a,x) && \text{Swap summations} \\
&= \sum_a p_a \mathbb{E}_q[c(a,X)] && \text{Definition of } \mathbb{E}_q[c(a,X)] \\
&\geq \sum_a p_a \cdot D && \text{Since } D = \min_{a \in A} \mathbb{E}_q[c(a,X)] \\
&= D && \text{Sum over all possible algorithms } a
\end{aligned}
$$

□

Application to the paging problem

Theorem 16.7. *Any (randomized) algorithm has competitive ratio* $\Omega(\log k)$ *against an oblivious adversary.*

Proof. Fix an arbitrary deterministic algorithm $\mathcal{A}$. Let $n = k+1$ and $|\sigma| = m$. Consider the following random input sequence σ where the i-th page is drawn from $\{1, \ldots, k+1\}$ uniformly at random.

By construction of σ, the probability of having a cache miss is $\frac{1}{k+1}$ for $\mathcal{A}$, regardless of what $\mathcal{A}$ does. Hence, $\mathbb{E}[c_{\mathcal{A}}(\sigma)] = \frac{m}{k+1}$.

On the other hand, an optimal offline algorithm may choose to evict the page that is requested furthest in the future. As before, we denote a phase as a maximal run where there are k distinct page requests. This means that $\mathbb{E}[c_{OPT}(\sigma)] = \text{Expected number of phases} = \frac{m}{\text{Expected phase length}}$.

To analyze the expected length of a phase, suppose there are i distinct pages so far, for $0 \leq i \leq k$. The probability of the next request being new is $\frac{k+1-i}{k+1}$, and one expects to get $\frac{k+1}{k+1-i}$ requests before having $i+1$ distinct pages. Thus, the expected length of a phase is $\sum_{i=0}^{k} \frac{k+1}{k+1-i} = (k+1) \cdot H_{k+1}$. Therefore, $\mathbb{E}[c_{OPT}(\sigma)] = \frac{m}{(k+1) \cdot H_{k+1}}$.

So far we have obtained that $D = \mathbb{E}[c_{\mathcal{A}}(\sigma)] = \frac{m}{k+1}$; from Yao's Minimax Principle we know that $C \geq D$, hence we can also compare the competitive ratios $\frac{C}{\mathbb{E}[c_{OPT}(\sigma)]} \geq \frac{D}{\mathbb{E}[c_{OPT}(\sigma)]} = H_{k+1} = \Theta(\log k)$. □

Remark The length of a phase is essentially the coupon collector problem with $n = k + 1$ coupons.