CS **Summer**

Online Algorithms

Authors: Shaleen Baral

Contents

1. Introduction

1.1. Approximation Algorithms

Definition 1.1.1 (Optimization Problem): An *optimization problem* Π is a 5-tuple $(\mathcal{I}, \mathcal{O}, s, q, g)$:

a. $\mathcal{I}$: the set of *instances*.
b. $\mathcal{O}$: the set of *solutions*.
c. $s : \mathcal{I} \to \mathcal{P}(\mathcal{O})$: for every instance $I \in \mathcal{I}$, $s(I) \subseteq \mathcal{O}$ denotes the set of feasible solutions for I.
d. $q : \mathcal{I} \times \mathcal{O} \to \mathbb{R}$: for every instance $I \in \mathcal{I}$ and every feasible solution $O \in s(I)$, $q(I, O)$ denotes the measure of I and O.
e. $g \in \{\max, \min\}$.

An *optimal solution* for an instance $I \in \mathcal{I}$ of Π is a solution $\texttt{OPT}(I) \in s(I)$ such that

$$q(I, \texttt{OPT}(I)) = g\{q(I, O) \mid O \in s(I)\}.$$

If $g = \min$, we call Π a *minimization problem* and refer to q as the *cost*. If $g = \max$ we call Π a *maximization problem* and refer to q as the *gain*.

Definition 1.1.2 (Consistent): An algorithm is *consistent* for an (optimization) problem Π if it computes a feasible solution for every given instance.

Definition 1.1.3 (Approximation Algorithms): Let Π be an optimization problem, and let $\texttt{ALG}$ be a consistent algorithm for Π. For $r \geq 1$, $\texttt{ALG}$ is an *r-approximation algorithm* for Π if, for every $I \in \mathcal{I}$

$$q(\texttt{OPT}(I)) \leq r \cdot q(\texttt{ALG}(I))$$

if Π is a maximization problem, or

$$q(\texttt{ALG}(I)) \leq r \cdot q(\texttt{ALG}(I))$$

if Π is a minimization problem.

The *approximation ratio* of $\texttt{ALG}$ is defined as

$$r_{\texttt{ALG}} = \inf\{r \geq 1 \mid \texttt{ALG} \text{ is an } r\text{-approximation algorithm for } \Pi\}.$$

Remark: Sometimes we define r-approximation so that $r \in (0, 1]$.

Definition 1.1.4 (Simple Knapsack Problem): The *simple knapsack problem* is a maximization problem specified as follows:

An instance I is given by a sequence of $n + 1$ numbers– $w_1, w_2, ..., w_n, W$. A feasible set is any set $O \subseteq [n]$ such that

$$\sum_{i \in O} w_i \leq W.$$

The gain of a solution O and a corresponding instance I is given by

$$q(I, O) = \sum_{i \in O} w_i.$$

The goal is to maximize this gain.

Remark: Think of the numbers w_i as being weights of items and W as being the maximum weight capacity of the knapsack.

Algorithm 1: KN-GREEDY

$O = \emptyset$
$s = 0$
sort such that $w_1 \geq w_2 \geq \cdots \geq w_n$.
while $i < n$ and $s + w_{i+1} \leq W$ **do**
  $O = O \cup w_{i+1}$
  $s = s + w_{i+1}$
  $i = i + 1$
end
return O

Proposition 1.1.1: KN-Greedy is a polynomial-time 2-approximation algorithm for the simple knapsack problem.

Proof: Running time is $\mathcal{O}(n)$, which is polynomial. The loop invariants ensure that the algorithm is consistent.

Now, we show that this is a 2-approximation algorithm. Consider the labeling of the weights after sorting i.e. assume $w_1 \geq w_2 \geq \cdots \geq w_n$. If the algorithm accepts all the items, then this is clearly the optimal solution too, $\texttt{OPT} = \texttt{ALG}$.

Suppose that the algorithm rejects the $(r+1)$-th item. If $\sum_{i=1}^{r} w_i \geq \frac{W}{2}$ then we are done as

$$\texttt{ALG} = \sum_{i=1}^{r} w_i \geq \frac{W}{2} \geq \frac{\texttt{OPT}}{2}.$$

Next, if $\sum_{i=1}^{r} w_i < \frac{W}{2}$ then note that $w_{r+1} \leq w_r < \frac{W}{2}$. Since the $(r+1)$-th item is rejected,

$$\texttt{ALG} = \sum_{i=1}^{r} w_i > W - w_{r+1} > \frac{W}{2} \geq \frac{\texttt{OPT}}{2}.$$

To see that this bound is tight with $r = 2$, we construct a family of instances whose approximation ratio approaches 2. Consider the instance with $n = 3$, with weights

$$\frac{W}{2} + 1, \frac{W}{2}, \frac{W}{2}$$

and maximum weight capacity W. Then, the algorithm always accepts the item with weight $\frac{W}{2} + 1$ and cannot accept any more whereas the optimum solution would accept the two items with weight $\frac{W}{2}$. The approximation ratio is then,

$$\frac{\texttt{OPT}}{\texttt{ALG}} = \frac{W}{\frac{W}{2} + 1} = 2 \cdot \frac{W}{W + 1}$$

As $W \to \infty$, $\texttt{OPT}/\texttt{ALG} \longrightarrow 2$.

□

Definition 1.1.5 (FPTAS): A *fully polynomial-time approximation scheme* is an algorithm that takes as input an $\varepsilon > 0$ and an instance of a (optimization) problem and returns an output value that is at least $(1-\varepsilon)$ $\texttt{OPT}$ and at most $(1+\varepsilon)$ $\texttt{OPT}$. Importantly, the running time of the algorithm must be polynomial in both the size of the input and $\frac{1}{\varepsilon}$.

1.2. Online Algorithms

Definition 1.2.1 (Online Problem): An *online problem* Π is a 5-tuple $(\mathcal{I}, \mathcal{O}, s, q, g)$:

a. $\mathcal{I}$: the set of *instances*. Every instance $I \in \mathcal{I}$ is a sequence of *requests* $I = (x_1, x_2, ..., x_n)$ with $n \in \mathbb{N}$.
b. $\mathcal{O}$: the set of *solutions*. Every output $O \in \mathcal{O}$ is a sequence of *answers* $O = (y_1, y_2, ..., y_n)$ with $n \in \mathbb{N}$.
c. $s : \mathcal{I} \to \mathcal{P}(\mathcal{O})$: for every instance $I \in \mathcal{I}$, $s(I) \subseteq \mathcal{O}$ denotes the set of *feasible solutions* for I.
d. $q : \mathcal{I} \times \mathcal{O} \to \mathbb{R}$: for every instance $I \in \mathcal{I}$ and every feasible solution $O \in s(I)$, $q(I, O)$ denotes the measure of I and O.
e. $g \in \{\max, \min\}$.

An *optimal solution* for an instance $I \in \mathcal{I}$ of Π is a solution $\texttt{OPT}(I) \in s(I)$ such that

$$q(I, \texttt{OPT}(I)) = g\{q(I, O) \mid O \in s(I)\}.$$

If $g = \min$, we call Π a *online minimization problem* and refer to q as the *cost*. If $g = \max$ we call Π a *online maximization problem* and refer to q as the *gain*.

Definition 1.2.2 (Online Algorithm): Let Π be an online problem and let $I = (x_1, x_2, ..., x_n)$ be an instance of Π. An *online algorithm* $\texttt{ALG}$ for Π computes the output $\texttt{ALG}(I) = (y_1, y_2, ..., y_n)$, where y_i only depends on $x_1, x_2, ..., x_i$ and $y_1, y_2, ..., y_{i-1}$; we also require $\texttt{ALG}(I) \in s(I)$, that is, $\texttt{ALG}(I)$ is a fesible solution for I.

Definition 1.2.3 (Competitive Ratio): Let Π be an online problem, and let $\texttt{ALG}$ be a consistent online algorithm for Π. For $c \geq 1$, $\texttt{ALG}$ is *c-competitive* for Π if there is a constant $\alpha \geq 0$ such that, for every instance $I \in \mathcal{I}$.

$$q(\texttt{OPT}(I)) \leq c \cdot q(\texttt{ALG}(I)) + \alpha$$

if Π is an online maximization problem, or

$$q(\texttt{ALG}(I)) \leq c \cdot q(\texttt{OPT}(I)) + \alpha$$

if Π is an online mimization problem. If these inequalities holds with $\alpha = 0$ we call $\texttt{ALG}$ *strictly c-competitive*. $\texttt{ALG}$ is *optimal* if it is strictly 1-competitive.

The *competitive ratio* is defined as

$$c_{\texttt{ALG}} = \inf\{c \geq 1 \mid \texttt{ALG} \text{ is } c\text{-competitive for } \Pi\}.$$

If the competitive ratio of $\texttt{ALG}$ is constant and the best that is achievable by any online algorithm for Π, we call $\texttt{ALG}$ *strongly c-competitive*.

Remark: Commonly, we refer to $\texttt{OPT}(I)$ as the *optimal offline solution*. It may not even be achievable online as it may require knowledge of the complete instance!

Remark: If the competitive ratio of an online algorithm $\texttt{ALG}$ is at most c, where c is a constant, we call $\texttt{ALG}$ *competitive* or if $\alpha = 0$, *strictly competitive*. If the algorithm does not possess a competitive ratio with an upper bound that is idenpednent of the input length, we call it *not competitive*. It is fine to call an online algorithm

competitive if its competitive ratio depends on some parameter of the problem being studied. This classification isn't always clear cut but is still often helpful.

Why do we have the additive constant α when defining the competitive ratio? The additive constant allows us to ignore finitely many exceptional instances on which the online algorithm may perform poorly. Particularly, to prove lower bounds, the constant α forces us to construct infinitely many instances with increasing costs or gains. The following lemmas and propositions make this notion more explicit.

Lemma 1.2.1: Let Π be an online minimization problem and let $\mathcal{I} = \{I_1, I_2, ...\}$ be an infinite set of instances of Π such that $|I_i| \leq |I_{i+1}|$ and such that the number of different input lengths in $\mathcal{I}$ is infinite. Suppose further that $q(\texttt{OPT}) > 0$ on $\mathcal{I}$. If there is an increasing, unbounded function $c : \mathbb{N}^+ \to \mathbb{R}^+$ such that,

$$\frac{q(\texttt{ALG}(I_i))}{q(\texttt{OPT}(I_i))} \geq c(n) \text{ where } n = |I_i|,$$

then ALG isn't competitive.

Proof: Suppose for contradiction that ALG is competitive, that is

$$q(\texttt{ALG}(I_i)) \leq c' \cdot q(\texttt{OPT}(I_i)) + \alpha.$$

Then,

$$[c(n) - c'] \cdot q(\texttt{OPT}(I_i)) \leq \alpha.$$

This clearly contradicts the fact that α is a constant.

□

Lemma 1.2.2: Let Π be an online minimization problem and let $\mathcal{I} = \{I_1, I_2, ...\}$ be an infinite set of instances of Π such that $|I_i| \leq |I_{i+1}|$ and such that the number of different input lengths in $\mathcal{I}$ is infinite. Suppose further that $q(\texttt{ALG}) > 0$ on $\mathcal{I}$. If there is an increasing, unbounded function $c : \mathbb{N}^+ \to \mathbb{R}^+$ such that,

$$\frac{q(\texttt{OPT}(I_i))}{q(\texttt{ALG}(I_i))} \geq c(n) \text{ where } n = |I_i|,$$

then ALG isn't competitive.

Remark: Since this is a minimization problem, $q(\texttt{OPT}) > 0$ also implies $q(\texttt{ALG}) > 0$.

Proof: Same idea as the prior proof.

□

Proposition 1.2.1: Let Π be an online minimization problem, and let $\mathcal{I} = \{I_1, I_2, ...\}$ be an infinite set of instances of Π such that $|I_i| \leq |I_{i+1}|$, and such that the number of different input lengths in $\mathcal{I}$ is infinite. Let ALG be an online algorithm for Π. If there is some constant $c \geq 1$ such that

a. $\dfrac{q(\texttt{ALG}(I_i))}{q(\texttt{OPT}(I_i))} \geq c, \text{ for every } i \in \mathbb{N}^+$

b. $\lim_{i \to \infty} q(\texttt{OPT}(I_i)) = \infty$

then ALG isn't a $(c - \varepsilon)$-competitive online algorithm for Π, for any $\varepsilon > 0$.

Proof: Suppose for contradiction that ALG is a $(c-\varepsilon)$-competitive algorithm for some $\varepsilon > 0$. Then, there is a constant α such that

$$q(\texttt{ALG}(I_i)) \leq (c-\varepsilon) \cdot q(\texttt{OPT}(I_i)) + \alpha$$
$$\implies \frac{q(\texttt{ALG}(I_i))}{q(\texttt{OPT}(I_i))} - \frac{\alpha}{q(\texttt{OPT}(I_i))} \leq c - \varepsilon$$

for every $i \in \mathbb{N}^+$. By a), the first term above is at least c and by b), we can find instances for which the second term is less than ε. Thus, we have a contradiction.

□

Proposition 1.2.2: Let Π be an online maximization problem, and let $\mathcal{I} = \{I_1, I_2, ...\}$ be an infinite set of instances of Π such that $|I_i| \leq |I_{i+1}|$, and such that the number of different input lengths in $\mathcal{I}$ is infinite. Let ALG be an online algorithm for Π. If there is some constant $c \geq 1$ such that

a. $\dfrac{q(\texttt{OPT}(I_i))}{q(\texttt{ALG}(I_i))} \geq c, \text{ for every } i \in \mathbb{N}^+$

b. $\lim_{i \longrightarrow \infty} q(\texttt{OPT}(I_i)) = \infty$

then ALG isn't a $(c-\varepsilon)$-competitive online algorithm for Π, for any $\varepsilon > 0$.

Proof: Suppose for contradiction that ALG is a $(c-\varepsilon)$-competitive online algorithm for some $\varepsilon > 0$. Then, there is some constant α such that

$$\frac{q(\texttt{OPT}(I_i))}{q(\texttt{ALG}(I_i))} - \frac{\alpha}{q(\texttt{ALG}(I_i))} \leq c - \varepsilon$$

for every $i \in \mathbb{N}^+$. Furthermore, by a) and b), if $q(\texttt{ALG}(I_i))$ were bounded by a constant, it wouldn't be competitive at all. Thus, it is fair to assume that $\lim_{i \longrightarrow \infty} q(\texttt{ALG}(I_i)) = \infty$. By a), the first term above is at least c and from the previous sentence, we can find instances for which the second term is less than ε. Thus, we have a contradiction.

□

Remark: Propositions 1.2.1 and 1.2.2 need to be carefully interpreted. For example, they don't rule out the possibility for a $\left(c - \frac{1}{n}\right)$-competitive algorithm.

1.3. Paging: Basics

Definition 1.3.1 (Paging): The *paging problem* is an online minimization problem.

Suppose there are $m \in \mathbb{N}^+$ memory pages $p_1, p_2, ..., p_m$, which are stored in the main memory. An instance is a sequence $I = (x_1, x_2, ..., x_n)$ such that $x_i \in \{p_1, p_2, ..., p_m\}$, for all $i \in [n]$. This means that page x_i is requested in time step T_i.

An online algorithm ALG for paging maintains a *cache* memory of size k with $k < m$, represented by the tuple $B_i = \left(p_{j_1}, p_{j_2}, ..., p_{j_k}\right)$ for time step T_i. Initially, the cache is initialized as $B_0 = (p_1, p_2, ..., p_k)$, that is, with the first k pages. If in some time step T_i, a page x_i is requested and $x_i \in B_{i-1}$, ALG outputs $y_i = 0$. Conversely, if $x_i \notin B_{i-1}$, ALG has to choose a page $p_j \in B_{i-1}$, which is then removed from the cache to make room for x_i. In this case, ALG outputs $y_i = p_j$ and the new cache content is $B_i = \left(B_{i-1} \setminus p_j\right) \cup x_i$.

The cost is defined as

$$q(\texttt{ALG}(I)) = |\{i \in [n] \mid y_i \neq 0\}|$$

and the goal is to minimize it.

Definition 1.3.2 (k-Phase Partition): Let $I = (x_1, x_2, ..., x_n)$ be an arbitrary instance of paging. A *k-phase partition of I* assigns the requests from I to consecutive disjoint phases $P_1, P_2, ..., P_N$ such that

a. Phase P_1 starts with the first request for a page that is not initially in the cache. Then, P_1 contains a maximum-length subsequence of I that contains at most k distinct pages.
b. For any i with $2 \leq i \leq N$, phase P_i is a maximum-length subsequence of I that starts right after P_{i-1} and again contains at most k distinct pages.

One possible strategy is given by the following algorithm.

Algorithm 2: `FIFO`

The cache is organized as a queue. Whenever a page must be evicted, the one residing in the cache for the longest time is chosen. The first k pages may be removed arbirarily.

We analyze this algorithm by considering its behavior on different

Lemma 1.3.1: Any optimal solution `OPT` must make at least one page fault for every phase in a k-phase partition.

Proof: Let $I = (x_1, x_2, ..., x_N)$ be any instance of paging and consider I's k-phase partition $P_1, P_2, ..., P_N$. WLOG, assume $x_1 \notin \{p_1, p_2, ..., p_k\}$. Shift the k-phase partition $\{P_i\}_{i \in [N]}$ by a single page to form the partition $\{P_i'\}_{i \in [N]}$. Note that the last phase P_N' may be empty but all others have the same length as before (i.e. $|P_i| = |P_i'|$).

For $i \in [N-1]$, all phases P_i had maximum length (with respect to containing k-distinct pages) and thus, the corresponding phases P_i' contain k pages that differ from the page p' that was last requested before the start of P_i'. Since p' is in the cache of `OPT` at the beginning of P_i' and there are k more distinct requests different from p', `OPT` has to cause one page fault during P_i'. This gives us $N-1$ pages faults for `OPT` plus an additional one on x_1 at the beginning before any of the phases $\{P_i'\}$.

□

Proposition 1.3.1: `FIFO` is strictly k-competitive for paging.

Proof: Let $I = (x_1, x_2, ..., x_N)$ be any instance of paging and consider I's k-phase partition $P_1, P_2, ..., P_N$. WLOG, assume $x_1 \notin \{p_1, p_2, ..., p_k\}$.

We start by showing that in any fixed phase P_i, $i \in [N]$, `FIFO` does not cause more than k page faults during P_i. By definition there are at most k pages requested in this phase. Suppose p is the first page in phase P_i that causes a page fault for `FIFO`. Further suppose that C_i is the set of all pages added to the cache in phase P_i. Then p must be the first element of C_i that is evicted from the cache. However, when p is loaded into the cache there are $k-1$ other pages in the cache that must be removed before p. So, p must remain in the cache for the next $k-1$ page faults. Every element of $C_i - p$ can cause at most one page fault in the next $k-1$ page faults. Since there are at most $k-1$ distinct pages requested in P_i, it must be the case that phase P_i ends in the next $k-1$ page faults too. Thus, no element causes more than one page fault in phase P_i. Consequently, there are at most k page faults in total during any one phase.

Thus, by Lemma 1.3.1,

$$\texttt{ALG} \leq k \cdot N \leq k \cdot \texttt{OPT}.$$

□

In fact, this is the best we can hope for.

Proposition 1.3.2: No online algorithm for paging is better than k-competitive.

Proof: For convenience, take n to be a multiple of k. Consider instances with $k+1$ pages $p_1, p_2, ..., p_k, p_{k+1}$. Recall that the cache is initialized $(p_1, p_2, ..., p_k)$. Since the cache size is k, there is exactly one page, at any given time step, that isn't in the cache of ALG. The idea is that the adversary always requests precisely this uncached page to obtain an instance of length n. Since the adversary knows ALG, it can always forsee which page will be replaces by ALG if a page fault occurs (for example, it can just run ALG to determine this).

More formally, the following algorithm can construct the instance for us.

Algorithm 3: Paging Adversary

$I = (p_{k+1})$
$i = 1$
while $i \leq n - 1$ do
  $p =$ the page that is currently note in the cache of ALG
  $I = I \cup p$
  $i = i + 1$
end
return I

By construction, this causes a page fault at every time step. Hence, ALG has a total cost of n on instance I.

Now, we study the optimal cost OPT for this instance. We divide the input into distinct consecutive phases such that each phase consists of exactly k time steps. Note that ALG makes k page faults in every phase. So, we want to show that OPT makes at most one page fault in every phase.

For any phase P_i, consider the first time step T_j, $j \in [(i-1)k + 1, ik]$, wherein the first page fault occurs. Note that there P_i consists of exactly $ik - j \leq k - 1$ more time steps, so at msot $k - 1$ more distinct pages are requested. Since the cache stores k pages, there must be at least one page p' in the cache that isn't requested during this phase and OPT chooses p' to evict. If there are more than one candidate for p', OPT can break the tie by choosing the page whose first request is the latest among all such pages [TERRIBLE EXPOSITION: should just explicitly say that Longest Forward Distance is the local optima or not leak details of the OPT strategy without meaning to!]. We have, on this instance,

$$\texttt{ALG}(I) = n = k \cdot \frac{n}{k} \leq k \cdot \texttt{OPT}(I).$$

Now if , as $n \to \infty$, the number of page faults caused by OPT is constant or bounded, ALG isn't competitive at all (Lemma 1.2.1). On the other hand, if the number of page faults by OPT increases with n, then ALG cannot be better than k-competitive (Proposition 1.2.2).

□

A natural counterpart to FIFO is the following strategy.

Algorithm 4: LIFO

The cache is organized as a stack. Whenever a page must be evicted, the one that was most recently loaded into the cache is chosen. On the first page fault, an arbirary page may be removed.

This is a terrible strategy! Intuitively, this is because we end up using only one of the cells in the cache.

Proposition 1.3.3: `LIFO` is not competitive for paging.

Proof: We use Lemma 1.2.1 by showing that, for every n, there is an instance of paging of length n such that $\frac{q(\texttt{LIFO})}{q(\texttt{OPT})}$ grows propotionally with n.

Consider an instance of length n on $m = k + 1$ total pages where the adversary always requests the same two pages. Since the cache is initialized with pages $p_1, ..., p_k$, the adversary starts off by requesting page p_{k+1}. As a result, `LIFO` must evict a page p_i from the cache. Since the adversary knows that `LIFO` chooses p_i, it requests it in time step T_2 and `LIFO` removes p_{k+1}. The adversary keeps requesting these two pages, constructing the instance I,

$$(p_{k+1}, p_i, p_{k+1}, p_i, ...)$$

We see that on this instance, `LIFO` causes a page fault at every time step. On the other hand, the optimal solution `OPT` just removes a page p_j with $j \neq i$ in time step T_1 and has overall cost 1 because both p_i and p_{k+1} will be in the cache from this time step onwards.

□

Intuitively, we may expect the frequency of access to be a good heuristic. This gives us the following strategy dubbed *Least Frequently Used*.

Algorithm 5: `LFU`

On a page fault, the page that was, so far, least frequently used is removed.

Unfortunately, this isn't much better in the worst case.

Proposition 1.3.4: `LFU` is not competitive for paging.

Proof: Again, we want to employ Lemma 1.2.1. The idea is very similar to the adversarial construction for `LIFO`. Consider the instance I given by,

$$(\underbrace{p_1, p_1, ..., p_1}_{n' \text{ requests}}, \underbrace{p_2, p_2, ..., p_2}_{n' \text{ requests}}, ..., \underbrace{p_{k-1}, p_{k-1}, ..., p_{k-1}}_{n' \text{ requests}}, \underbrace{p_{k+1}, p_k, ..., p_{k+1}, p_k}_{2(n'-1) \text{ requests}})$$

of length $n'(k-1) + 2(n'-1)$. By the initial contents of the cache, no page faults occur in the first $n'(k-1)$ time steps for any online algorithm.

After that, all pages in the cache except for p_k have been requested $k+1$ times. Thus when p_{k+1} is requested, at time step $T_{n'(k-1)+1}$, `LFU` evicts page p_k. Next, the adversary requests page p_k and `LFU` evicts p_{k+1} as it has been requested only once. This is iterated $n'-1$ times until both p_k and p_{k+1} have been requested $n'-1$ times each. Note that `LFU` (by induction) makes a page fault in each of the last $2(n'-1)$ time steps.

On the other hand, the optimal solution $\texttt{OPT}(I)$ simply removes a page p_j with $j \neq k$ in time step $T_{n'(k-1)+1}$ and causes no more page faults. Since

$$n' = \frac{n+2}{k+1},$$

the competitive ratio of `LFU` can be bounded from below by

$$2(n'-1) = 2 \cdot \frac{n-k+1}{k+1}$$

which is a linear function of n.

□

Remark: Taking a closer look at the previous propositions, we see that the lower bound on the competitive ratio of `LIFO` is stronger than the one for `LFU` by a factor of $\frac{k+1}{2}$.

1.4. Paging: Marking

With some basic results out of the way, we shift our focus to a general class of algorithms for pagins known as *marking algorithms*. These play an important role in the context of randomized algorithms for paging.

Definition 1.4.1 (Marking Algorithm): A *marking algorithm* works in phases and *marks* pages that were already requested; it only removes pages that are not marked. If all pages in the cache are marked and a page fault occurs, the current phase ends and a new one starts by first, unmarking all pages in the cache. Before processing the first request, all pages get marked such that the first request that causes a page fault starts a new phase. In pseudocode, the general form is as shown below.

Algorithm 6: Marking Algorithm

```
1  mark all pages in the cache                                  ▷ first page fault starts new phase
2  for every request x do
3     if x ∈ cache
4        if x is unmarked
5           mark x
6        output "0"
7     else
8        if there is no unmarked page
9           unmark all pages in the cache                      ▷ start new phase
10       p = page somehow chosen among all unmarked cached pages
11       remove p and insert x at the old position of p
12       mark x
13       output "p"
14 end
```

Marking algorithms are really nice!

Lemma 1.4.1: There are at most k page faults in one marking phase.

Proof: A page that has been marked in one phase never gets unmarked until the next phase starts (Line 9). Furthermore, throughout a phase, a marked page remains in the cache as only unmarked pages are ever removed (Line 11). Since there are at most k pages in the cache, there are at most k pages that are marked in a single phase. In any one such phase $P_{\texttt{MARK},i}$, note that every page fault leads to a page being marked (Line 12). So, there are at most k page faults in one marking phase too.

□

Proposition 1.4.1: Every marking algorithm is strictly k-competitive.

Proof: Let MARK be a fixed marking algorithm. Let I denote the given input and consider its k-phase partition into N phases $P_1, ..., P_N$. By the same argument as Proposition 1.3.1, we conclude that any optimal algorithm OPT makes at least N page faults in total on I.

Now, we show that MARK makes at most k page faults in one fixed phase P_i with $i \in [N]$. We denote the $\overline{N}$ phases defined by MARK by $P_{\texttt{MARK},1}, P_{\texttt{MARK},2}, ..., P_{\texttt{MARK},\overline{N}}$. First, we claim that both $N = \overline{N}$ and that $P_i = P_{\texttt{MARK},i}$ for $i \in [N]$. The result we want follows from verifying these two claims since MARK makes at most k page faults in one phase $P_{\texttt{MARK},i}$ (Lemma 1.4.1).

Start by observing that P_1 and $P_{\texttt{MARK},1}$ start with the first request that causes a page fault. Every phase P_i except the last one is, by definition, is a maximum-length sequence of k distinct requests. Every requested page gets marked by MARK after being requested. When k distinct pages have been requested then all pages in MARK's cache are marked. With the $(k+1)$-th distinct page p' requested since the beginning of P_i, a new phase P_{i+1} starts. In this time step, a new phase $P_{\texttt{MARK},i+1}$ is also started as there is no unmarked page left in its cache to replace with p'. So, the phases P_i and $P_{\texttt{MARK},i}$ coincide.

□

1.5. Paging: Lookahead and Resource Augmentation