## Offline and Online Optimization Problems

An **optimization problem** can be a problem of cost minimization of profit maximization.

An optimization problem $\mathcal{P}$ of cost minimization consists of a set $\mathcal{I}$ of inputs and a cost function $C$. Associated with every input $I$ is a set of feasible outputs (or solutions) $F(I)$, and associated with each feasible solution $O$ in $F(I)$ is a positive real $C(I, O)$, representing the cost of the output $O$ with respect to the input $I$.

There are analogous concepts for profit maximization.

For example, consider the bin packing problem. This problem presents an unbounded number of uniform bins, each having some fixed size and capacity, say 1. An input is a sequence of items:

$$x_1, x_2, \ldots, x_n$$

where $x_i$ represents the size of the $i$-th item. All item sizes satisfy $0 < x_i \leq 1$. The goal is to "pack" all items into bins in the most compact way.



Given an input, a feasible solution is any assignment of all the input items $x_i$ such that the sum of the item sizes assigned to any bin does not exceed the bin capacity (which is 1).

The cost of a (feasible) solution is the number of bins used to pack all the items. Originally, this bin packing problem was studied as an "offline problem." That is, an algorithm is allowed to consider the entire list of items in order to compute the best solution.

In order to view bin packing as an "online problem,":

- each item $x_i$ must be assigned without knowledge of items $x_{i+1}, \ldots, x_n$,
- the assignment of items $x_1, \ldots, x_i$ must be a feasible solution for each $i$,
- when we pack $x_i$, we are not allowed to alter the bin assignment of previous packed items.

Thus, an online feasible solution is a sequence of assignments. Exactly as in the offline problem, the total cost of an online solution is the total number of bins used.

Equivalently, we can consider the cost of each online assignment of one item to be either 0 or 1, depending on whether or not the item assignment opened a new bin (during the online assignment) and the total cost becomes the sum of the individual item assignment costs.

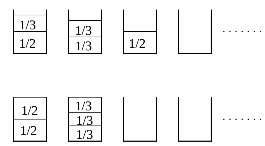


Suppose that we have the following sequence of weights:

$$1/2, 1/3, 1/3, 1/3, 1/2$$

The figure below shows the result of packing these weights into bins according to two strategies: the first (naive), packs the bins with the weights as they come, and uses three bins.

The second strategy, waits until it sees all the weights, and then packs them into the bins, in a clever way, to minimize space, and it uses only two bins.




The point of this example is that an online bin-packing algorithm is at a disadvantage, since it cannot wait and see all the weights, and then pack them, and naive bin packing does not yield an optimal solution (in terms of bins used).



Given any legal input $I$, an algorithm ALG for an optimization problem $\mathcal{P}$ computes a feasible (solution) $\mathsf{ALG}[I] \in F(I)$. The cost associated with this feasible output is denoted by $\mathsf{ALG}(I) = C(I, \mathsf{ALG}[I])$. An **optimal algorithm** OPT is such that for all legal inputs,

$$\mathsf{OPT}(I) = \min_{O \in F(I)} C(I, O)$$

An algorithm ALG is a $c$-**approximation algorithm** for a minimization problem $\mathcal{P}$ if there is a constant $\alpha \geq 0$ such that for all legal inputs:

$$\mathsf{ALG}(I) - c \cdot \mathsf{OPT}(I) \leq \alpha$$

Optimization problems in which the input is received in an online manner and in which the output must be produced online are called **online problems**.

In some problems, such as jobs scheduling and bin packing, both the online and offline versions of the problem are naturally meaningful.

However, many problems such as:

- paging
- telephone circuit switching
- investment planning
- CPU scheduling

are online problems. In particular, the two big problems in Operating Systems: paging and scheduling, are intrinsically online, and offline algorithms for them are not acceptable.

Still, even for intrinsically online problems, we could hypothesize an offline algorithm (that must have clairvoyant abilities) for the sake of comparative analysis. For example, we use the OPT paging algorithm as a paradigm for page-replacement algorithms.



**The Competitive Ratio and Competitiveness**

An online algorithm ALG is **$c$-competitive** if there is a constant $\alpha$ such that for all finite input sequences $I$,

$$\text{ALG}(I) - c \cdot \text{OPT}(I) \leq \alpha$$

When the **additive constant** $\alpha$ is less than or equal to zero (i.e., $\text{ALG}(I) \leq c \cdot \text{OPT}(I)$) we may say for emphasis that ALG is **strictly $c$-competitive**.

Allowing a positive constant $\alpha$ reflects the view that for intrinsic online problems such as paging we have an arbitrary long input sequence with unbounded cost. The constant $\alpha$ becomes insignificant as we consider longer and longer (and more costly) initial subsequences.

A (strictly) $c$-competitive online algorithm ALG is a $c$-approximation algorithm with the restriction that ALG must compute online. Thus, for each input $I$, a $c$-competitive algorithm is guaranteed to incur a cost within a factor $c$ of the optimal offline cost.

Note that the competitive ratio is always at least 1, and the smaller it is, the better ALG performs with respect to OPT. If ALG is $c$-competitive, we say that ALG attains a **competitive ratio** $c$. An algorithm is **competitive** if a "$c$" can be found for it.



Although $c$ may be a function of the problem parameters, it must be independent of the input $I$. For example, in a scheduling problem with $N$ machines, we might have a competitive ratio $c$ that depends on $N$, but $c$ must be independent of the number and type of jobs being scheduled.

The infimum over the set of all values $c$ such that ALG is $c$-competitive is called the **competitive ratio** of ALG, and is denoted by $\mathcal{R}(\text{ALG})$.

ASIDE: One way to analyze online algorithms is to view the problem as a game between an **online player** and a malicious **adversary**.

The online player runs an online algorithm on an input that is created by the adversary.

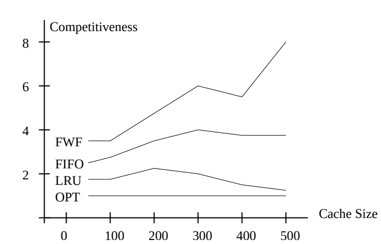
The adversary, based on the knowledge of the algorithm used by the online player, constructs the worst possible input so as to maximize the competitive ratio. That is, the adversary tries to make the task costly to the online player but, at the same time, inexpensive for the optimal offline algorithm.

We sometimes identify the adversary and the optimal offline algorithm as one entity: the **offline player**.



**Experimental Evidence**

The following figure summarizes the results of experimental evidence. It graphs the "competitiveness" of various paging algorithms as a function of the cache size with respect to a particular input sequence consisting of 692,057 requests to 642 distinct pages of 1024 bytes each. The sequence was generated by two "X windows" network processes, a "makefile" (program compilation), and a "disk copy" running concurrently. The requests include data reads and writes and instruction fetches.




The question now is: will our competitive analysis reflect these empirical findings?

Given the importance of memory hierarchies, paging (caching) continues to be a very active research area, and we need methods for measuring performance of page-replacement algorithms.

## LRU (Section 10.4.4)

The optimal (infeasible) algorithm for page replacement, OPT, evicts the page whose next request is the latest (and if some pages are never requested again, then one of them must be evicted, and the choice of which one is unimportant).

If we use the recent past as an approximation of the near future, then we will replace the page that:

*has not been used for the longest period of time*

This approach is the **Least Recently Used (LRU)** algorithm.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time.

The LRU algorithm is considered to be good, and is often implemented. The major problem is *how* to implement it.



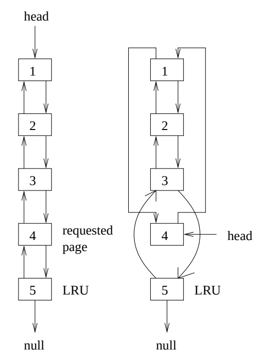
Two implementations are feasible:

- **Counters:** Keep track of the time when a given page was last referenced, updating the counter every time we request it. This scheme requires a search of the page table to find the LRU page, and a write to memory for each request. (Problems: clock overflow, and others).

- **Stack:** Keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the top of the stack is always the most recently used page, and the bottom is the LRU page. Because entries are removed from the middle of the stack, it is best implemented by a:

  Doubly-linked List






How many pointer operations have to be performed in the above example? In class we said that 6, where we counted them as follows:
-remove old head and add new head: 2 operations
-connect 4 with 1: 2 operations
-connect 3 with 5: 2 operations

However, we could have also counted disconnecting 3 with 4 and 4 with 5, giving 4 more pointer operations, giving us a total of 10. A third strategy would be not to count disconnecting pointers, in which case we would get half of these operations, 5. It does not really matter how we count, because the point is that in order to move a requested page (after a hit) to the top, we require a small constant number of pointer operations, regardless of how we count them.



### Exercises

(Not to hand in, but a good preparation for the test.)

1. List the pointer operations that have to be performed if the requested page is *not* in the cache. Note that you should list the pointer operations, not count them, since we just showed that there are three different (all reasonable) ways to count them. Again, the point is, that if a page has to be brought in from the slow memory to the cache, a small constant number of pointer operations have to be performed.

2. We are using a doubly-linked list to keep track of the LRU page. Explain what would go wrong if we used a normal linked list, that is, every page has a pointer to the next page ($i \longrightarrow j$, meaning that $i$ was requested more recently than $j$, but no page was requested later than $i$ and sooner than $j$). Is there is simple fix to the linked list that would make it work?