Lecture 7: Introduction to Online Algorithms

So far we have studied offline algorithms – also known as prescient algorithms – which have access to their full input sequence a priori. With full knowledge of the task beforehand, these algorithms can often produce more-optimal solutions. Differently, algorithms which receive their inputs in a serial manner, without knowledge of future inputs, are known as online algorithms.

1 Online Problem Examples

Today, we will explore three classic online problems:

1. The Canadian Traveler’s Problem
2. The Ski Rental Problem
3. The Paging Problem

1.1 Canadian Traveler’s Problem

It is winter in Canada. A Canadian traveler has a road map and must navigate a network of roads to get from point A to point B, but some sections of road are blocked by snow.

In the offline version of this problem, the traveler knows which roads are blocked before he begins his journey, turning this into a simple pathfinding problem. The traveler can use any shortest path algorithm, ignoring the roads that are blocked to determine an optimal path to his destination.

In the online version of this problem, the traveler must explore the road network to determine which routes are accessible; specifically, he must arrive at a road and traverse it before he will know whether it is blocked.
1.2 Ski Rental Problem

Skis cost $1/day to rent and $T$ to buy. After $S$ days, you will suffer a terrible accident and break your leg (which is why you should stay off the mountain in the first place); you will never ski again.

In the offline version of this problem, you know how many days will pass before the accident will occur before you decide to rent or buy skis (i.e., you know the value of $S$). Thus, minimizing cost is straightforward using algorithm $A_0$, the optimal offline algorithm:

- If $T < S$, then buy.
- If $T > S$, then rent every day until you break your leg.
- If $S = T$, then buying and renting will both cost the same.

The analysis of the online version of this problem is more challenging. In this version of the problem we do not know $S$ in advance. Observe that any deterministic strategy we apply can be described as follows: we will rent for $k$ days, and then buy on day $k + 1$ if we have not broken our leg by then.

1.3 Paging Problem

Consider a system where pages of data may be stored in a cache for quick access or in memory for slower access. Although we would prefer to store all data pages in the cache for quick access, this is expensive. Our system’s cache allows a maximum of $k$ pages of data to be quickly retrieved; all other data pages must be stored in memory and swapped into the cache if and when they are requested. The user requests a sequence of pages $\langle \sigma \rangle$.

In this problem, we consider a “hot” cache – one that is full when the problem begins. If page $\sigma_i$ is cached, it is quickly retrieved at no cost. But, if $\sigma_i$ is not cached, then some other page, $\sigma_j$, already in the cache must be evicted from the cache to make room for $\sigma_i$, resulting in a retrieval cost of 1. The latter case is referred to as a page fault. The cost of an algorithm on a given sequence $\langle \sigma \rangle$ is simply the number of page faults.
In the offline formulation of this problem, we know the full sequence of the data pages that will be requested in advance. Differently, in the online formulation, we only learn what pages will be requested as the requests arrive. Specifically, when we learn \( \sigma_i \)'s value at time \( i \), we must determine which cached data page should be evicted based on \( \sigma_1, \sigma_2, \ldots, \sigma_{i-1} \), without knowing the values of \( \sigma_{i+1}, \sigma_{i+2}, \ldots, \sigma_n \).

## 2 Measuring Online Algorithm Performance

Since online algorithms must make decisions without a priori knowledge of the full sequence of input, it is often impossible for them to find an optimal solution – even when time complexity is disregarded. Instead of comparing algorithms based on time complexity, the relative effectiveness of an online algorithm is measured with competitive analysis, which uses the performance of an offline algorithm as a benchmark.

**Definition 2.0.1.** An online algorithm \( A \) is said to be \( \alpha \)-competitive if for any input sequence \( \langle \sigma \rangle \), \( \text{COST}_A(\sigma) \leq \alpha \text{COST}_{\text{MIN}}(\sigma) \), where \( \text{COST}_A(\sigma) \) denotes cost of online algorithm \( A \) on \( \langle \sigma \rangle \) and \( \text{COST}_{\text{MIN}}(\sigma) \) denotes the cost of the optimal offline algorithm on \( \langle \sigma \rangle \).

The competitive ratio of an algorithm \( A \) is calculated:

\[
\frac{\text{COST}_A(\sigma)}{\text{COST}_{\text{MIN}}(\sigma)},
\]

where \( \text{MIN} \) denotes the optimal offline algorithm.

### 2.1 Ski Rental Problem

As already discussed, the optimal offline algorithm \( A_0 \) says to buy if and only if \( S \geq T \). Since we either rent for \( S \) days and spend $S$, or we buy immediately and spend $T$, the value of \( A_0 \) on any given input is simply \( \min\{S, T\} \).

Next, we will consider two possible online algorithms, \( A_1 \) and \( A_2 \), for the ski rental problem. The algorithm \( A_1 \) is simple: buy skis on the very first
day. The algorithm $A_2$ is defined as follows: 1) rent skis for the first $T - 1$ days, and 2) if the accident has not occurred by day $T$, then buy the skis.

Now, we will analyze each algorithm using competitive analysis, where $A_0$ is the optimal offline algorithm. We begin with analysis of online algorithm $A_1$. When $S \geq T$, both algorithms $A_0$ and $A_1$ result in the purchase of skis, resulting in a cost of $T$. However, $A_0$ and $A_1$ behave differently when $S < T$. In the worst case, the skiing accident occurs on the first day. In this case, $A_0$ provides the optimal solution of $1$; differently, $A_1$ leads to a cost of $T$. Thus, in the worst-case, $A_1$ has an associated cost that is a factor of $T$ greater than $A_0$, meaning that $A_1$ is $T$-competitive:

$$\frac{\text{COST}_{A_1}(\sigma)}{\text{COST}_{A_0}(\sigma)} = \frac{T}{1} = T.$$ 

Next, we will analyze online algorithm $A_2$. If $S < T$, then $A_0$ and $A_2$ both result in renting every day for a cost of $S$. However, if $S \geq T$, then $A_0$ and $A_2$ behave differently. Specifically, if we break our leg after day $T$, then $A_0$ will have a cost of $T$ from buying. Differently, $A_2$ will have a cost of $2T - 1$ (i.e., $T - 1$ from renting and an additional $T$ from buying). In this worst case:

$$\frac{\text{COST}_{A_2}(\sigma)}{\text{COST}_{A_0}(\sigma)} = \frac{2T - 1}{T} \leq 2,$$

meaning that $A_2$ is 2-competitive.

### 2.2 Paging Problem

The optimal offline algorithm has knowledge of the entire sequence of data page requests; this information allows the algorithm to evict either 1) a page that is never again requested or 2) the page that is requested again the farthest into the future. This prescient eviction strategy is known as Longest Forward Distance (LFD).
For the online formulation of the problem, there are many possible page-eviction strategies. A few such strategies are:

- Least Frequently Used (LFU). This algorithm will keep track of the number of times the page has been requested and remove the page that has been requested the fewest number of times so far.
- Least Recently Used (LRU). This algorithm will keep track of how long it has been since each page has been accessed, and will swap out the least recently used page.
- Most Recently Used (MRU). This algorithm works like LRU, except it swaps out the most recently used page.
- First-In First-Out (FIFO). This algorithm will always remove the page that has been in memory for the greatest amount of time.
- Last-In First-Out (LIFO). This algorithm will always remove the page that has been in memory for the least amount of time.
- Random Selection. This algorithm selects a page to remove at random.

LIFO can be shown to perform very poorly. Assume $k$ page slots and consider the following sequence: $⟨σ_1, ..., σ_{k-1}, p, q, p, p, q, q, p, q, p, q, ...⟩$, where $σ_i \neq p, σ_i \neq q, \forall i \in \{1...k-1\}$. LIFO will choose to evict $p$ to fit $q$, then it will evict $q$ to fit $p$, etc, for as long a sequence as we choose to construct, ensuring that there is no bound on the worst-case scenario of our competitive analysis.

We will compare LRU to LFD, but first we must demonstrate that LFD is in fact an optimal offline algorithm.

**Theorem 2.2.1.** LFD is an optimal algorithm for the offline paging problem.

**Proof.** Consider a finite sequence of page requests. Suppose there exists an algorithm $A$ that produces fewer page faults than LFD on some input sequence. In order to produce a different result, at some unique point in time $A$ and LFD must choose to do something different. Specifically, there
exists an \( i \) such that \( \sigma_i \) is not in fast memory and \( A \) removes some page \( p \) and \( LFD \) removes page \( q \), with \( p \neq q \).

Define \( t \) as the first time at which algorithm \( A \) removes page \( q \) after time \( i \). Let time \( j \) be the time at which the next request for \( p \) after time \( i \) occurs, and let \( l \) be the time at which the next request for \( q \) after time \( i \) occurs. Note that By definition of \( LFD \), \( i < j < l \) (otherwise \( LFD \) would have removed \( p \) at time \( j \), not \( q \)).

Now we define algorithm \( A_1 \) which is identical to \( A \), except at time \( i \) \( A_1 \) removes \( q \) instead of \( p \) (mimicking \( LFD \)); and at time \( t \) \( A_1 \) removes \( p \) (assuming \( \sigma_i \neq p \); if \( \sigma_i = p \) then \( A_1 \) would do nothing at time \( t \) as there would be no page fault). Our analysis can now be divided into two cases:

- Case 1: if \( t < j \), then \( \text{COST}_{A_1}(\sigma) = \text{COST}_A(\sigma) \), both have a page fault at time \( j \) when requesting \( p \).
- Case 2: if \( j \leq t < l \), \( A \) faults at time \( t \) and removes \( q \), while \( A_1 \) does not incur a page fault. Thus after \( t \), \( \text{COST}_{A_1}(\sigma) \leq \text{COST}_A(\sigma) + 1 \), that is, \( A_1 \) suffers one fewer page faults than \( A \).

Thus, \( A_1 \) is either same or better performance-wise than \( A \), but the page fault sets of \( A_1 \) and \( LFD \) are identical at least to time \( t \) which is strictly greater than \( i \). This argument can be repeated iteratively to generate an algorithm \( A_{i+1} \) from \( A_i \), where \( A_{i+1} \) matches \( LFD \) for a strictly greater length of time than \( A_i \), \( \forall i \). Since the sequence has finite length, it must be the case that eventually, we will derive algorithm \( A_n \) which matches \( LFD \) for the entire sequence which has the same or better performance than \( A \). More precisely we have \( \text{COST}_{LFD}(\sigma) = \text{COST}_{A_n}(\sigma) \leq \text{COST}_{A_{n-1}}(\sigma) \leq \ldots \leq \text{COST}_{A_1}(\sigma) \leq \text{COST}_{LFD}(\sigma) \), which is a contradiction.

There does not exist an algorithm \( A \) which outperforms \( LFD \). \qed

**Theorem 2.2.2.** \( LRU \) is a \( k \)-competitive strategy.

**Proof.** Assume an input sequence with non-trivial quantity of distinct pages (\( k + 1 \) or more). First observe that both algorithms start with at least \( k \) successes (to fill up the initial \( k \) slots). We want to show that \( \text{COST}_{LRU}(\sigma) \leq k \cdot \text{COST}_{LFD}(\sigma) \).
Let $\langle \sigma \rangle = \sigma_1, \sigma_2, ..., \sigma_i, [\sigma_{i+1}, ..., \sigma_j], [\sigma_{j+1}...\sigma_m], ...,$

where $\sigma_i$ is the first page fault and $\sigma_j$ is page fault $k + 1$, and $\sigma_m$ is page fault $2k + 1$. We will refer to the time encapsulated within a bracket as a “phase”. Note that it is the final element within a bracket that is a page fault; the first element within a bracket is not necessarily a page fault.

Our goal is to show that $LFD$ makes at least one page fault per phase, as $LRU$ makes precisely $k$ page faults in each phase by our construction.

Since there are $k$ page faults per phase, one of two cases must hold:

- Case 1: $LRU$ faults twice on some page $p$ in the phase. $k + 1$ distinct pages are requested, including $p$, during the phase, since $k$ pages must be requested in between the two faults on page $p$ for $LRU$ to fault on it twice. At least $k + 1$ distinct pages requested implies $LFD$ must fault at least once, since $LFD$ cannot have all of these pages already in memory.

- Case 2: $LRU$ faults on $k$ distinct pages in the bracket. Let $p_1$ denote the last page fault before the request phase (in previous phase).
  - Case 2a: There exists a page fault on $p_1$ by $LRU$ at some time in the current phase. Consider $p_1$ and the bracket following it. We may then conclude that $k$ distinct pages were requested prior to the fault on $p_1$, otherwise $LRU$ would not have removed $p_1$ and would not have faulted on it. Thus including $p_1$, $k + 1$ distinct pages were requested in the phase, thus at least 1 page fault is incurred by $LFD$.
  - Case 2b: There does not exist any page fault by $LRU$ on $p_1$ at some time in the current phase, so we need $k-1$ distinct pages to max out the remaining $k-1$ slots ($p_1$ is in memory at the start of the phase and never faulted on, so it is occupying one page slot throughout the entire phase). Since we have requested $k$ distinct pages, and have only $k - 1$ spaces for them, we can be certain at least 1 page fault is incurred by $LFD$ within the phase.

Thus, $LRU$ is $k$-competitive, as $LFD$ faults at least once for every $k$ faults made by $LRU$. □
Theorem 2.2.3. No deterministic algorithm is better than $k$-competitive.

Claim 2.2.4. For any deterministic online algorithm $A$, there exists a sequence of requests $\langle \sigma \rangle$ such that $\text{COST}_A(\sigma)$ is arbitrarily large and $\text{COST}_A(\sigma) \geq k \cdot \text{COST}_{OPT}(\sigma)$, as long as there are at least $k+1$ pages in our universe.

Proof. Given a deterministic online algorithm $A$, we will construct a sequence to force $A$ to produce a solution that is at least $k$ times worse than the solution of $OPT$ for the same sequence.

- First, request pages $p_1$ through $p_k$.
- Request $p_{k+1}$, and denote the page $A$ will remove to make room for $p_{k+1}$ as $q$, which we can determine since $A$ is deterministic.
- Request $q$, and denote the page $A$ will remove to make room for $q$ as the 'new $q$' and repeat this step $k \cdot d$ times (for some integer $d$).

The sequence constructed above has length $j = k(d+1)$, and $A$ executing on this sequence yields $kd$ page faults, as the first $k$ requests are the only non-faults. Hence $\text{COST}_A(\sigma) = kd$.

If the optimal offline algorithm faults on some page $\sigma_i$, it will not fault on the next at least $k-1$ requests. Hence,

$$\text{COST}_{OPT}(\sigma) \leq j/k = (d+1) = (d+1) \cdot \frac{kd}{kd} = \frac{\text{COST}_A(\sigma)}{k} \cdot \frac{d+1}{d}$$

Multiply both sides by $k$ and taking the limit as $d \to \infty$ yields

$$k \cdot \text{COST}_{OPT}(\sigma_j) \leq \text{COST}_A(\sigma_j).$$

Therefore no deterministic algorithm can be better than $k$-competitive. \qed

In addition to providing a bound on the competitiveness of a deterministic online algorithm, this example demonstrates that an adversary could, knowing exactly how the algorithm works, construct a sequence of page requests which will force the algorithm to perform poorly.
3 Randomized Online Algorithms

In practice, this adversary is too strong, and it does not serve as a realistic performance metric. To weaken the adversary, we introduce randomness into the algorithm to give it an aspect that the adversary cannot predict.

**Definition 3.0.1.** A randomized online algorithm is a distribution of deterministic online algorithms $A_X$, where $X$ is a set of random event outcomes.

**Definition 3.0.2.** A randomized online algorithm $A$ is $\alpha$-competitive against an offline adversary if there exists a constant $C$ such that for all input sequences $\sigma$,

$$E[C_A(\sigma)] \leq \alpha C_{\text{min}}(\sigma) + C \quad (1)$$

where $C_A$ is the cost incurred by $A$, $C_{\text{min}}$ the minimum cost, and the expectation is over all possible coin flips for $A$ (not over the input sequences).

4 Randomized Marking Algorithm

Algorithm M:

1. Begin by initializing all pages as marked.

2. When page $p$ is requested:

   (a) If $p$ is not in memory:

      i. If all pages in memory are marked, unmark them all.

      ii. Swap $p$ into memory, and swap out a random, uniformly selected unmarked page.

   (b) Mark page $p$ in memory and serve it.

As a page gets marked once it is used, the mark serves as an indicator that a page was used more recently than the unmarked pages. In that sense, this algorithm is somewhat like LRU.

To help our analysis, we introduce the following definitions.
**Definition 4.0.1.** For any sequence $\sigma$ of input requests to $M$, we say that a request $\sigma_i$ is an *unmarking request* if all pages in memory are marked when $\sigma_i$ arrives, and $\sigma_i$ is a request for a page not already in memory.

We use the unmarking requests to divide $\langle \sigma \rangle$ into phases:

$$\sigma = \sigma_1, \sigma_2, \sigma_3, \ldots, [\sigma_i, \ldots], [\sigma_j, \ldots], \ldots$$

where $\sigma_i$ and $\sigma_j$ are unmarking requests.

For $i > 0$, define 0 phase $i$ to be the subsequence of $\sigma$ starting at the $i^{th}$ unmarking request, up to but not including the $(i+1)^{st}$ unmarking request (or to the end of the sequence, if there are exactly $i$ unmarking requests). For each phase $i$, we use $S_i$ to denote the set of pages that were in memory just before the start of phase $i$.

**Remark 4.0.2.** At the start of each phase, no pages in memory are marked. At the end of the phase, all $k$ pages are marked. Therefore the phase ends exactly when $k$ distinct pages are accessed, meaning that the phases only depend on the input sequence and are independent of the randomness of the algorithm.

**Remark 4.0.3.** Once a page has been accessed the first time in a phase, it will be marked and will remain marked during the phase. Marked pages are not thrown away until the end of the phase. Therefore, for each page, $M$ incurs a cost of at most 1 in a phase; if $M$ incurs a cost of 1, this cost is incurred the first time the page is requested.

**Remark 4.0.4.** Importantly, our definition of a phase depends only on the input sequence $\sigma$. The phases do not depend on the randomized choices the algorithm makes for swapping pages out. Furthermore, $S_i$ also depends only on the input sequence. $S_i$ will always consist of the previous $k$ distinct pages requested before phase $i$.

By the Remark 4.0.5, we need only look at the first-time requests of the pages to analyze the cost of a phase. We will divide such requests into two
types: those we always have to pay for, and those we might not have to pay for.

**Definition 4.0.5.** A first-time request for page $p$ in phase $i$ is a clean request if $p \notin S_i$. $M$ will have to pay a cost of 1 for such a request, no matter the results of the coin-tosses.

**Definition 4.0.6.** A first-time request for page $p$ in phase $i$ is a dirty request if $p \in S_i$. This means that we may not have to pay for $p$, since it’s already in memory at the start of the phase. But if $p$ was randomly evicted to make room for an earlier clean request, then we do incur a charge of 1.

Finding the expected cost of the dirty requests will help us analyze $M$.

### 4.1 Expected cost of a dirty request

Suppose $\sigma_j$ is a dirty request to page $p$ in phase $i$. Suppose that up to this point in the phase, there have been $s$ dirty requests and $c$ clean requests. We know that when $p$ is requested, the $c + s$ pages requested so far this phase must be in memory.

**Lemma 4.1.1.** The expected cost of the $s + 1$st dirty request in a phase is $\frac{c}{k - s}$, where $c$ is the number of preceding clean requests in this phase.

**Proof.** Use $L_i$ to denote the subset of $S_i$ of pages in memory that have not yet been requested. As $s$ pages of $S_i$ have been requested, and the clean requests, by definition, were not part of $S_i$ to begin with, $|L_i| = |S_i| - s = k - s$.

Let $U$ denote the set of $k - s - c$ pages left in memory that have not yet been requested this phase. By definition, $U \subseteq L_i$, and specifically $U$ consists of the pages of $L_i$, minus the ones that were randomly evicted for the clean requests. Since all members of $L_i$ have stayed unmarked from the start of the phase until now, they are all equally likely to have been evicted.

Let $f$ denote the probability that a page in $L_i$ is not in memory when $\sigma_j$ arrives. Equivalently, $(1 - f)$ is the probability it is in memory, and hence is in $U$. These are equal for all such pages. Hence, we have

$$k - s - c = |U| = E[|U|] = \sum_{q \in L_i} Pr(q \in U) = \sum_{q \in L_i} (1 - f) = (1 - f)(k - s) (2)$$
Solving \( k - s - c = (1 - f)(k - s) \) for \( f \) shows \( f = \frac{c}{k-s} \).

As \( f \) is the probability that any page in \( L_i \) is not in memory, it is in particular the probability that \( p \) is not in memory. In that case, we incur a charge of 1, and in the other case a charge of zero. Hence the expected cost for the first request to \( p \) is \( f = \frac{c}{k-s} \). \qed

4.2 Bounding the expected total cost of \( M \) on a phase

Now, we will bound the expected cost of the dirty request. Suppose there are \( l_i \) clean requests in phase \( i \). Since a phase has \( k \) requests, there must be \( k - l_i \) dirty requests.

Lemma 4.2.1. The expected cost of \( M \) on phase \( i \) of \( \sigma \) is no more than \( l_i H_k \) where \( H_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k} \) is the \( k \)th Harmonic number.

Proof. The number of clean requests before a given dirty request is bounded by \( l_i \), so by lemma 4.1.1, the expected cost of all dirty requests is bounded above by

\[
D = \frac{l_i}{k} + \frac{l_i}{k-1} + \frac{l_i}{k-2} + \ldots + \frac{l_i}{k-(k-l_i-1)}
\]

The total cost of all clean requests is simply \( l_i \), so the total expected cost is bounded above by

\[
D + l_i = l_i(1 + \frac{1}{l_i+1} + \frac{1}{l_i+2} + \ldots + \frac{1}{k}) \leq l_i H_k
\]

4.3 The competitive factor of \( M \)

Consider any offline algorithm \( A \). Let \( \sigma \) be any input sequence; we will compare \( A \)'s behavior to \( M \)'s on \( \sigma \). We divide \( \sigma \) into phases based on \( M \)'s algorithm. Let \( C_i(A) \) be the total cost \( A \) incurs during phase \( i \).
We then define a potential function $\phi_i$, which is the number of pages in $A$’s memory that are not in $M$’s memory, just before the start of phase $i$ (and $\phi_{n+1}$ the number at the end of the algorithm). This function is well defined because the pages in $M$’s memory at the start of a phase are deterministic, determined only by $\sigma$.

Using these, we establish the following bounds on the costs incurred by $A$ during a phase.

**Lemma 4.3.1.** $C_i(A) \geq l_i - \phi_i$

**Proof.** $M$ receives $l_i$ clean requests in phase $i$. These pages were not in $M$’s memory at the start of the phase. By definition of the potential function, at most $\phi_i$ of them will be in $A$’s memory. In other words, the amount that won’t be in $A$’s memory is at least $l_i - \phi_i$.

**Lemma 4.3.2.** $C_i(A) \geq \phi_i + 1$

**Proof.** At the end of phase $i$, $A$ has $\phi_{i+1}$ pages in memory that $M$ does not. Hence $M$ also has $\phi_{i+1}$ pages that $A$ does not. Call that set of pages $P_i$.

Note that every page in $M$’s memory was accessed at least once during the phase, so all pages in $P_i$ must have been accessed at some point in phase $i$. But since they were all evicted by $A$ before the end of the phase, $A$ made at least $|P_i| = \phi_{i+1}$ evictions in phase $i$.

The total cost of an algorithm is the sum of its costs over all phases. Combining the previous two lemmas, we obtain $C_i(A) \geq \frac{1}{2} (l_i - \phi_i + \phi_{i+1})$, which results in a telescoping series:

$$C(A) \geq \frac{1}{2} \left[ (l_1 - \phi_1 + \phi_2) + (l_2 - \phi_2 + \phi_3) + ... + (l_n - \phi_n + \phi_{n+1}) \right]$$

$$= \frac{1}{2} \left[ \sum_i l_i + (-\phi_1 + \phi_2 - \phi_2 + \phi_3 - \phi_3 + ... + \phi_n - \phi_n + \phi_{n+1}) \right]$$

$$= \frac{1}{2} \left[ \sum_i l_i - \phi_1 + \phi_{n+1} \right]$$

$$\geq \frac{1}{2} \sum_i l_i$$

(5)
Let $C(M)$ be the total expected cost of $M$. From the result of section 4.2, we see that $C(M) \leq H_k \sum_i l_i$.

So we have that $\sum_i l_i \leq 2C(A)$. Combining these two bounds shows that $C(M) \leq H_k \sum_i l_i \leq 2H_k C(A)$. Therefore $M$ is $2H_k$-competitive.

There exists a proof that no randomized algorithm can do better, but we won’t get into that.

5 Acknowledgement

These lecture notes are heavily adapted from the notes of several students, who took Lenore Cowen’s Comp 260 in previous years:

• Srikanth Ravi (2004)
• John Trafton (2009)
• Alex Tong (2016)
• Colin Hamilton (2016)
• Isaiah Mindich (2018)

The original lecture notes were adapted from the notes of a class taught by Michel Goemans at MIT in 1992.