CS264: Beyond Worst-Case Analysis
Lecture #3: Online Paging and Resource Augmentation[*]

Tim Roughgarden[†]

September 29, 2014

1 Preamble

This course covers many different methods of analyzing and comparing algorithms. Periodically, as in Section 2, we pause to review the "big picture" and suggest methods for keeping tracking of the course's main ideas, their goals, and the problems for which they are most likely to be useful. Section 3 begins our study of the online paging problem — introduced briefly in Lecture #1 — we'll also study this problem in the next lecture. We'll see that traditional "competitive analysis" fails to illuminate the problem in several respects: it does not give accurate performance predictions, it does not give good guidance on how to pick a cache size, and it gives only weak information about which caching policy to use. Section 6 covers "resource augmentation," an alternative (but still worst-case) method of analyzing online algorithms that gives more meaningful performance guarantees. The next lecture presents an analysis that more sharply differentiates between different paging algorithms by modeling structure in data.

2 The Big Picture

There is a strong analogy between the organization of this course and that of most undergraduate algorithms courses. In an undergrad course like CS161, the primary goal is to develop a toolbox for algorithm *design*.[1] You learn that there is no "silver bullet" — no single algorithmic idea will solve every computational problem that you'll ever encounter. There are, however, a handful of powerful design techniques that enjoy wide applicability: divide and conquer, greedy algorithms, dynamic programming, proper use of data structures,

[*]©2014, Tim Roughgarden.

[†]Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: `tim@cs.stanford.edu`.

[1]Some of the topics covered are for analysis purposes, like the vocabulary of asymptotic analysis, but these are the exceptions.

etc. It's a bit of an art to figure out which techniques are best suited for which problems, but through practice you can hone this skill. Finally, these algorithm design techniques are taught largely through representative — and typically famous and fundamental — problems and algorithms. This kills two birds with one stone: students both sharpen their ability to apply a given technique through repeated examples, and the examples are problems or algorithms that every card-carrying computer scientist would want to know, anyways. For example, divide-and-conquer is taught using the fundamental problem of sorting, and students learn famous algorithms like MergeSort and QuickSort. Similarly, greedy algorithms are taught using the minimum spanning tree problem, and the algorithms of Kruskal and Prim. Dynamic programming via shortest-path problems, and the algorithms of Bellman-Ford and Floyd-Warshall. And so on.

The goals and organization of CS264 are comparable in many respects; the primary difference is the high-level goal of supplying you with a toolbox for algorithm *analysis*, and specifically ways to rigorously compare different algorithms. We've already seen a few analysis approaches — traditional worst-case analysis, instance optimality, and parameterized analysis — and will see many more.Once again, no single way of analyzing algorithms is the "right way" for all computational problems, and it's not always clear which analysis framework is best suited for which problem. But the lectures and homework will introduce you to many such frameworks, each applied in several examples. And to maximize the value of these lectures, whenever possible we choose as examples problems and algorithms that are interesting in their own right, such as online paging, linear programming, and clustering.

3 Online Paging

Recall the online paging (a.k.a. online caching) problem from Lecture #1. This is a real-world problem, but also simple enough to showcase many different alternatives to worst-case analysis. This section introduces a formal model for analyzing online paging algorithms due to Sleator and Tarjan [5].[2]

- There is a slow memory with N pages.
- There is a fast memory (a *cache*) that can only hold $k < N$ of the pages at a time.
- Page requests arrive "online," one request per time step.[3]
- If the page request p_t at time t is already in the cache, zero cost is incurred.
- If p_t is not in the cache, it needs to be brought in; if the cache is full, one of its k pages must be evicted (without knowing what the future requests will be). One unit of cost

[2]A good general reference for this section and the next is [2, Chapter 3].

[3]This use of the word "online" grows more anachronistic with each year. When the field of "online algorithms" was invented in the 1980s [5], the word "online" did not have the connotations it does today — that was 10 years before the World Wide Web existed, for example.

is incurred in this case.[4]

In our running notation, we define $\text{cost}(A, z)$ as the number of "page faults" (a.k.a. "cache misses") that A incurs on the page request sequence z.

Recall that if we had clairvoyance and knew all future page requests, then we have a solid understanding of OPT, the caching policy that minimizes the number of page faults.

Theorem 3.1 (Belady's Theorem [1]) *The* Furthest-in-the-Future *algorithm, which always evicts (on a cache miss) the page that will be requested furthest in the future, always minimizes the number of page faults.*

The proof is a slightly tricky greedy exchange argument; see e.g. [2, Theorem 3.1].

Recall also that when future requests are unknown, the "gold standard" in practice is the *Least Recently Used (LRU)* policy, which on a cache miss evicts the page whose most recent request is as far back in the past as possible. Many empirical studies show that LRU performs very well on "typical" request sequences — not much worse than the offline optimal algorithm, and better than other obvious online algorithms like first-in first-out (FIFO) (e.g. [6, §2.4]). The usual explanation one learns is: "real data exhibits locality — with recent requests likely to be requested again soon — and LRU automatically adapts to and exploits this locality."

Thus in some sense we already know the "correct" algorithm — or at least a near-optimal one — in the form of LRU. But if LRU is the answer, what is the question?

4 Competitive Analysis

This section explains the dominant paradigm for comparing online algorithms: *competitive analysis.* The following definition is key.

Definition 4.1 (Competitive Ratio [5]) The *competitive ratio* of an online algorithm A is its worst-case performance relative to an optimal *offline* algorithm OPT, which has full knowledge of the page sequence z up front:

$$\max_z \frac{\text{cost}(A, z)}{\text{cost}(OPT, z)}.$$

The competitive ratio is always at least one; the closer to one the better.[5] In competitive analysis, we interpret online algorithm A as "better than" another one B if and only if it has a smaller competitive ratio. Similarly, in competitive analysis, "optimal" online algorithms are those with the smallest-possible competitive ratio.

[4]A more general model allows arbitrary changes to the cache at every time step, whether or not there is a hit or miss, with the cost incurred equal to the number of changes. We will focus on the stated model, which corresponds to "demand paging" algorithms.

[5]A detail: usually (but not always) one ignores additive terms in the competitive ratio. Equivalently, we think about infinite sequences of inputs z such that $\text{cost}(OPT, z) \to \infty$.

We can interpret the competitive ratio as a form of instance optimality (Lectures #1 and #2): if an online algorithm A has a competitive ratio of α, then it is instance optimal with optimality ratio α — that is, it has cost at most α times that of every (online or offline) algorithm on every input. Thus it is no surprise that small competitive ratios are relatively rare: this translates to instance optimality with the additional severe restriction that the designed algorithm A is online.

5 Some Basic Worst-Case Upper and Lower Bounds

So what happens if we apply competitive analysis to the online paging problem? What does it say about the LRU policy versus other online policies?

5.1 Goals of Algorithmic Analysis

Before answering these questions, let's briefly recall from Lecture #1 our three goals of defining a formal performance measure and analyzing algorithms mathematically. At the end of the section, we'll give competitive analysis a report card according to how well it achieves these goals.

1. **Explanation (or Prediction) Goal.** Explain or predict the empirical performance of algorithms.
2. **Comparison Goal.** Rank different algorithms according to their performance. Ideally, identify "optimal" algorithms.
3. **Design Goal.** Guide the development of new algorithms.

5.2 A Lower Bound for all Deterministic Algorithms

Since an upper bound on the competitive ratio is an even stronger assertion than instance optimality, we expect there to be non-trivial lower bounds. Thus, before analyzing any specific algorithm, let's figure out what's the best we could hope for. The following lower bound draws a "line in the sand" that applies to all deterministic paging algorithms.

Theorem 5.1 (Lower Bound for All Paging Algorithms [5]) *Every deterministic paging algorithm has competitive ratio at least k.*

Proof: Take $N = k + 1$ and fix a deterministic online algorithm A. Since there is always some page missing from A's cache, one can define inductively a sequence σ so that A faults on every single page request. The furthest-in-the-future (FIF) algorithm, whenever it incurs a page fault on the sequence σ, has k candidates for eviction, and one of these will not be among the next $k - 1$ requests. Thus the FIF algorithm follows every cache miss with at least $k-1$ cache hits. The competitive ratio of A is therefore at least $|\sigma|/(|\sigma|/k) = k$, which proves the theorem. ■

Remark 5.2 (Randomized Paging Algorithms) Theorem 5.1 is stated only for deterministic paging algorithms, so you're probably wondering about randomized ones. There are some pretty cool randomized online paging algorithms, and the competitive ratios are much better: $O(\log k)$ is achievable and also the best possible. See [2, Chapter 4] for a survey. We won't discuss these randomized algorithms, as most of them are rather different from the standard ones used in practice, and because the online paging problem seems solvable empirically (e.g., by LRU) without resorting to randomization.[6]

5.3 An Upper Bound for LRU

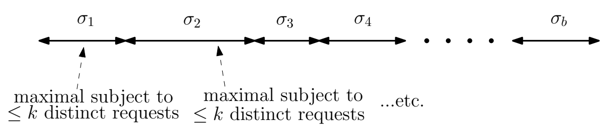
The bad news is that the lower bound in Theorem 5.1 is laughably huge — k is pretty big in most systems — far worse than the small percentage error one observes for reasonable paging algorithms (LRU, FIFO, etc.) on "real data". This rules out taking competitive ratios literally as performance predictions. Equally disturbingly, the competitive ratio of every paging algorithm increases linearly with the cache size. Looking at the proof of Theorem 5.1, we see a particularly transparent example of the "Murphy's Law" data model — no matter what the cache size k is, the input can be chosen so that the online paging algorithm faults at every time step (meanwhile the offline optimal algorithm's page fault rate is decreasing with k). This suggests that devoting resources (e.g., transistors on a microchip) to bigger caches is pointless and that they would be better spent anywhere else — this advice is obviously misleading and inconsistent with practice.

But if we recall how stringent the definition of a competitive ratio is — again, even more so than instance optimality — we shouldn't be surprised to see such high competitive ratios. *And this does not necessarily mean that the theoretical results are meaningless or useless.* Hope remains that competitive analysis can achieve the Comparison Goal — accurate ordinal information about which online paging algorithms are the best. The following theorem is good news along these lines.

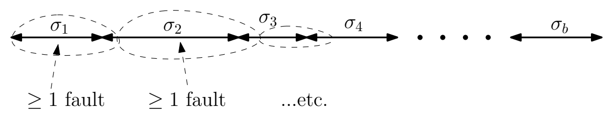
Theorem 5.3 (Upper Bound for LRU [5]) *The competitive ratio of the LRU algorithm for online paging is at most k, the size of the cache.*

Proof: Consider an arbitrary request sequence σ. We need to prove both an upper bound on the number of faults that LRU incurs, and a lower bound on the number of faults incurred by the optimal offline algorithm. A useful idea for accomplishing both goals is to break σ into *blocks* $\sigma_1, \sigma_2, \ldots, \sigma_b$. Here σ_1 is the maximal prefix of σ in which only k distinct pages

[6]A caveat: worst-case analysis of randomized algorithms sometimes helps explain why deterministic algorithms perform well on "real-world instances" of a problem. For example, the proof that randomized Quicksort is fast (with high probability) on every input also shows that deterministic Quicksort is fast on almost every input. Thus the worst-case analysis of randomized QuickSort provides an excellent theoretical explanation of the speed of deterministic QuickSort on "real data", for essentially any reasonable definition of "real data" (modulo the usual "already sorted" exception). It is not clear that the randomized online paging algorithms with good competitive ratios are similar enough to algorithms like LRU to warrant a similar interpretation.


σ_1
σ_2
σ_3
σ_4
σ_b
maximal subject to $\leq k$ distinct requests
maximal subject to $\leq k$ distinct requests
...etc.

(a) Blocks of a Request Sequence



(b) Page Fault Lower Bound

Figure 1: Proof of Theorem 5.3. In (a), the blocks of a page request sequence; the LRU algorithm incurs at most k page faults in each. In (b), the optimal offline algorithm incurs at least one page fault in each "shifted block".

are requested; the block σ_2 starts immediately after and is maximal subject to only k distinct pages being requested (ignoring what was requested in σ_1); and so on.

The first important point is that LRU faults at most k times within a single block — at most once per page requested in the block. The reason is that once a page is brought into the cache, it won't be evicted until k other distinct pages get referenced, which can't happen until the following block. Thus LRU incurs at most bk page faults, where b is the number of blocks. See Figure 1(a).

Second, we claim that an optimal offline algorithm must incur at least $b-1$ page faults. To see this, consider the first block plus the first request of the second block. Since σ_1 is maximal, this represents requests for $k+1$ distinct pages, and no algorithm can serve them all without a page fault. Similarly, suppose the first request of σ_2 is the page p. After an algorithm serves the request for p, the cache contains only $k-1$ pages other than p. But by maximality of σ_2, the rest of σ_2 and the first request of σ_3 contain requests for k distinct pages other than p; these cannot all be served without incurring another page fault. And so on, resulting in at least $b-1$ cache misses. See Figure 1(b). This upper bound $bk/(b-1)$ on the competitive ratio approaches k as $b \to \infty$, so this completes the theorem. ■

5.4 An Upper Bound for Flush-When-Full

Theorems 5.1 and 5.3 show that LRU has the smallest-possible competitive ratio among the class of all deterministic online paging algorithms. (While a silly algorithm like LIFO — "last-in, first out" — does not, as can be seen via a sequence that alternates requests between two different pages.) This is definitely a point in favor of competitive analysis.

But the plot thickens if we consider the Flush-When-Full (FWF) algorithm.

Example 5.4 (Flush-When-Full (FWF)) The Flush-When-Full algorithm works as follows. When the cache is full and a page fault is incurred, the FWF algorithm evicts *its entire cache.*[7] Note that, in the notation of the proof of Theorem 5.3, the cache flushes correspond precisely to the ends of the blocks $\sigma_1, \sigma_2, \ldots, \sigma_{b-1}$, with the FWF algorithm suffering exactly k faults (the mass eviction) per block.

[7]Strictly speaking this is outside our demand paging model, but you get the idea.

The FWF algorithm is "obviously" worse than LRU, in the sense that it never suffers fewer page faults, and suffers strictly more faults on most inputs. Nonetheless, *the competitive ratio of the FWF algorithm is also k.* The reason is that, since the FWF algorithm faults exactly k times per block, the proof of Theorem 5.3 still applies verbatim. Indeed, this algorithm could only have been discovered by worst-case analysts asking the question: what are the minimal properties required for the proof of Theorem 5.3 to work?

5.5 The Report Card

Recall the three goals described at the beginning of the section. For paging problems, competitive analysis does not fare particularly well. For the Explanation Goal, we already discussed at length how the overly pessimistic guarantees inevitable in competitive analysis cannot be taken literally. Competitive analysis earns a middling grade on the Comparison Goal — it successfully identifies LRU as an optimal online algorithm, but it also identifies as optimal inferior algorithms like FWF. We're not exploring the Design Goal here — for the online paging problem, we already have a good idea about the "right" way to solve the problem — but in general the clean framework of competitive analysis has been extremely successful in spurring new ideas for online algorithms for lots of different problems.

In the rest of this and next lecture, we explore a number of alternative approaches to analyzing online paging algorithms, with the aim of better achieving the Explanation and Comparison Goals.

6 Resource Augmentation and Interpretations

This section introduces *resource augmentation*, where the idea is to compare a protagonist (like LRU) endowed with "extra resources" to an all-powerful opponent that is handicapped by "less resources." Other than this twist, we use the standard (worst-case) competitive analysis framework. Naturally, weakening the abilities of the offline optimal algorithm can only lead to better competitive ratios. We can draw an analogy with our relaxed definition of instance optimality in Lectures #1 and #2, where we compared the performance of our algorithm to that of a subset of algorithms (on each input), rather than to all other algorithms. There, we restricted attention to "natural" algorithms; here, to algorithms required to make use of a smaller cache. While this is probably not the first restriction you would think of, it's a useful one.

Resource augmentation was perhaps first used by Sleator and Tarjan [5] for the online paging problem,[8] and its benefits are immediately apparent from the arguments we developed in the previous section. Recall the main steps in the proof of Theorem 5.3: after breaking the input σ into blocks $\sigma_1, \ldots, \sigma_b$, we argued that:

(1) LRU incurs at most k faults per block (at most once per distinct requested page, since LRU won't evict a requested page until the next block).

[8]The actual phrase "resource augmentation" is from [4]; see also [3].

(2) OPT has at least one fault per "shifted block": if p is the first request of a block σ_i, then afterward the cache of OPT has p and $k-1$ pages other than p, while the rest of the block σ_i together with the first request of block σ_{i+1} include k requests for distinct pages other than p.

Looking at the argument for (2), we see that a more general statement holds:

(2') If OPT is restricted to a cache of size $h \leq k$, then it incurs at least

$$\underbrace{k}_{\text{requests other than } p} - \underbrace{(h-1)}_{\text{pages in cache other than } p}$$

page faults per shifted block. We conclude the following.

Theorem 6.1 (Resource Augmentation Bound for LRU [5]) *The competitive ratio of the LRU algorithm with cache size k is at most*

$$\frac{k}{k-h+1}$$

with respect to the optimal offline algorithm with a cache of size $h \leq k$.

For example, if LRU has roughly double the cache size of OPT, then it is *2-competitive.* This guarantee is much more interesting from the perspective of our Prediction Goal than the competitive ratio of k proved in Theorems 5.1 and 5.3.[9]

7 Interpretations

Some obvious questions are: what does the guarantee in Theorem 6.1 mean? Should you be impressed by a resource augmentation guarantee like this? The concern is that the comparisons is "apples vs. oranges" — sure the optimal offline algorithm is powerful in that it knows all of the future requests, but it's artificially hobbled by a small cache.

7.1 A Two-Step Approach to System Design

The first justification is not mathematical but nevertheless interesting and useful. If we adopt the philosophy that the point of rigorous guarantees for algorithms is to give good advice about how to solve problems and build systems, then resource augmentation bounds offer a compelling two-step approach.

[9]We're still not getting sharp predictions of performance, of course, but at least we're now in the right ballpark. One can't expect sharp predictions without modeling "real-world" inputs in some way.

1. The first step is to estimate the amount of resources (e.g., cache size) that guarantees acceptable performance (e.g., page fault rate below a given target) assuming an optimal algorithm.[10] It is presumably simpler to solve this problem than to reason simultaneously about the cache size and paging algorithm design decisions.

2. The second step is to scale up the resources to realize the resource augmentation bound — for example, to double the cache size and invoke Theorem 6.1 to guarantee acceptable performance under (say) the LRU algorithm.

7.2 Loosely Competitive Online Algorithms

The second interpretation of Theorem 6.1 is mathematical. Young [7] proved (a generalization of) the following guarantee, which we state informally now and make precise in due time.

Informal Theorem [7]: *For every request sequence σ, the LRU algorithm is "provably excellent" on σ* for most cache sizes k.

In effect, Young's result shows that a resource augmentation guarantee like Theorem 6.1 — an apples vs. oranges comparison between an online algorithm with a big cache and an offline algorithm with a small cache — has interesting implications for online algorithms even compared with offline algorithms with the same cache size. Now, in light of Theorem 5.1, you should be asking, "what's the catch?" Young's result dodges the lower bound in that theorem by permitting two quite reasonable relaxations. The first is obvious from the informal theorem statement above: the guarantee holds only for "most" cache sizes, and LRU might perform poorly for a few cache sizes. This is a reasonable relaxation because only die-hard disciples of the "Murphy's Law" principle would expect "real data" to be adversarially tailored to the cache size.[11] The second relaxation is to allow "provably excellent performance" to mean one of two things — either good performance relative to the optimal offline algorithm (as usual), or good performance in the absolute sense of a small page fault rate. This result is another way to phrase an performance guarantee for the LRU algorithm that is much more meaningful for our Prediction Goal than the competitive ratio of k proved in Theorems 5.1 and 5.3.

There is simple and accurate intuition behind Young's result. Consider a request sequence σ and a cache size k. One case is that the number of page faults of LRU is roughly the same (within a factor of 2, say) with the cache sizes k and $2k$. In that case, Theorem 6.1 immediately implies that LRU has a good competitive ratio (in the traditional sense) when the cache size is k. In the second case, LRU's performance is improving rapidly as one supplements the cache with extra pages. But since there is a bound on the maximum fluctuation

[10]Remember: approximating the "optimal algorithm" is only meaningful when the performance of the optimal algorithm is good in some absolute sense!

[11]Typically it will be independent of the cache size. In highly optimized applications the request sequence might depend on the cache size, but in a helpful, rather than harmful, way.

of LRU's performance (between no page faults and faulting every time step), its performance can only change rapidly for a bounded number of different cache sizes.

More precisely, fix a request sequence σ and let b be a parameter. Let $\text{cost}(A, k, \sigma)$ denote the number of page faults incurred by the paging algorithm A with a cache size of k on the page sequence σ. Theorem 6.1 and the previous paragraph imply that, for every cache size k, either:

$$\text{cost}(LRU, k+b, \sigma) < \frac{1}{2} \cdot \text{cost}(LRU, k, \sigma) \tag{1}$$

or

$$\text{cost}(LRU, k, \sigma) \leq 2 \cdot \frac{k+b}{b+1} \cdot \text{cost}(OPT, k, \sigma), \tag{2}$$

where we are invoking Theorem 6.1 with $k+b$ and k playing the roles of k and h, respectively.

Call a cache size k *bad* if (1) holds. Consider the set of bad cache sizes; for every such size, adding b extra pages to the cache decreases the cost of LRU on σ by at least a factor of 2. If there are at least ℓ bad cache sizes between 1 and $k-b$ for some k, then we can find at least ℓ/b bad cache sizes in this interval that are each at least b apart (just take every bth bad cache size). In this case, we have

$$\text{cost}(LRU, k, \sigma) < 2^{-\ell/b} \cdot \text{cost}(LRU, 1, \sigma). \tag{3}$$

(We are also using the fact that $\text{cost}(LRU, t, \sigma)$ is nonincreasing in t — see Homework #2.)

Thus, once

$$\ell \geq b \cdot \log_2 \tfrac{1}{\epsilon}, \tag{4}$$

we have

$$\text{cost}(LRU, k, \sigma) \leq \epsilon \cdot |\sigma|,$$

where $|\sigma|$ is the length of the request sequence σ. Young [7] makes the compelling argument that if ϵ is sufficiently small (less than the access time to fast memory, say), then LRU's performance is superb in an absolute sense and we could care less about its competitive ratio.[12]

Here is the precise statement of Young's result.

Theorem 7.1 (LRU is Loosely Competitive [7]) *For every $\epsilon, \delta > 0$ and positive integer n, for every request sequence σ, for all but a δ fraction of the cache sizes k in $\{1, 2, \ldots, n\}$, the LRU algorithm satisfies either:*

(1) cost$(LRU, k, \sigma) = O(\frac{1}{\delta} \log \frac{1}{\epsilon}) \cdot$ cost(OPT, k, σ); or

(2) cost$(LRU, k, \sigma) \leq \epsilon \cdot |\sigma|$.

[12]While this may seem like an obvious point, such appeals to good absolute performance are underutilized in theoretical research on algorithms.

Theorem 7.1 says that, for every request sequence σ, every cache size k falls into one of three cases. In the first case, LRU with cache size k is competitive with OPT in the usual (non-resource augmentation) sense. In the second case, LRU has excellent performance in an absolute sense. In the third case neither of these two goods events occurs, but fortunately this happens for only a δ fraction of the possible cache sizes.

We have essentially already proved Theorem 7.1. We want δn bad cache sizes between 1 and some number t to force the condition that $\text{cost}(LRU, k, \sigma) \leq \epsilon|\sigma|$ for all cache sizes $k \geq t$, so we take $\ell = \delta n$; using (4), this forces $b = \delta n / \log_2 \epsilon^{-1}$. Then, for all but $\ell = \delta n$ cache sizes k, either $\text{cost}(LRU, k, \sigma) \leq \epsilon \cdot |\sigma|$ or (by (2)) the LRU algorithm has competitive ratio

$$\frac{2(k+b)}{b+1} \leq \frac{2(n+b)}{b+1} = \Theta\left(\frac{1}{\delta}\log\frac{1}{\epsilon}\right).$$

In [7] this guarantee is phrased as: LRU is (ϵ, δ)*-loosely* $O(\frac{1}{\delta}\log\frac{1}{\epsilon})$*-competitive.*

Note that the parameters δ, ϵ, and n of Theorem 7.1 are for the analysis only — no "tuning" of the LRU algorithm is needed — and Theorem 7.1 holds simultaneously for all choices of these parameters. The larger the fraction δ of bad cache sizes or the absolute performance bound ϵ that can be tolerated, the better the relative performance guarantee in case (1).

8 Key Take-Aways

Resource augmentation is a useful analysis tool for problems where it makes sense — problems parameterized by a resource like space, capacity, processor speed, and so on. For the online paging problem, it yields interpretable performance guarantees, such as the LRU algorithm with double the cache size having at most twice as many faults as the offline optimal algorithm (with the original cache size). Unlike traditional competitive analysis, this approach illuminates the benefit of larger cache sizes and offers a clean two-step approach to system design (first size a system for the optimal offline solution, then double the size). Since [5], the idea has been applied successfully in a range of contexts, especially in scheduling problems (see e.g. [3]).

Loose competitiveness (Theorem 7.1) translates the "apples vs. oranges" guarantee offered by resource augmentation (Theorem 6.1) into a guarantee for LRU vs. the offline optimal algorithm with the same cache size, at the expense of two relaxations: allowing a constant fraction of the cache sizes k to yield bad performance, and for the other cache sizes accepting either good relative performance or good absolute performance as an acceptable outcome.

Both our resource augmentation and loose competitiveness guarantees are for worst-case inputs. This input-by-input guarantee can clearly be viewed as a feature. It can also be viewed as a bug, however: LRU is the paging algorithm of choice in practice *because of properties of "real" data* — if we work in a model that cannot articulate such properties, then we cannot expect to separate LRU from other reasonable paging algorithms like FIFO (see Homework #2). Thus, neither of these two guarantees offer progress over traditional competitive analysis on our Comparison Goal. We tackle this issue head-on in the next lecture, where we parameterize inputs according to the "degree of locality," and use this parameter to show rigorous senses in which the LRU algorithm is strictly better than other algorithms, including FIFO.

References

[1] L. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5(2):78–101, 1967.

[2] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[3] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47(4):617–643, 2000. Preliminary version in *FOCS '95*.

[4] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32(2):163–200, 2002. Preliminary version in *STOC '97*.

[5] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commincations of the ACM*, 28(2):202–208, 1985. Preliminary version in *STOC '84*.

[6] N. E. Young. *Competitive Paging and Dual-Guided Algorithms for Weighted Caching and Matching*. PhD thesis, Princeton University, Department of Computer Science, 1991.

[7] N. E. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.