Recitation 9: Hashing and Online Algorithms

1 Hashing

1.1 Universal Hashing

Recall from lecture that a family $\mathcal{H}$ of hash functions is called universal if those functions map keys from $U \rightarrow \{0, 1, 2, \ldots, m - 1\}$ and

$$\forall x \neq y \in U : \Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}.$$  

$U$ is called the universe of keys, and this definition says that, for a hash function $h$ drawn at random from the family $\mathcal{H}$, the probability that any two keys in the universe collide is at most $1/m$.

The following important theorem follows directly from linearity of expectation applied to this definition: Consider some $S \subseteq U$ and some key $x \in S$. If we choose some $h \in \mathcal{H}$ at random and hash all keys in $S$ into $m$ slots, then

$$\mathbb{E}[\# \text{ of collisions with } x] \leq \frac{|S|}{m}.$$  

Furthermore, we can say that the expected number of elements from $S$ mapped to $h(x)$ is $\leq 1 + \frac{|S|}{m}$. This is true because $x$ obviously maps to $h(x)$, yielding the “1”, and there are, in expectation, at most $\frac{|S|}{m}$ other elements that also map to $h(x)$. Thus, it follows that insertion, deletion, and search have an expected running time of $O(1 + \frac{|S|}{m})$. Note that, when all elements map to the same slot, we encounter a worst-case running time of $\Theta(|S|)$ for these operations.

1.2 Perfect Hashing

Our goal in perfect hashing is, given $n$ keys, to construct a static hash table of size $\Theta(n)$ such that the worst-case lookup time is $\Theta(1)$. (“Static” means that all keys are known ahead of time.)
1.2.1 Attempt 1 — $\Theta(n^2)$ space

We can easily achieve perfect hashing if we allow our hash table to use $\Theta(n^2)$ space. Specifically, note that there are $\binom{n}{2}$ pairs of keys. By the definition above, the probability that a universal hash function maps a pair of (distinct) keys to the same value is at most $\frac{1}{m}$. It follows from linearity of expectation (much like the theorem above) that

$$\mathbb{E}[\text{total # of collisions}] \leq \binom{n}{2} \cdot \frac{1}{m}.$$ 

Now, let the size of the hash table be $n^2$ — i.e., set $m = n^2$. It follows that

$$\mathbb{E}[\text{total # of collisions}] \leq \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2}.$$

Markov’s inequality states that $\Pr[X \geq c \cdot \mathbb{E}[X]] \leq \frac{1}{c}$. Letting the random variable $X$ be the number of collisions and $c$ be 2, we find that $\Pr[\text{# of collisions} \geq 1] \leq \frac{1}{2}$. Thus, $\Pr[\text{no collisions}] > \frac{1}{2}$.

If we pick some hash function $h \in \mathcal{H}$ at random, there is at least a probability of $1/2$ that it will result in no collisions. Consequently, we can keep picking $h \in \mathcal{H}$ at random until we find one that results in no collisions; the expected number of times we have to do this is 2. While we end up with no collisions (and thus a worst-case running time of $\Theta(1)$ for a lookup), this method requires that our hash table use $\Theta(n^2)$ space.

1.2.2 Attempt 2 — $\Theta(n)$ space

It is indeed possible to achieve perfect hashing using $\Theta(n)$ space. The idea is to use a two-level hash table with universal hashing. In particular, we are again given $n$ keys. The “first level” consists of a hash table $A$, which uses a hash function $h$ to map the $n$ keys to $m$ slots. Each slot $i$ at the first level stores a hash function $h_i$ and a pointer to another hash table $B_i$. Each hash table $B_i$ is considered part of the “second level”, and has $m_i$ slots. These slots in $B_i$ store the actual data. Thus, if we wish to lookup the key $x$, we first look in the first level for a slot $i = h(x)$, and then return $B_i[h_i(x)]$. 

2
In order to achieve our desired running time of \( \Theta(1) \) and our desired space of \( \Theta(n) \), we set the following properties:

- Let the size of the first level hash table be \( n \). That is, set \( m = n \).
- We do not want collisions in the second level hash tables. By using the method in Attempt 1, we can avoid collisions by setting the size of the second level hash table to \( n_i^2 \), where \( n_i \) is the number of keys \( x \) such that \( h(x) = i \). That is, set \( m_i = n_i^2 \).

First, note that the first level hash table clearly has size \( \Theta(n) \). There are \( n \) second level hash tables, each of size \( \Theta(n_i^2) \), so the total size of the second level hash tables is \( \Theta \left( \sum_{i=0}^{n-1} n_i^2 \right) \). It can be shown that

\[
    n \leq \mathbb{E} \left[ \sum_{i=0}^{n-1} n_i^2 \right] \leq 2n.
\]

Thus, the expected total storage of the second level is \( \Theta(n) \), so the expected storage overall is \( \Theta(n) \).

Like in Attempt 1, we can keep picking hash functions \( h \in \mathcal{H} \) at random for both levels until the total storage is \( \Theta(n) \) and there are no collisions in the hash tables \( B_i \). This yields a total actual storage of \( \Theta(n) \) and, because there are no collisions in the second level, the running time of the lookup operation is \( \Theta(1) \). Hence, we constructed a method that gives a constant lookup time with linear space, our original goal for perfect hashing.

# 2 Online Algorithms

An online algorithm is an algorithm that can process inputs in a serial fashion without knowing all the inputs in advance. Because online algorithms are forced to make decisions based only on the input they have received so far, it is often impossible for them to achieve an optimal result. We use competitive analysis to compare the performance of an online algorithm with the optimal offline algorithm that knows all the inputs in advance.

Let \( A(\sigma) \) be an online algorithm with input sequence \( \sigma \), and let \( A^*(\sigma) \) be an optimal offline algorithm on the same input sequence. Let \( C_A(\sigma) \) and
$C_{A^*}(\sigma)$ be their respective costs. We say that the online algorithm $A$ is $\alpha$-competitive if, for every input sequence $\sigma$,

$$C_A(\sigma) \leq \alpha C_{A^*}(\sigma).$$

We also say that $\alpha$ is the competitive ratio of algorithm $A$. Keep in mind that this is comparing the cost of an algorithm, not the quality of a solution itself.

## 2.1 Paging

Paging is an important problem in the design of computer systems. We can model a machine’s memory as consisting of two parts:

- An unlimited number of pages of slow memory.
- A cache consisting of $k$ pages of fast memory.

When a page is requested, if the requested page is not in the cache (a cache miss), some other page in the cache must be evicted so that the requested page can take its place in the cache. A paging strategy is the method that chooses which page to evict on a cache miss.

Below are some examples of paging strategies:

- **Random** — Evict a random page.
- **FIFO** — Evict the page added to the cache before any other pages.
- **Frequency counts** — Evict the least frequently used page.
- **LRU** — Evict the least recently used/requested page.

Note that these strategies are all online algorithms because they can only process page requests in a serial fashion, without knowing all the requests in advance.

We will first show that the LRU strategy is $k$-competitive. Then, we will show that all deterministic online paging algorithms have a lower bound of $k$ for the competitive ratio.
2.1.1 LRU is $k$-competitive

The cost of an online paging algorithm can be considered to be its number of cache misses, so this is what we will analyze.

We can first partition the input sequence into phases. The first phase begins immediately after the LRU strategy first encounters a cache miss. A phase ends immediately after the LRU strategy encounters $k$ cache misses since the start of the phase, and the next phase begins at this point. Thus, by our definition, the LRU strategy has $k$ cache misses per phase.

Now we will show that the optimal offline algorithm has at least 1 cache miss per phase. First, note that, in some phase, there are two possibilities for the LRU strategy:

- It encounters a cache miss twice for the same page $p$ in this phase.
- It encounters cache misses on $k$ distinct pages in this phase. If the above case does not apply, this case must apply because the strategy has $k$ cache misses per phase.

Consider some phase such that the LRU strategy encounters a cache miss twice for the same page $p$ in this phase. Since the LRU strategy encountered a cache miss twice, $p$ must have been evicted at some point in between the two requests; thus, by the LRU strategy, there must have been at least $k$ other distinct pages requested in between the two requests for $p$. It follows that there are at least $k + 1$ distinct pages requested in this phase. Hence, any optimal offline algorithm must encounter at least one cache miss in this phase.

Consider some phase such that the LRU strategy encounters cache misses on $k$ distinct pages in this phase. Let page $p$ be the last cache miss of the previous phase; it follows that $p$ is in the cache at the start of this phase. Consider two possibilities:

- If $p$ is one of the $k$ cache misses (by the LRU strategy) in this phase, then at least $k$ other distinct pages must have been requested (otherwise $p$ would not have been evicted). So, in this case, there are at least $k + 1$ distinct pages requested in this phase and, as a result, any optimal offline algorithm must encounter at least one cache miss in this phase.

- If $p$ is not one of the $k$ cache misses (by the LRU strategy) in this phase, then $k$ distinct pages are requested, none of which is $p$. But $p$
must also be in the cache of the optimal offline algorithm at the start of this phase. Those \( k \) distinct pages will be put in the cache by the optimal offline algorithm in this phase; at some point this will cause a cache miss and evict \( p \). Thus, the optimal offline algorithm must encounter at least one cache miss in this phase.

Therefore, after considering all of our cases, we see that any optimal offline algorithm must encounter at least 1 cache miss per phase.

If \( A \) is the LRU strategy and \( A^* \) is the optimal algorithm, we have \( C_A = k \) and \( C_{A^*} \geq 1 \). Since

\[
C_A \leq kC_{A^*}
\]

the LRU strategy is \( k \)-competitive.

### 2.1.2 Deterministic paging algorithms have a lower bound of \( k \)-competitive

To show a lower bound of \( k \) for the competitive ratio of deterministic online paging algorithms, we will use the concept of an adversary. In particular, the adversary will devise a worst-case input sequence for the online algorithm. It will also “service” that sequence by playing the role of the optimal offline algorithm when given this input sequence.

Let \( A \) be the online paging algorithm. The adversary knows the algorithm \( A \), and devises an input sequence in which each request comes from a set of \( k + 1 \) pages. This set of \( k + 1 \) pages consists of the \( k \) pages initially in the cache and some other page. At every point, the adversary requests of \( A \) the one page that is outside of the cache. The adversary can do this for any number of requests, and on each request \( A \) encounters a cache miss. Hence, if \( C_A(\sigma) \) is the cost of \( A \) on the input sequence \( \sigma \), then \( C_A(\sigma) = |\sigma| \).

Now, the adversary will “service” this input sequence in the best possible way — i.e., it will act as the optimal offline algorithm. At every cache miss, the adversary will choose to evict the page whose next request occurs farthest in the future. Suppose it encounters a cache miss and chooses to evict the page \( x \) from the cache. Since there are \( k + 1 \) pages in the input sequence, \( x \) is currently the only page missing from the cache; thus, the next cache miss occurs when \( x \) is next requested. By the eviction strategy, \( x \) is the page in the cache whose request is farthest in the future, so there must be at least \( k - 1 \) pages before \( x \) is next requested. Therefore, there are at least \( k - 1 \) requests between any two cache misses, so the optimal offline algorithm encounters a
cache miss on at most every $k$'th request. If $C_A^*(\sigma)$ is the cost of the optimal offline algorithm on this input sequence $\sigma$, then $C_A^*(\sigma) \leq \lceil \frac{|\sigma|}{k} \rceil$.

The condition for $\alpha$-competitiveness is $C_A(\sigma) \leq \alpha C_A^*(\sigma)$, or

$$|\sigma| \leq \alpha \left\lceil \frac{|\sigma|}{k} \right\rceil.$$ 

This is satisfied only when $\alpha \geq k$, which shows that any deterministic online paging algorithm has a lower bound of $k$ for its competitive ratio.

Note that this theorem only applies to deterministic online paging algorithms because an adversary would not be able to devise a worst-case input sequence to a randomized online paging algorithm.