3 The Paging Problem

In this chapter, we consider the paging problem already introduced in Example 1.7 in more detail.

The paging problem is one of the fundamental and oldest online problems. In fact, this problem inspired the notion of competitive analysis in [20].

In the paging problem, we are given a memory system consisting of two levels. Each level can store memory pages of a fixed size. The second level consists of slow memory, which stores a fixed set $P = \{p_1, \ldots, p_N\}$ of N pages. The first level is the fast memory (cache), which can store an arbitrary k element subset of P. Usually $k \ll N$.

The memory system is faced with a sequence $\sigma = r_1, \ldots, r_n$ of page requests. Upon a request to page p_i, the page is accessed and must be presented in the cache. If p_i is already in the cache, we have a *cache hit* and the system does not need to do anything. If p_i is not in the cache, we speak of a *cache miss* and the system must copy p_i into the cache, thereby replacing another page in the cache. A paging algorithm must, therefore, decide which page to evict from the cache upon a page fault. An online algorithm for the paging problem must process request r_i before r_{i+1} is revealed to the algorithm.

In this chapter, we consider a simple cost model for the paging problem: the *page fault model*. Under this model, the goal is simply to minimize the number of page faults. This model can be justified by the fact that a cache hit means fast (almost neglectible) access times, whereas a page fault involves access to the slow memory and evicting operations. Typical applications of the paging problem include the organization of CPU/memory systems and caching of disk accesses.

3.1 An Optimal Offline Algorithm

The paging problem is one of the few online problems where an optimal offline algorithm is known and, moreover, this algorithm has efficient (polynomial) running time. We first consider the offline problem where we are given a known sequence $\sigma = r_1, \ldots, r_n$ of page requests.

Algorithm LFD (longest forward distance) Upon a page fault, the algorithm evicts that page from the cache whose next request is furthest into the future.

Algorithm LFD is clearly an offline algorithm since it requires complete knowledge of future page requests.

Lemma 3.1 *Let* ALG *be a deterministic paging algorithm and let* $\sigma = r_1,\ldots,r_n$ *be an arbitrary request sequence. Then, for* $i = 1,\ldots,n$*, there is an algorithm* ALG_i *with the following properties:*

(i) ALG_i *serves the first* $i-1$ *requests in the same way as* ALG*.*

(ii) If the i*th request amounts to a page fault, then* ALG_i *evicts that page from the cache whose next request is furthest into the future (i.e., it uses the* LFD*-rule).*

(iii) $\text{ALG}_i(\sigma) \leqslant \text{ALG}(\sigma)$*.*

Proof: We construct ALG_i from ALG. If the ith request does not involve a page fault, then there is nothing to prove. Thus, assume for the remainder of the proof that $r_i = f$ produces a page fault for ALG.

We let p be the page that ALG replaces by f and q the page which would have been replaced according to the LFD-rule. It suffices to treat the case $p \neq q$.

Algorithm ALG_i replaces q by f. The contents of the cache after r_i are $X \cup \{p\}$ for ALG_i and $X \cup \{q\}$ for ALG, where X is a set of $k-1$ common pages in the cache. Note that $f \in X$ and that the next request to page p must occur before the next request to q since the LFD-rule caused q to be evicted and not p.

After request r_i, the algorithm ALG_i processes the remaining requests as follows until page p is requested again (page p must be requested again since q is requested after p and we assumed that $p \neq q$):

- If ALG evicts q from the cache, then ALG_i evicts p.
- If ALG evicts $x \neq q$ from the cache, then ALG_i also evicts x.

It is easy to see that, until ALG evicts q, the caches of ALG_i and ALG are of the form $X' \cup \{p\}$ and $X' \cup \{q\}$, respectively. Thus, the first case causes both caches to unify. From this moment on, both algorithms work the same and incur the same number of page faults. Hence, if the first case occurs before the next request to p, there is nothing left to show.

Now consider the situation that the first case does not occur before the next request to p. Upon the next request to p (which, as we recall once more, must occur before the next request to q), ALG incurs a page fault, while ALG_i has a cache hit. ALG now replaces some page r in the cache by p. If $r = q$, the caches unify and we are done. Otherwise, the caches of ALG_i and ALG are of the form $X'' \cup \{r\}$ and $X'' \cup \{q\}$, respectively, so the next request to q causes ALG_i to have a page fault whereas ALG has a cache-hit. At this moment, we let ALG_i replace r by q, so both caches unify again. Clearly, the number of page faults of ALG_i is at most that of ALG. □

Corollary 3.2 LFD *is an optimal offline algorithm for paging.*

Proof: Let OPT be an optimal offline algorithm and consider any request sequence $\sigma = r_1,\ldots,r_n$. Applying Lemma 3.1 with $i = 1$ gives us an algorithm OPT_1 with $\text{OPT}_1(\sigma) \leqslant \text{OPT}(\sigma)$. We re-apply the Lemma with $i = 2$ to OPT_1 and obtain OPT_2 with $\text{OPT}_2(\sigma) \leqslant \text{OPT}_1(\sigma) \leqslant \text{OPT}(\sigma)$. Repeating this process yields OPT_n, which serves σ just as LFD and satisfies $\text{OPT}_n(\sigma) \leqslant \text{OPT}(\sigma)$. □

Lemma 3.3 *Let* $N = k+1$*. Then we have* $\text{LFD}(\sigma) \leqslant \lceil |\sigma|/k \rceil$*, i.e.,* LFD *has at most one page fault for every* k*th request.*

Proof: Suppose that LFD has a page fault on request r_i and evicts page p from its cache X. Since $N = k+1$, the next page fault of LFD must be on a request to p. When this request occurs, due to the LFD-rule, all $k-1$ pages of $X \setminus \{p\}$ must have been requested in between. □

3.2 Deterministic Online Algorithms

An online algorithm must decide which page to evict from the cache upon a page fault. We consider the following popular caching strategies:

FIFO (first in/first out) Evict the page that has been in the cache for the longest amount of time.

LIFO (last in/first out) Evict the page that was brought into cache most recently.

LFU (least frequently used) Evict a page that has been requested least frequently.

LRU (least recently used) Evict a page whose last request is longest in the past.

We first address the competitiveness of LIFO.

Lemma 3.4 LIFO *is not competitive.*

Proof: Let $p_1,\ldots,p_k$ be the initial cache content and p_{k+1} an additional page. Fix $\ell \in \mathbb{N}$ arbitrary and consider the following request sequence:

$$\sigma = (p_{k+1},p_k)^\ell = \underbrace{(p_{k+1},p_k),\ldots,(p_{k+1},p_k)}_{\ell \text{ times}}$$

Here, we assume that the pages are numbered such that LIFO evicts p_k on the first request (note that this page is chosen deterministically). Hence, LIFO has a page fault on every request, so $\text{LIFO}(\sigma) = 2\ell$. Clearly, $\text{OPT}(\sigma) = 1$. Since we can choose ℓ arbitrarily large, there are no constants c and α such that $\text{LIFO}(\sigma) \leqslant c \cdot \text{OPT}(\sigma) + \alpha$ for every sequence. □

The strategy LFU is not competitive either:

Lemma 3.5 LFU *is not competitive.*

Proof: Let $p_1,\ldots,p_k$ be the initial cache content and p_{k+1} an additional page. Fix $\ell \in \mathbb{N}$ arbitrary and consider the following request sequence:

$$\sigma = p_1^\ell,\ldots,p_{k-1}^\ell,(p_k,p_{k+1})^\ell.$$

After the first request to p_k, LFU incurs a page fault on every remaining request. Hence, $\text{LFU}(\sigma) \geqslant 2\ell - 1$. On the other hand, an optimal offline algorithm can evict p_1 upon the first request to p_{k+1} and, thus, serve the sequence with only one page fault. □

3.2.1 Phase Partitions and Marking Algorithms

In order to analyze the performance of LRU and FIFO, we use an important tool known as k-*phase partition*. Given a request sequence σ, phase 0 is defined to be the empty sequence. Phase $i+1$ consists of the maximum subsequence of σ starting with the request directly after the last request of phase i such that the phase contains requests to at most k different pages.

$$\sigma = \overbrace{r_1,\ldots,r_{j_1}}^{k \text{ different pages}}, \overbrace{r_{j_1+1},\ldots,r_{j_2}}^{k \text{ different pages}},\ldots, \overbrace{r_{j_r+1},\ldots,r_n}^{\leqslant k \text{ different pages}}$$

The k-phase partition is uniquely defined and independent of any algorithm. Every phase, except for possibly the last phase, contains requests to exactly k different pages.

Given the k-phase partition, we define a marking of all pages in the universe as follows: At the beginning of a phase, all pages are unmarked. During a phase, a page is marked upon the first request to it. Notice that the marking does not depend on any particular algorithm. We call an algorithm a *marking algorithm* if it never evicts a marked page.

Theorem 3.6 *Any marking algorithm is* k*-competitive for the paging problem with cache size* k*.*

Proof: Let ALG be a marking algorithm. We consider the k-phase partition of the input sequence σ. ALG incurs at most k page faults in a phase: upon a fault on page f, the page is marked and brought into the cache. Hence, since ALG never evicts a marked page and at most k different pages are requested in a single phase, ALG can incur at most k page faults in a phase.

We now slightly modify our partition of the input sequence to show that an optimal offline algorithm must incur at least one page fault less than there are (nonempty) phases in the partition: Let ℓ be the last phase of the sequence. We define segment 0 to consist of only the first request of σ and segment i for $i = 1,\ldots,\ell-1$ to start with the second request in phase i and end with the first request of phase $i+1$.

Now consider some segment $i \in \{1,\ldots,\ell-1\}$. The last request in segment i must be to a page that has not been requested in phase i since otherwise phase i would not be maximal. At the end of segment $i-1$, the cache of OPT contains the first page requested during phase i, say p. Until the end of phase i, there will be requests to exactly $k-1$ other pages and, until the end of segment i, requests to exactly k other pages. Hence, since OPT can only keep k pages in the cache and one position is already occupied by p, it must incur a page fault within segment i. This shows that $\text{OPT}(\sigma) \geqslant \ell-1$. As $\text{ALG}(\sigma) \leqslant \ell \cdot k$, we obtain $\text{ALG}(\sigma) \leqslant k \cdot \text{OPT}(\sigma) + k$. □

Lemma 3.7 LRU *is a marking algorithm.*

Proof: Suppose that LRU evicts a marked page p during a phase on some request r_i. Consider the first request r_j to p during the phase, which caused p to be marked (then, by definition, $j < i$). At this point in time, p is the page in the cache of LRU whose last request has been most recently. Let $Y = X \cup \{p\}$ denote the cache contents of LRU after request r_j. In order to have LRU evict p, all other pages in X must have been requested during the phase (or replaced by different pages due to page faults, in which case we denote the new set of pages by Y again). Thus, there have been requests to all k pages in Y during the phase. Clearly, $r_i \notin Y$ since otherwise LRU would not incur a page fault on request r_i. But this means that $k+1$ different pages have been requested during the phase, which contradicts the definition of the phase partition. □

Corollary 3.8 LRU *is* k*-competitive for the paging problem with cache size* k*.* □

In contrast, FIFO is not a marking algorithm. Nevertheless, FIFO can be shown to be competitive by a slight modification of the proof of Theorem 3.6

Theorem 3.9 FIFO *is* k*-competitive for the paging problem with cache size* k*.*

Proof: See Exercise 3.1. □

3.2.2 A Lower Bound for Deterministic Algorithms

Theorem 3.10 *Let* ALG *be any deterministic online algorithm for the paging problem with cache size* k*. If* ALG *is* c*-competitive, then* $c \geqslant k$*.*

Proof: Let $\{p_1,\ldots,p_k\}$ denote the initial cache contents and let p_{k+1} be an additional page. We define a request sequence $\sigma = r_1, r_2, \ldots$ inductively as follows: We start with $r_1 := p_{k+1}$. If ALG evicts page p upon request r_i, then $r_{i+1} := p$. Clearly, ALG has a page fault on every request, so $\text{ALG}(\sigma) = |\sigma|$. On the other hand, by Lemma 3.3, $\text{OPT}(\sigma) \leqslant \lceil |\sigma|/k \rceil$, which proves the claim since we can choose $|\sigma| := l \cdot k$ for every $l \in \mathbb{N}$. □

3.3 Randomized Online Algorithms

In this section, we present a randomized algorithm that beats the lower bound for deterministic algorithms stated in Theorem 3.10. More specifically, it achieves a competitiveness of $2H_k$, where

$$H_k = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k}$$

denotes the k*th Harmonic number*. Observe that $\ln k < H_k \leqslant 1 + \ln k$. Hence, randomization gives an exponential boost in competitiveness for the paging problem. Our randomized algorithm RANDMARK works as follows:

RANDMARK Initially, all pages are unmarked. Upon a request to a page p that is not in the cache, p is marked and brought into the cache. The page to be evicted is chosen uniformly at random among all unmarked pages in the cache (if all pages in the cache are already marked, then all marks are erased first).

Upon a request to a page p that is already in the cache, p is marked.

The following lemma shows that RANDMARK deserves its name (it is a randomized marking algorithm):

Lemma 3.11 *Let $\sigma = r_1, \ldots, r_n$ be any input sequence. After each request r_i, the marked pages of* RANDMARK *are exactly the pages that are marked with respect to the k-phase partition of σ. In particular,* RANDMARK *never evicts a page that is marked with respect to the k-phase partition of σ and, at the end of a phase, the cache of* RANDMARK *contains all pages requested in the phase.*

Proof: We prove the claim by induction on i. After r_1, the only marked page with respect to both markings is the page that was requested in r_1. For the induction step, we distinguish two cases:

Case 1: r_i is the last request of a phase.
In this case, exactly the k pages requested in the phase ending with r_i are marked with respect to the k-phase partition after r_i and only page r_{i+1} is marked with respect to the k-phase partition after r_{i+1}. Hence, by induction hypothesis, exactly the k pages requested in the phase ending with r_i are marked pages of RANDMARK after r_i. In particular, exactly these k pages are in RANDMARK's cache after r_i. Thus, RANDMARK must have a page fault on request r_{i+1} since r_{i+1} starts a new phase (so it must be different from the k pages requested in the phase ending with r_i). Hence, RANDMARK erases all its marks before marking page r_{i+1} and bringing it into the cache. Thus, page r_{i+1} is the only marked page also for RANDMARK after r_{i+1}.

Case 2: r_i is *not* the last request of a phase.
If page r_{i+1} has already been requested in the phase at least once before request r_{i+1}, the statement for $i+1$ follows directly from the induction hypothesis since both sets of marked pages remain unchanged. If r_{i+1} has not been requested before in the phase, at most $k-1$ pages are marked with respect to the k-phase partition after r_i. Hence, by induction hypothesis, at most $k-1$ pages in the cache of RANDMARK are marked with respect to RANDMARK's marking after r_i. Consequently, page r_{i+1} is added to both sets of marked pages on request r_{i+1} and the sets are still identical after r_{i+1}.

□

Theorem 3.12 RANDMARK *is $2H_k$-competitive against an oblivious adversary.*

Proof: For the analysis, we again use the k-phase partition.

We call a page that is requested during a phase and that is not in the cache of RANDMARK at the beginning of the phase a *new page*. All other pages requested during a phase will be called *old pages*.

Consider an arbitrary phase i and denote by m_i the number of new pages in the phase. RANDMARK incurs a page fault on every new page. Notice that RANDMARK has *exactly* m_i faults on new pages since a marked page is never evicted until the end of the phase.

We now compute the expected number of page faults on old pages. To do so, we estimate the probability of a page fault on the jth request to an old page for each j. Note that, since RANDMARK keeps each requested page in the cache until the end of the phase, repeated requests to the same page cannot cause RANDMARK to have additional page faults, so we may assume that all requests to (old or new) pages are to *different* pages.

There are requests to at most $k-m_i$ (different) old pages in phase i (the last phase might not be complete). Let $\ell = \ell(j)$ denote the number of (different) new pages requested before the jth request to an old page.

When the jth old page is requested, RANDMARK has evicted exactly ℓ unmarked old pages from the cache due to the previous requests to new pages, and $(j-1)$ old pages have been marked due to the $(j-1)$ previous requests to old pages. Hence, there are exactly $k-\ell-(j-1)$ unmarked old pages left in the cache. Since the total number of unmarked old pages at this point is exactly $k-(j-1)$ and the jth old page requested must be one of these, the probability that it is in the cache is exactly $(k-\ell-(j-1))/(k-(j-1)) = 1-\ell/(k-(j-1))$. Consequently, RANDMARK has a page fault on the jth request to an old page with probability exactly $\ell/(k-(j-1)) \leqslant m_i/(k-j+1)$. Hence, the total number X_i of page faults in phase i satisfies

$$\mathbb{E}[X_i] \leqslant m_i + \sum_{j=1}^{k-m_i} \frac{m_i}{k-j+1} = m_i(H_k - H_{m_i} + 1) \leqslant m_i H_k,$$

and by linearity of expectation we obtain that $\mathbb{E}[\text{RANDMARK}(\sigma)] = \sum_i \mathbb{E}[X_i]$.

We now show that $\text{OPT}(\sigma) \geqslant \sum_i m_i/2$. Consider a phase $i \geqslant 2$. Together, phase $i-1$ and phase i contain requests to at least $k+m_i$ different pages. Since OPT can keep only k pages in the cache, it will incur at least m_i page faults during phases $i-1$ and i together. Since OPT must have exactly m_1 page faults in the first phase (since it starts with the same cache content as RANDMARK), we get $\text{OPT}(\sigma) \geqslant \sum_i m_{2i}$ and $\text{OPT}(\sigma) \geqslant \sum_i m_{2i+1}$, which amounts to

$$\begin{aligned}\text{OPT}(\sigma) &\geqslant \frac{1}{2}\left(\sum_i m_{2i} + \sum_i m_{2i+1}\right) = \sum_i \frac{m_i}{2} = \sum_i \frac{m_i H_k}{2H_k} \\ &\geqslant \frac{1}{2H_k}\sum_i \mathbb{E}[X_i] = \frac{1}{2H_k}\mathbb{E}[\text{RANDMARK}(\sigma)].\end{aligned}$$

This proves the claim. □

We now address the question how well randomized algorithms can perform.

Theorem 3.13 *Any randomized algorithm for the paging problem with cache size k has competitive ratio at least H_k against an oblivious adversary.*

Proof: Our construction uses a subset of the pages P of size $k+1$ consisting of the initial cache contents and one additional page. Let $\bar{p}$ be a distribution over the input sequences

with the property that the ℓth request is drawn uniformly at random from P independent of all previous requests.

Clearly, any deterministic paging algorithm faults on each request with probability $1/(k+1)$. Hence, we have

$$\mathbb{E}_{\bar{p}}\left[\mathrm{ALG}(\sigma^n)\right] = \frac{n}{k+1}, \tag{3.1}$$

where σ^n denotes a random request sequence of length n. By Yao's Principle, the number

$$r := \lim_{n\to\infty} \frac{n}{(k+1)\mathbb{E}_{\bar{p}}\left[\mathrm{OPT}(\sigma^n)\right]}$$

is a lower bound on the competitive ratio of any randomized algorithm against an oblivious adversary. Thus, our goal is to show that $r \geqslant H_k$. The desired inequality will follow if we can show that

$$\lim_{n\to\infty} \frac{n}{\mathbb{E}_{\bar{p}}\left[\mathrm{OPT}(\sigma^n)\right]} \geqslant (k+1)H_k. \tag{3.2}$$

We have a closer look at the optimal offline algorithm. We partition randomly generated sequences into stochastic phases as follows: The partition is like the k-phase partition with the sole difference that the starts/ends of the phases are given by random variables.

Let X_i, $i = 1,2,\ldots$, be the sequence of random variables in which X_i denotes the number of requests in phase i in an infinite sequence of requests generated from $\bar{p}$. Notice that the X_i are all independent and identically distributed. The jth phase starts with request r_{S_j+1} and ends with request $r_{S_{j+1}}$, where $S_j := \sum_{i=1}^{j-1} X_i$. A random request sequence σ^n of length n can be considered as a prefix of such an infinite request sequence. If we let $N(n)$ denote the number of complete phases in σ^n, we then have

$$N(n) = \max\{j : S_{j+1} \leqslant n-1\}.$$

Our proof of (3.2) proceeds in three steps:

(i) We show that

$$\mathrm{OPT}(\sigma^n) \leqslant N(n)+1. \tag{3.3}$$

(ii) We prove that

$$\lim_{n\to\infty} \frac{n}{\mathbb{E}_{\bar{p}}\left[N(n)\right]} = \mathbb{E}_{\bar{p}}\left[X_i\right]. \tag{3.4}$$

(iii) We establish that the expected value $\mathbb{E}_{\bar{p}}\left[X_i\right]$ satisfies

$$\mathbb{E}_{\bar{p}}\left[X_i\right] = (k+1)H_k. \tag{3.5}$$

The inequalities (3.3) – (3.5) imply that

$$\lim_{n\to\infty} \frac{n}{\mathbb{E}_{\bar{p}}\left[\mathrm{OPT}(\sigma^n)\right]-1} = \lim_{n\to\infty} \frac{n}{\mathbb{E}_{\bar{p}}\left[\mathrm{OPT}(\sigma^n)-1\right]} \geqslant (k+1)H_k. \tag{3.6}$$

But since $\mathbb{E}_{\bar{p}}\left[\mathrm{OPT}(\sigma^n)\right] \to \infty$ for $n \to \infty$, the limits on the left hand sides of (3.2) and (3.6) coincide.

(i) By Corollary 3.2, we know that $\mathrm{OPT} = \mathrm{LFD}$. Moreover, we have seen in the proof of Lemma 3.3 that, if $\mathrm{OPT} = \mathrm{LFD}$ has a page fault on some request and evicts a page p from its cache X, the next page fault must be on a request to p and all $k-1$ pages of $X \setminus \{p\}$ must have been requested in between. Hence, the next page fault cannot occur until the start of the next phase, which shows that OPT has at most one page fault in each phase (including the last phase, which may be incomplete). This proves (3.3).

(ii) The family of random variables $\{N(n) : n \in \mathbb{N}\}$ forms a *renewal process*. Equation (3.4) is a consequence of the *Renewal Theorem* (see [6, Appendix E]).

(iii) Our problem is an application of the *coupon collector's problem* (see Exercise 3.2). We are given $k+1$ coupons (corresponding to the pages in P) and a phase ends one step before we have collected all $k+1$ coupons. Thus, the expected value $\mathbb{E}_{\bar{p}}[X_i]$ is one less than the expected number of trials in the coupon collectors problem, which, by Exercise 3.2, is $(k+1)H_{k+1}$. Hence, we obtain

$$\mathbb{E}_{\bar{p}}[X_i] = (k+1)H_{k+1} - 1 = (k+1)H_k.$$

□

3.4 Exercises

Exercise 3.1
Prove Theorem 3.9.

Exercise 3.2
In the coupon collector's problem, there are n types of coupons and, at each trial, a coupon is chosen at random. Each random coupon is equally likely to be any of the the n types and the random choices of the coupons are mutually independent. Let X denote the number of trials required to collect at least one of each type of coupon. Show that $\mathbb{E}[X] = nH_n$.

Hint: For $i = 1, \ldots, n$, consider the number Z_i of trials needed to obtain the ith type of coupon after i types have already been collected. What is the expected value of Z_i?

Exercise 3.3 (Online graph matching)
Consider the following online variant of the *matching Problem*: Given is a bipartite graph $G = (H \cup D, R)$, i.e., $H \cap D = \varnothing$ and each directed edge $r \in R$ is of the form $r = (h, d)$, where $h \in H$ and $d \in D$.

Suppose $G = (V, E)$ is an undirected graph. A *matching* is a subset $M \subseteq E$ such that M contains no two incident edges. In a *maximum matching* it is not possible to add an edge without destroying the matching property. A matching M is *perfect* if each node in V is incident to an edge in M.

The dating service *Online-Matching* received data from n men $H = \{h_1, \ldots, h_n\}$ who are interested in a wonderful woman. The service organizes a dance party where these men can find their love. Therefore, invitations have been sent to n promising women $D = \{d_1, \ldots, d_n\}$. With the invitation, they received an overview of all interested men. Each woman creates a list of her preferred men. At the entrance to the party, each woman is assigned a dance partner from her list. If there is no man on a girl's list left without a partner, then she is paid some financial compensation and is sent home. Of course, the goal of the dating service is to pair up as many people as possible.

This problem can be modelled as an *online* version of the problem of finding a perfect matching: We are given a bipartite graph $G = (H \cup D, E)$ with $2n$ nodes and we assume that G has a perfect matching.

An online algorithm knows all men H from the beginning. A request r_i consists of a neighborhood $N(d_i) = \{h \in H : (h, d_i) \in E\}$ of a "woman-node" d_i. The online algorithm has to decide upon the arrival of the request to which man in $N(d_i)$ this woman should be assigned (if possible). The sequence $\sigma = r_1, \ldots, r_n$ consists of a permutation of women and the objective is to obtain as many couples as possible.

Now consider the following *very simple* algorithm: as soon as a woman arrives, she is assign to *any* man on her preference list that is still single. If there is no such man left, then she is sent home.

(a) Prove that this algorithm is 2-competitive.

(Hint: Assume that M is a maximum matching with $|M| < n/2$. Denote by H' the set of men who got a partner through M. Then, $|H'| < n/2$. Make use of the fact that G has a perfect matching.)

(b) Show that *every* deterministic online algorithm (and even each randomized online algorithm against an adaptive offline adversary) has a competitive ratio of no less than 2 for the problem above.