Online Algorithms

lecture notes
incomplete

The aim of the course is to give an introduction into the competitive analysis of the so-called "online" (also spelled "on-line") algorithms. In the online setting, the algorithm gets its input piecewise (the meaning of "piecewise" varies and is problem-specific), and to each piece it has to make an irrevocable decision before getting the next one. Each such decision sequence has an associated (nonnegative real-valued) cost which the algorithm tries to minimize. Clearly, such an algorithm cannot always produce an optimal decision sequence – the competitiveness of the online algorithm is a quantity c such that the sequence generated by the algorithm has a cost which is at most c times larger than that of the optimal sequence.

A bit more formally, an input of a problem is a sequence usually denoted $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_n)$ of requests σ_i, coming from a set. To each such $\boldsymbol{\sigma}$ there is the set $S(\boldsymbol{\sigma})$ of solutions, each having the form $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_n)$, that is, the request σ_i is served by the action (or the response) τ_i. Each solution $\boldsymbol{\tau}$ has some nonnegative real cost $C(\boldsymbol{\tau})$.

The offline optimum cost $\mathrm{Opt}(\boldsymbol{\sigma})$ of the input $\boldsymbol{\sigma}$ is the value $\min\{C(\boldsymbol{\tau}) : \boldsymbol{\tau} \in S(\boldsymbol{\sigma})\}$.

A (deterministic) online algorithm A cannot see the future and has to compute τ_{i+1} from $(\sigma_1, \ldots, \sigma_{i+1})$, that is, $\tau_{i+1} = A(\sigma_1, \ldots, \sigma_{i+1})$, in a way that if $\boldsymbol{A}(\boldsymbol{\sigma}) = (A(\sigma_1), A(\sigma_1, \sigma_2), \ldots, A(\boldsymbol{\sigma}))$ denotes the action sequence generated by A for the input $\boldsymbol{\sigma}$, then $\boldsymbol{A}(\boldsymbol{\sigma}) \in S(\boldsymbol{\sigma})$ has to hold (that is, A always should generate a valid solution). Then, the cost of the solution produced by A is $C(A(\boldsymbol{\sigma}))$. The algorithm A is called c-competitive for some quantity c if for each possible input sequence $\boldsymbol{\sigma}$ we have $\frac{C(\boldsymbol{A}(\boldsymbol{\sigma}))}{\mathrm{Opt}(\boldsymbol{\sigma})} \leq c$.

Let us see two examples.

The ski rental problem

The base form of the ski rental problem[1] is the following. We are spending our (presumably winter) holiday of unknown length in a ski resort. In the beginning of each day, we have to decide whether we rent a pair of skis (this action is denoted R) for a unit cost, or buy them (this action is denoted B) for an integer cost of $B > 1$. If we already bought a pair of skis during the season, we can simply use them for free (this action is denoted S, as "skip").

In the formal setting given above, an input is a sequence $\boldsymbol{\sigma} = (1, 1, 1, \ldots, 1)$ of some length N, the number of days our holiday lasts for. Usually we say in this case that the (input) season has length N. For such an input, a solution is either the sequence $(\mathrm{R}, \ldots, \mathrm{R})$ of length N, having the cost N (this corresponds to the case when we rent the skis during the whole season), or a sequence of the form $(\mathrm{R}, \ldots, \mathrm{R}, \mathrm{B}, \mathrm{S}, \ldots, \mathrm{S})$ of length N (this corresponds to the case when we rent the skis for a number of days, then eventually we buy them and after that, we only use them freely). If we buy the skis on the ith day, then the cost of this solution is $i - 1 + B$ (we rent for $i - 1$ days for a unit cost, then finally buy the skis for an additional cost of B). Although the definition of the problem would allow additional rents or even purchases after the very first purchase, it is clear that no sane algorithm would do that.

Determining the offline cost is easy for this problem: if the season lasts for at most B days, then the best option is to rent the skis during the whole season. Otherwise, the best option

[1] okay, I'm stopping using colors till I finish the draft for the whole semester. Eventually the lecture notes will be more colorful

is to buy the skis on the very first day and use them during the whole season. Formally, the reader can verify that $\mathrm{Opt} = \min\{N, B\}$ (N always denotes the length of the season).

Le us see an example for an online approach for the case when $B = 4$. A possible algorithm is the following: on the first two days of the season, let us rent the skies and on the third day (if there is a third day at all), let us buy them and continue to use them from that point. Formally, if we denote this algorithm by A_3, then $A_3(1) = A_3(1,1) = \mathrm{R}$, $A_3(1,1,1) = \mathrm{B}$ and $A_3(1,1,\ldots,1) = \mathrm{S}$ whenever the length of the input sequence is already larger than 3. Clearly, this is an online algorithm according to the definition above. Now analysing the behaviour of A_3 we get the following:

- If $N = 1$, then $\mathrm{Opt} = 1$ and the cost of A_3 is also 1 (it rents the skis on the first day). The ratio $C(A_3)/\mathrm{Opt}$ is 1 (meaning we produce an optimal solution in this case).
- If $N = 2$, then $\mathrm{Opt} = 2$, as well as the cost of A_3, making the ratio again 1.
- If $N = 3$, then $\mathrm{Opt} = 3$. Now A_3 decides to buy the skis on the third day, thus its cost becomes 6 ($1 + 1$ for renting and 4 for the eventual purchase) and the ratio is $6/3 = 2$.
- If $N \geq 4$, then $\mathrm{Opt} = \min\{N, B\} = 4$ and the cost of A_3 remains 6, making the ratio to be $6/4 = 1.5$.

Amongst these, 2 is the worst (i.e. largest) possible ratio, thus A_3 is a 2-competitive online algorithm for the ski rental problem when $B = 4$.

If instead, we buy on the fourth day (and let us denote the resulting algorithm by A_4), then we get a ratio of 1 for the cases $N < 4$, and a ratio of $7/4$ for the cases $N \geq 4$, making A_4 a $7/4$-competitive algorithm. Since $7/4 < 2$, we say that A_4 is a better algorithm than A_3 for this problem (with $B = 4$).

Since the structure of the input is very restricted, only a sequence of ones, we can see that all the (deterministic) online algorithms for the ski rental problem have the following form:

- either the algorithm rents the skis for $i - 1$ days for some constant i, then buys them on the ith day – this algorithm is denoted A_i;
- or the algorithm rents the skis unconditionally, no matter what the length is – this one is denoted A_∞.

It is clear that A_∞ has no bounded competitiveness: for any given $c > 1$, we can make the season last for at least $c \cdot B$ days, in which case the cost of A_∞ becomes at least $c \cdot B$ while the optimum is B, hence the ratio becomes at least c for this input.

Now let us analyze the competitiveness of A_i. We consider three cases, depending on whether $i < B$, $i = B$ or $i > B$ holds.

- If $i < B$, then let us set $N = i$. Then, for this particular input, the optimal cost is $\mathrm{Opt} = \min\{i, B\} = i$, while the cost of A_i is $i-1+B$. This makes the ratio $\frac{i-1+B}{i} = 1+\frac{B-1}{i}$ which is at least 2, since $B - 1 \geq i$.
- If $i > B$, then let us set $N = i$ again. Then, the optimal cost is $\mathrm{Opt} = B$. while the cost of A_i is again $i - 1 + B$, making the ratio $\frac{i-1+B}{B} = 1 + \frac{i-1}{B} \geq 2$ again, this time due to $i - 1 \geq B$.

- Now for the case $i = B$, then we claim that A_B has a competitive ratio of $2 - \frac{1}{B}$. To see that this is the worst possible ratio (there was no need to settle that fact in the previous two cases as we needed only a lower bound for those), we again split the analysis into two subcases, depending whether $N < B$ or $N \geq B$ holds.
  - If $N < B$, then Opt $= N$ and the algorithm rents the skies for the whole season, making the ratio to be 1 (that is, the output is optimal in these cases).
  - If $N \geq B$, then Opt $= B$, and the algorithm purchases the skies on the Bth day, making its cost to be $2B - 1$ and thus the ratio becomes $\frac{2B-1}{B} = 2 - \frac{1}{B}$, as claimed.

Thus, we get that A_B has a competitive ratio of exactly $2-\frac{1}{B}$ (since we analyzed all the possible cases for N), while A_i has a competitive ratio of at least 2 (since we showed a "bad enough" input for each such algorithm) for all the other choices of i. Since A_∞ is not competitive at all, we can conclude that A_B is the optimal online algorithm for the ski rental problem, having the competitive ratio of $2 - \frac{1}{B}$.

The paging problem

In the paging problem, there are two (positive integer) parameters n and k. We assume $1 < k < n$ (the problem becomes trivial in the other cases). The parameter k is called the size of the cache, while the parameter n is called the size of the disk.

An input sequence has the form $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_t)$ (now n denotes the disk size, we cannot use it to stand for the length of the input sequence) where $1 \leq \sigma_i \leq n$ for each i, that is, each request asks for a "page" of the disk.

In a response sequence, each individual response τ_i is a number between 0 and n, inclusive. The intuition is that we have a cache memory C that can hold up to k pages of the disk, this memory is empty in the beginning. During the serving the request, the algorithm has to ensure that the requested page is in the cache: if it's already in the cache, then there's nothing to be done and it's free (that action is encoded by the response 0), otherwise, if it's not but the cache is not yet full, the algorithm may load the page from the disk for a unit cost (this action is also encoded by the response 0), otherwise (that is, the cache has already k pages but neither of them is the same as the requested one), the algorithm has to drop one of the cached pages, and load the requested page in place of the discarded one, for a unit cost. If the algorithm happens to drop page p, then this action is encoded by the number p.

Formally, the action sequence $(\tau_1, \ldots, \tau_t)$ is a solution for the input sequence $(\sigma_1, \ldots, \sigma_t)$ if and only if for each $0 \leq i \leq t$, the following cache configuration C_i defined inductively has a size of at most k: $C_0 = \emptyset$ and $C_{i+1} = C_i - \{\tau_{i+1}\} \cup \{\sigma_{i+1}\}$.

To see this in action, suppose the cache size is $k = 3$, the input sequence is $(1, 2, 3, 1, 2, 1, 4, 5, 1, 2, 3, 4)$ and the response sequence is $(0, 0, 0, 0, 0, 0, 3, 4, 0, 0, 2, 1)$, then this corresponds to the following sequence of cache configurations: initially, the cache is empty, $C_0 = \emptyset$. Then, a request comes for the page 1, which is responded by a 0, meaning the new configuration is $C_1 = \emptyset - \{0\} \cup \{1\} = \{1\}$ (removing the element 0 from a set that do not contain 0 at all does not change anything). Then, a request comes for page 2, which is again responded by a 0, making the new configuration to be $C_2 = \{1\} - \{0\} \cup \{2\} = \{1, 2\}$ (again, removing the 0 from the set $\{1\}$ which does not contain 0 in the first place does not change the set. Then, inserting 2 to this set yields the set $\{1, 2\}$). After that, the request 3 is responded by a 0, making $C_3 = \{1, 2\} - \{0\} \cup \{3\} = \{1, 2, 3\}$. Note that in this phase, the cache gradually

gets filled up without discarding a single page: we say that the algorithm filled the cache. Now, a request of 1 comes again, which is responded by a 0 (which, in this case means that 1 is already in the cache). The new configuration is $\{1,2,3\} - \{0\} \cup \{1\} = \{1,2,3\}$ which is fine as it still has at most $k = 3$ elements. Then, the upcoming requests for the pages 2 and 1 are also responded by zeros, the cache still holds the values $\{1,2,3\}$ after that.

Now the request for page 4 comes, which is responded by the first nonzero entry 3 in our response sequence. This makes the next cache configuration to be $\{1,2,3\} - \{3\} \cup \{4\} = \{1,2,4\}$. That is, if our response is a positive integer j, that means "drop the page j and load the requested page in place of that". Now comes a request for page 5, which is responded by a 4, making the new configuration to be $\{1,2,4\} - \{4\} \cup \{5\} = \{1,2,5\}$. Then, request for page 1 is served by a 0 which is also fine since 1 is already present in the cache and thus the new configuration $\{1,2,5\} - \{0\} \cup \{1\} = \{1,2,5\}$ still has only three entries. Similarly, the next request is for page 2, this is handled by a 0 again. The next request is for page 3, this can only be handled by either a 1, a 2 or a 5 (we have to discard one of our cache pages), we decide to handle it by a 2, making the new configuration to be $\{1,3,5\}$, and when the final request 4 arrives, that can be handled by a 1 again.

Since during the whole computation the size of our cache is at most $k = 3$, this response sequence is a solution. The total cost of this solution is 7 (there is a cost of 3 in the phase during which the cache gets filled, then there is a cost for each "cache miss" when we have to drop one of our pages – these are the responses which are encoded by some positive number).

A single step of an algorithm will in some cases be depicted as $C_i \xrightarrow[\tau_{i+1}]{\sigma_{i+1}} C_{i+1}$, like $\{1,2,4\} \xrightarrow[4]{5} \{1,2,5\}$, that is, we had the cache configuration $\{1,2,4\}$, we got the request for page 5, we discarded the page 4 and we ended up in the configuration $\{1,2,5\}$. We might drop the τ, the σ or both if they are not needed in the analysis.

The reader is encouraged to verify that for this particular input, 7 is the optimal cost.

It is easy to see that any algorithm that discards a page when the cache is not yet full can be transformed to another one which loads the request without discarding the page, and only discards the page upon the first cache miss when the cache is already full – this latter algorithm always produces a solution which is at least as good as the original one. So from this point on, we assume every algorithm (online or not) for the paging problem first fills its cache, and discards pages only after that. The difference lies in the strategy of choosing which page to discard.

A number of algorithms might come to mind for this problem:

- LFU, or Least Frequently Used, discards the page to which the least number of requests arrived so far.
- FIFO, or First In, First Out, manages a "fair" rotation in the following sense: it always discards the page which was loaded the longest time ago.
- LRU, or Least Recently Used, discards the page who had access (not only loading) the longest time ago.
- LIFO, or Last In, First Out, discards the page which was accessed (or loaded... neither of them makes any sense anyways) most recently.

All the above algorithms are online. An offline algorithm is LFD, or Longest Forward Distance,

which discards the page which will not be requested for the longest time in the future[2].

Example runs

Let us consider the input sequence $(1, 2, 3, 1, 2, 1, 4, 5, 1, 2, 3, 4)$ with cache size $k = 3$. All the above algorithms begin with filling the cache, and reach the configuration $\{1, 2, 3\}$, when the request for the page 4 comes in.

- LFU counts the number of requests. So far page 1 had 3 requests, page 2 had 2 requests and page 3 had a single one, thus 3 is dropped and the new configuration becomes $\{1, 2, 4\}$. Handling the request 5, the page 4 is dropped (as now it is the page with a single request so far), yielding $\{1, 2, 5\}$. Then 1 and 2 are served. For handling 3, the algorithm drops the page 5, having a single request so far and the configuration becomes $\{1, 2, 3\}$. Finally, for handling the request 4, the page 3 is dropped as it had two requests so far while page 2 had three and page 1 had 4. (In the case of ties, let's say LFU drops the candidate having the least number.) So, LFU has a cost of $3 + 4 = 7$.
- When handling the first request for page 4, FIFO drops page 1 as it was the first one to be loaded, making the configuration to be $\{2, 3, 4\}$. Then, for serving 5, 2 is dropped, we are at $\{3, 4, 5\}$. Now for serving 1, 3 is dropped and we are at $\{1, 4, 5\}$. Then comes the 2, and 4 gets dropped as it's present since the 7th request, 5 is present since the 8th and 1 is present since the 10th. So we are at $\{1, 2, 5\}$. Then comes the request for 3, we discard the 5 and we are at $\{1, 2, 3\}$ again. Finally, to serve 4, we drop the 1. The cost of FIFO is 9.
- When LRU handles the first request for page 4, it drops the 3 since 1 is accessed just in the previous request and 2 is accessed right before that but 3 was accessed four steps ago. So the new configuration is $\{1, 2, 4\}$. Then, for serving the 5, we drop 2, yielding $\{1, 4, 5\}$. Luckily, 1 is already in the cache now (but its "recentness" gets updated at this point). For handling the request 2, we discard 4, and we are at $\{1, 2, 5\}$. Then, we drop 5 and 1, when handling 3 and 4, respectively. The cost of LRU is 8.
- LIFO discards 1 when 4 is requested as 1 was accessed just before that. So we are at $\{2, 3, 4\}$ and discard the 4 (as it's the most recently touched page) when the request for 5 comes by, so we are at $\{2, 3, 5\}$. Now LIFO handles the request for 1 by dropping 5, and we are at $\{1, 2, 3\}$. Then, the requests $1, 2, 3$ are luckily all in the cache, and the last request for page 4 makes LIFO discard 3 since that's the one we just touched in the previous step. LIFO has a cost of 7.
- Finally, running LFD means the following. When the request for the first 4 comes in, we see that in the future, 1 will be requested within 2 steps, 2 will be requested within 3 steps and 3 will be requested within four steps, thus LFD discards 3 and we are at $\{1, 2, 4\}$. Then when the request for 5 comes in, we see that 1 will be requested in the next step, 2 in the one after that and 4 will be only requested later, thus we discard the 4 and we are at $\{1, 2, 5\}$. Now the requests for 1 and 2 are served without a cache miss. For serving 3, we can discard now anything since none of the pages we have cached get requested anymore, so let's say we drop 1 (being the least one amongst the candidates) and end up at $\{2, 3, 5\}$, then for serving 4 we end up similarly by choosing 2, at $\{3, 4, 5\}$. The cost of LFD is 7. (The reader might verify that our first sample run was a potential LFD run.)

[2]LFD actually produces the optimal solution. The proof of this will be added to the lecture notes later.

Bad algorithms

Even though LFU and LIFO perform well on these particular inputs, they are not competitive. To see that, for every $c > 1$ we have to create an input sequence on which the corresponding ratio gets at least c.

When the cache size is k, then let us consider a long enough input sequence

$$1, 2, 3, \ldots, k, k+1, k+2, k+1, k+2, \ldots, k+1, k+2.$$

To handle this sequence, LIFO first fills up the cache, then discards k and loads $k+1$, then discards $k+1$ and loads $k+2$, then discards $k+2$ to load $k+1$ and so on, essentially using a single cache page for $k+1$ and $k+2$. Clearly, this solution misses the cache on each request, so if the length of the sequence is at least $c \cdot (k+2)$, then the cost of LIFO is also at least $c \cdot (k+2)$.

On the other hand a possible solution discards 1 after filling the cache, then discards 2. At this point both $k+1$ and $k+2$ are in the cache so the remaining request will become served at no cost, making the cost of this solution to be $k+2$. Hence the ratio is at least c, which can be an arbitrarily large number, so LIFO is not competitive.

For this particular input, LFU performs relatively well, since after the second request for $k+2$ it won't discard $k+1$ nor $k+2$. However, for the input sequence

$$1^\ell, 2^\ell, \ldots, k^\ell, k+1, k+2, k+1, k+2, k+1, k+2 \ldots, k+1, k+2,$$

where σ^ℓ means the sequence $\sigma, \sigma, \ldots, \sigma$ of length $\ell = (k+2)c$, and the total number of $k+1, k+2$ blocks is ℓ, LFU will use the same single slot for handling the requests for $k+1$ and $k+2$ since these are requested less than ℓ times till the end, while the others are requested at least ℓ times during the initial phase. The optimal solution would of course discard 1 and 2 (say), when serving $k+1$ and $k+2$ respectively and thus the optimum is at most $k+2$, while LFU has a cost of $k + 2(k+2)c$, making the ratio more than $2c$ for an arbitrarily large c.

The best hope

Even when $n = k+1$, any deterministic online algorithm A can be tricked as follows: whenever A discards some page p, then the next request will happen to be p.

For example, running LRU with a cache size of $k = 3$, we might construct the input sequence that starts by $(1, 2, 3, 4)$, then for serving 4, LRU discards the 1 so we request 1, to handle the 1 LRU discards the 2, so we request 2, etc, that is

$$\{1,2,3\} \xrightarrow[1]{4} \{2,3,4\} \xrightarrow[2]{1} \{1,3,4\} \xrightarrow[3]{2} \{1,2,4\} \xrightarrow[4]{3} \ldots$$

and we can enforce a cache miss for each step. So, for any deterministic online algorithm A there is an input of length m, for an arbitrary m, on which A has a cost of m.

At the same time, LFD (which is not an online algorithm) has a cost of at most $\frac{m}{k}$ on any input sequence of length m (here we use the assumption that $n = k+1$), after filling the cache: to see this, assume at a given step LFD discards p from the configuration C. This might happen if p does not occur at all after this step as a request: this would imply that all the rest of the sequence can be handled with a cost of 0 (as the size of the cache is k, all the other possible requests are cached at this point). So assume p occurs at some point. Since p is just discarded

from the cache of size k and $n = k + 1$, the next cache miss will be at the next request for p. But that next request has to be preceded by at least one request to each of the other $k - 1$ members of C, by the definition of LFD. Hence, each cache miss is followed by at least $k - 1$ cache hits, thus LFD makes at most m/k cache misses after the cache builds up, and one more possibly for the very last cache miss which might be followed by an instant termination of the input sequence, so that's a total of $k + (m/k) + 1$ cache misses for a sequence of length of at least $k + m + 1$. When m tends towards infinity, the ratio $\frac{k+m+1}{k+(m/k)+1}$ approaches k, implying that the best online deterministic algorithm for the paging problem cannot be better than k-competitive, not even for the case $n = k + 1$.

Good algorithms

In this subsection we show that LRU and FIFO, and a whole class of online algorithms called "marking" algorithms are k-competitive. Along with the result of the previous section we conclude that they are optimal in this sense.

A marking algorithm works as follows.

- Initially, it just fills the cache as all the other ones.
- From that point, each member of the cache can be either marked or unmarked. After filling the cache, let us mark all the pages in the cache.
- From that point, when a request comes in for a page p...
  - If p is in the cache, then we mark p and proceed.
  - Otherwise, let us do the following.
    - If all the pages in the cache are marked, then let us erase all the marks.
    - At this point there is at least one unmarked page. Let us choose one unmarked page by some algorithm, discard that page, load p and mark it.

For example, if we get the input $(1, 2, 3, 2, 4, 2, 4, 3, 5, 1, 2, 1, 5, 4, 3, 5)$, $k = 3$ and we (say) always choose the unmarked page to be discarded to be the one having the smallest possible index, then the run is (star denoting marked pages): first we fill the cache, reach the configuration $\{1^*, 2^*, 3^*\}$. Then handle the request 2: it's in the cache, already marked, so there's nothing to do. Then, to handle 4, we first erase all the marks since all members of the cache are marked. Then, we could drop 1, 2 or 3; since we choose the smallest possible one, we drop 1, load 4 and mark it. We are at $\{2, 3, 4^*\}$. Then, a request for 2 arrives: since 2 is already in the cache, it's free but nevertheless, we mark the entry 2 and we are at $\{2^*, 3, 4^*\}$. Then we get a request for page 4, which is already in the cache and is marked, fine. Then, the request for 3 gets the page 3 marked a well (still for free), we are at $\{2^*, 3^*, 4^*\}$. The incoming request for page 5 gets all the marks erased, and we drop the page with the smallest index, that is, 2, load and mark page 5. We are at $\{3, 4, 5^*\}$. Then for handling the request 1, we drop the page 3 as that's the smaller among the two candidates, load and mark 1. We are at $\{1^*, 4, 5^*\}$. Now to handle 2, we drop 4 (that's the only possibility we have right now), load and mark 2, we are at $\{1^*, 2^*, 5^*\}$. Then, the request for 1 is free, and its entry is already marked, as well as the next request for page 5. Now a request for page 4 arrives, we erase all the marks and drop page 1, load and mark page 4, we are at $\{2, 4^*, 5\}$. Now handling the request for page 3, we arrive to $\{3^*, 4^*, 5\}$, finally to handle 5, we simply mark the 5 and end up at $\{3^*, 4^*, 5^*\}$.

For any marking algorithm, the whole input sequence can be partitioned into phases as follows: the borders of the phases are the time points where the algorithm erases all the marks. For this particular input and selection algorithm, the phases are

$$(1,2,3,2),\ (4,2,4,3),\ (5,1,2,1,5),\ (4,3,5).$$

It turns out that the actual algorithm for choosing the unmarked page to discard does not play any role for determining the phases: the first phase is a maximal prefix of the input sequence in which there are at most k different requests (thus, the second phase starts with the $k+1$th disjoint request, in this case, $1, 2$ and 3 are the pages requested during the first phase, then 4 is a different one, starting the second phase, in which $2, 3$ and 4 are requested, then the 5, being the fourth distinct page to be requested begins the next phase, in which $1, 2$ and 5 are requested, and 4 starts the next phase and so on.

Indeed: at the end of each phase, the cache consists of exactly those pages that were requested during this specific phase. To see this, we can use induction: the statement holds after the very first phase. Then, assuming a full cache of unmarked pages in the beginning of a phase, to each "new" request an unmarked page gets marked (either because it was already in the cache from the previous phase, but yet unmarked, or because it caused a cache miss, in which case we drop an unmarked page, and load-and-mark this new one). Hence all the pages will get marked when k different requests arrive during the phase, and when the $k+1$th comes in, that will cause erasing all the marks and the beginning of the new phase.

Since by definition, the algorithm marks a page for each cache miss (and possibly marks a page even when there is no cache miss), and k pages get marked during a single phase, we immediately get that any marking algorithm induces at most k cache misses during a phase. (This is also true for the first phase, when the cache gets filled.)

Now consider any algorithm working on the same input sequence. After handling the first request p of a phase, we know that the cache contains p. In this phase, there are at least $k-1$ pairwise different requests to be handled, which are all different from the first request of the next phase. (In the example above, after handling the request for page 4 in the second phase, there are still requests for 2 and 3 in this phase and the next one starts with a 5, a total of k pairwise different requests.) Moreover, these k requests are all different from p, which implies that not all of them can be in the cache after handling p (there are only $k-1$ possible entries for the k different requests), which means that there will be at least one cache miss from the second request of a phase till the first request of the next phase, inclusive. Hence, the optimal solution has at least ℓ cache misses for an input sequence consisting of ℓ phases (to be more precise, $k+\ell-2$ as the first phase will have k cache misses, but the last phase might have none), while any marking algorithm will make at most $k \cdot \ell$ cache misses, making the ratio $\frac{k \cdot \ell}{k+\ell-2} \leq k$. Hence, any marking-type online algorithm is at least k-competitive. Since we already know that they cannot be better than k-competitive, this means that the competitive ratio of any marking-type algorithm is exactly k. (Observe that this upper bound holds for any choice of n, not only for $n = k+1$.)

LRU is marking

The LRU algorithm can be formulated as follows:

- for each cache entry we also store a timestamp,
- if we have to discard an entry, we choose the one having the oldest timestamp,

- and whenever we load a page or get a request for a page already in the cache, we update its timestamp.

Without the explicit use of markings, LRU still counts as a marking algorithm: all we have to verify is that LRU cannot discard a page during a phase for which there was already an incoming request during the very same phase. But this is clear: if a page had an incoming request during the current phase, then its timestamp belongs to the current phase. If this is the oldest timestamp, that means all the timestamps belong to the current phase, which in turn implies that all the pages in the cache were already requested during the current phase, and now we have to discard one of them due to a cache miss – which means that we are not in this phase anymore but in the beginning of the next one. So indeed, during each phase, the cache entries which were requested in the phase are "protected", thus LRU is a marking algorithm.

Hence, LRU is k-competitive.

Although FIFO is not a marking algorithm, it is also k-competitive[3].

Randomized algorithms

Upon receiving a request σ_{i+1}, a randomized online algorithm might take into account the previous requests $\sigma_1, \ldots, \sigma_i$, its own responses $\tau_1, \ldots, \tau_i$ and a random number in order to compute τ_{i+1}. Note that the deterministic algorithms do not need to take their own responses into account as they themselves can compute their response sequence knowing $\sigma_1, \ldots, \sigma_i$.

Randomized ski rental

For example, a possible randomized algorithm for the ski rental problem might be the following:

- We keep renting till the $\frac{3}{4}B$th day;
- On the $\frac{3}{4}B$th day, we flip a coin.
  - If it's heads, then we buy the skis on this day.
  - Otherwise, we continue renting till the Bth day, on which (if that day comes at all) we buy the skis.

Clearly, we cannot talk about "the" definitive cost of a randomized online algorithm on a particular input. Instead, we might compute its expected value and compare it to the optimal offline cost, getting a(n expected) competitive ratio for a particular input; taking the maximum[4] of these ratios we get the competitive ratio of the algorithm itself.

For this particular algorithm A, we can proceed as follows:

- If the length N of the season is less than $\frac{3}{4}B$, then the algorithm will keep renting till the last day, no matter what, for a total cost of N. As this is the optimal cost as well, the ratio is 1 in these cases.

[3]The proof of this will be added to the lecture notes later.

[4]supremum, technically but that's okay

- If $\frac{3}{4}B \leq N < B$, then the optimal cost is still $\min\{N, B\} = N$. We can compute the expected cost as follows: with $\frac{1}{2}$ probability, we buy the skis on the $\frac{3}{4}B$th day. This choice has a cost of $(\frac{3}{4}B - 1 + B)$. Also with an $\frac{1}{2}$ probability, we don't buy the skis on that day and we won't reach the second "checkpoint" since N is less than B – this choice has a cost of N. Summing up the costs, weighted by their probabilities we get an expected cost $E(A) = \frac{1}{2}(\frac{3}{4}B - 1 + B) + \frac{1}{2}(N)$. Applying the assumption $\frac{3}{4}B \leq N$ (thus $B \leq \frac{4}{3}N$) we get that

$$E(A) = \frac{1}{2}(\frac{3}{4}B - 1 + B) + \frac{1}{2}(N) \leq \frac{1}{2}(N + \frac{4}{3}N) + \frac{1}{2}N = \frac{5}{3}N,$$

  making the ratio to be $\frac{5}{3}$.

- Finally, if $B \leq N$, then the optimal cost is B. At the same time, we pay either $\frac{3}{4}B - 1 + B$ (if we buy on the first checkpoint), or $2B - 1$ (if we buy on the second checkpoint), each with probability $\frac{1}{2}$. Then, the expected cost of A is

$$E(A) = \frac{1}{2}(\frac{3}{4}B - 1 + B) + \frac{1}{2}(2B - 1) \leq \frac{15}{8}B,$$

  making the ratio to be $\frac{15}{8}$ in this case.

Amongst all the possibilities, $\frac{15}{8}$ is the highest one, thus the competitive ratio of the above (randomized) algorithm A is $\frac{15}{8}$ (or better). When $B \geq 8$, this is already better than the $2 - \frac{1}{B}$ we had for the optimal deterministic online algorithm.

Of course, the choice of the two checkpoints, and the probability distribution of $\frac{1}{2} - \frac{1}{2}$ was rather arbitrary and only illustrates the technique itself. The reader is encouraged to devise a better algorithm, either by moving the probability (that's a bit easier to optimize, ends up in solving a linear system of a single variable) or one of the checkpoints (which in turn ends up in solving a quadratic equation.

Exercise

Optimize the probability for the previous scenario: give the best possible p such that the following randomized algorithm:

With probability p, we buy on the $\frac{3}{4}B$th day; with probability $1 - p$, we buy on the Bth day.

Solution

TODO

Exercise

Optimize the first checkpoint for the previous scenario: give the best possible $\alpha < 1$ such that the following randomized algorithm

With probability $\frac{1}{2}$, we buy on the $(\alpha \cdot B)$th day; with probability $\frac{1}{2}$, we buy on the Bth day.

Solution

TODO

Later, after a crash course to game theory, we'll see how to devise an optimal randomized online algorithm for this problem.

Randomized marking

For the paging problem, we can make the marking algorithm randomized by choosing the page to be discarded uniformly at random – so if there are, say, three unmarked pages, then each of them will be discarded by a probability of $\frac{1}{3}$ and so on. It is clear that the algorithm remains k-competitive even in the worst possible case, since during a phase it will still have at most k cache misses.

However, we can show that the competitive ratio of the algorithm improves by an order of magnitude by exploiting randomness. To do this, we have to refine our phase-based analysis we applied for the deterministic case. For a recap, for an input sequence of the paging problem with cache size k, we defined the first phase as a maximal-length prefix of the input containing at most k different requests, this determines the starting point of the second phase which is defined as the maximal-length subsequence of the input starting at that request and containing at most k different requests, and so on. So for $k = 3$, the following is a partitioning of an input to phases:

$$(1, 2, 3, 2),\ (4, 2, 4, 3),\ (5, 1, 2, 1, 5),\ (4, 3, 5)$$

Let ℓ denote the number of phases. We know that on the boundaries, the cache consists of exactly the pages that were requested during the previous phase, as this holds for all variants of the marking algorithms, randomized being no exception here.

For each integer $1 \leq i \leq \ell$, let t_i stand for the number of "new" requests during phase i: a request is new if it was not requested during the previous phase. As an extension, let us consider each request of the first phase to be new. So in the previous example, $1, 2$ and 3 are the new requests in phase 1, making $t_1 = 3$ (t_1 is always k if the cache gets filled). In the second phase, there are requests for the pages 2, 3 and 4, but as 2 and 3 were already requested during the first phase, only 4 is considered to be new, making $t_2 = 1$. In the third phase, 5 and 1 are new (observe that only the previous phase counts, it does not matter that 1 was requested two phases before), making $t_2 = 2$ and finally, $t_4 = 2$ as 3 and 4 are new but 5 is not.

Clearly, the number of "old" (as in, not new) requests is $k - t_i$ in phase i, assuming the phase is not the last one.

Now let us try to analyze an arbitrary solution. We claim that during phases i and $i + 1$ together, any solution makes at least t_{i+1} cache misses. Indeed, in the beginning of phase i we have k cache entries, and during the next two phases a grand total of $k + t_{i+1}$ pairwise different requests arrive: k during the ith phase and an additional t_{i+1} during the next one. As only k can be cached in the beginning of the phase, this implies at least t_{i+1} cache misses during these two phases. Now if we denote the number of cache misses in phase i by M_i, we get that the total cost of the solution is $M_1 + \ldots + M_\ell$, which is at least $\frac{M_1}{2} + \frac{M_1+M_2}{2} + \frac{M_2+M_3}{2} + \ldots + \frac{M_{\ell-1}+M_\ell}{2}$, which is by the previous argument at least $\frac{t_1}{2} + \frac{t_2}{2} + \ldots \frac{t_\ell}{2}$, that is, $\frac{1}{2}\sum_{i=1}^{\ell} t_i$ is a lower bound for the offline optimal solution.

Now let us try to bound (from above) the expected number of cache misses during phase i. Clearly, each new request definitely induce a cache miss, so that's t_i for sure. Now we bound from above the probability of the yth old request, $y = 1, \ldots, k - t_i$, inducing a cache miss. Let x be the number of new requests preceding the yth old request, clearly $x \leq t_i$. The worst possible scenario for this yth old request is that when the $y - 1$ old requests come first, each marking its own entry in the cache, then the x new requests arrive, bombarding the remaining $k-(y-1)$ cache entries (among which the yth old request is still there), occupying x from those uniformly at random, since this distribution maximizes the chance of the original cache entry holding the yth old request gets overwritten (thus inducing a cache miss when requested). Of course the larger the x, the larger the chance of a cache miss. So, the probability of a cache miss is at most the probability with which we hit a particular entry amongst $k - (y - 1)$, if we choose t_i from them uniformly at random. This chance is proportional to t_i, namely $\frac{t_i}{k-(y-1)}$: indeed, the chance of not overwriting the entry (thus NOT causing a cache miss) with the first shot is $\frac{k-(y-1)-1}{k-(y-1)}$. Then, if the first new request did not drop the target entry, then the second has a chance of $\frac{k-(y-1)-2}{k-(y-1)-1}$ (as there are now one less entries to discard, the previously hit one being now marked), and so on. In total, the probability of avoiding is

$$\frac{k-(y-1)-1}{k-(y-1)} \cdot \frac{k-(y-1)-2}{k-(y-1)-1} \cdot \frac{k-(y-1)-3}{k-(y-1)-2} \cdot \ldots \cdot \frac{k-(y-1)-t_i}{k-(y-1)-(t_i-1)} = \frac{k-(y-1)-t_i}{k-(y-1)},$$

which makes the chance of being hit (thus the chance of a cache miss) to be $\frac{t_i}{k-(y-1)}$.

That's an upper bound of the expected cost of serving the yth old request during phase i. As the expected values are additive we get that the total cost of serving all the old requests is at most

$$\frac{t_i}{k-(1-1)} + \frac{t_i}{k-(2-1)} + \ldots + \frac{t_i}{k-(k-t_i-1)},$$

which further equals to

$$t_i\Big(\frac{1}{k} + \frac{1}{k-1} + \ldots + \frac{1}{t_i+1}\Big),$$

which is, as $t_i \geq 1$ (there's at least one new request in each phase, namely the very first request of the phase – that's why it's a new phase) larger than

$$t_i\Big(\frac{1}{k} + \frac{1}{k-1} + \ldots + \frac{1}{2}\Big),$$

which, together with the cost t_i of the new requests, makes the expected cost of the randomized algorithm during phase i to be at most $t_i \sum_{j=1}^{k} \frac{1}{j}$.

Hence, as the total expected cost is then upperbounded by $\Big(\sum_{i=1}^{\ell} t_i\Big) \sum_{j=1}^{k} \frac{1}{j}$, makes the ratio to be at most $2 \sum_{j=1}^{k} \frac{1}{j}$.

This quantity $\sum_{j=1}^{k} \frac{1}{j}$ is well-known, is usually denoted H_k and is known to be around $\ln k$. (Upperbounded by $1 + \ln k$ via integrating $\frac{1}{x}$ with respect to x from 1 to k and noting that this integral is larger as the sum can be found as a total area of disjoint rectangles, strictly below the $1/x$ line but that's not needed in this course.)

Which means that the competitive ratio of the randomized marking algorithm is at most $2H_k \approx 2 + 2 \ln k$, and that's much better than the deterministic variants' k when k is large enough.

Game theory - a handy tool to analyze randomized algorithms

(Disclaimer: this part is not intended to be a complete crash course in game theory – we define here only those core notions which we actually use for the analysis of randomized online algorithms.)

A matrix game is defined by a cost (or payoff) matrix (duh) $C \in \mathbb{R}^{n \times m}$ for some integers n and m, that is, a bunch of real numbers arranged into n rows and m columns. The game is played between two competing players: player A and player B. In the "pure" variant of the game, A picks a row, say row i, of the matrix, B (independently) picks a column, say column j of the matrix and the value of this run of the game is $C_{i,j}$: the entry in the ith row and jth column of C. The two players are competing in the following sense: A tries to maximize this value, while B tries to minimize it.

As an example, when $C = \begin{pmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{pmatrix}$, then if A picks the first row, then no matter what B does, the value of this run will be at least 0 (the minimum entry in this row). If A picks the second row, then the value of this run will be at least -1, while if A picks the third row, then the value of the run will be at least -2. Since A wants to maximize the value, he[5] can ensure a value of at least 0, by picking the first row. In general, A can ensure a value of $\max_i \min_j C_{i,j}$.

Turning the focus to player B, if she picks the first column, then the value A can make out of this run is at most 0. If she picks the second column, then no matter what A does, the value will be at most 1, and by picking the third column, the value will be at most 2. Since player B wants to minimize the value of the run, by picking the first column she can ensure a value of at most 0. In general, player B can ensure a value of $\min_j \max_i C_{i,j}$.

In this particular game, it happens so that

$$\max_i \min_j C_{i,j} \;=\; \min_j \max_i C_{i,j},$$

which means that both players can force that the value of the run is simultaneously at least 0 and at most 0 as well. In such cases, the optimal play for both of them is to pick a row (column, resp.) which maximizes (minimizes, resp.) the corresponding term, in this case, A should pick the first row and B should pick the first column. If any of them deviates from this strategy, then they will lose and the other player will gain something.

This entry (i^*, j^*) (for which $\min_j C_{i^*,j} = \max_i C_{i,j^*}$) is the so-called saddle point of the matrix, or a pure Nash equilibrium of the game and, if exists, gives an easy way for an optimal deterministic strategy for both of the players.

However, if we consider the game matrix $C = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$, then the situation is not so clear: player A can ensure a value of at least -1, while player B can ensure a value of at most 1. Since these two values do not coincide, there is no single optimal pure strategy for any of the players: if A picks the first row, then B should pick the second column, getting a value of -1. But then, if A knows that B plays the second column, then A should pick the second row, getting a value of 1. Should B know that, she would rather pick the first column and so on, resulting in an "oscillating" behavior that does not happen in the case when there is a Nash equilibrium: if an equilibrium is present, then if any player changes their mind, they lose something (while the other player still playing their equilibrium strategy).

[5] usually, player A is regarded as male while player B is regarded as female

Instead, we study so-called mixed strategies for these games. In a mixed strategy, the strategy of A is a probability distribution $\boldsymbol{p} = (p_1, \ldots, p_n)$ (that is, $0 \leq p_i \leq 1$ and $\sum p_i = 1$), and the strategy of B is also a probability distribution $\boldsymbol{q} = (q_1, \ldots, q_m)$. The game becomes randomized: A picks each row i with the probability p_i, while B picks column j with probability q_j. Then, the entry (i, j) is picked with the probability $p_i \cdot q_j$, and has the value $C_{i,j}$. This means that the expected value of the game is $\sum_{i=1}^{n} \sum_{j=1}^{m} p_i q_j C_{i,j}$, which, if we use matrix notation, happens to be $\boldsymbol{p} C \boldsymbol{q}^T$ where ^{T} means transpose of the vector.

Observe that a pure strategy is a special case of a mixed one: if A deterministically picks the ith row, that's him playing according to the mixed strategy e_i (the ith unit vector) that sets p_i to 1 and all the others to 0.

Similarly to the argument we used in the case of pure games, A can ensure an expected value of at least $\max_{\boldsymbol{p}} \min_{\boldsymbol{q}} \boldsymbol{p} C \boldsymbol{q}^T$, while player B can ensure an expected value of at most $\min_{\boldsymbol{q}} \max_{\boldsymbol{p}} \boldsymbol{p} C \boldsymbol{q}^T$.

Contrary to the case of pure strategies, the Fundamental Theorem of Game Theory states that these two values always coincide:

$$\max_{\boldsymbol{p}} \min_{\boldsymbol{q}} \boldsymbol{p} C \boldsymbol{q}^T \;=\; \min_{\boldsymbol{q}} \max_{\boldsymbol{p}} \boldsymbol{p} C \boldsymbol{q}^T.$$

The value itself is called the value of the game, while the pair $(\boldsymbol{p}^*, \boldsymbol{q}^*)$ of the optimal strategies is called the mixed equilibrium strategy of the game.

For example, if in our example matrix $C = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$ player A uses the strategy $(\frac{1}{4}, \frac{3}{4})$ and player B uses the strategy[6] $(\frac{3}{4}, \frac{1}{4})$, then the expected value of the game is

$$(\frac{1}{4}, \frac{3}{4}) \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} \frac{3}{4} \\ \frac{1}{4} \end{pmatrix} \;=\; (-\frac{1}{2}, \frac{1}{2}) \begin{pmatrix} \frac{3}{4} \\ \frac{1}{4} \end{pmatrix} \;=\; -\frac{1}{4}.$$

However, if B knows that A plays $(\frac{3}{4}, \frac{1}{4})$ for strategy, she can do better by playing $(1, 0)$ instead: then the value of the game becomes

$$(\frac{1}{4}, \frac{3}{4}) \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \;=\; (-\frac{1}{2}, \frac{1}{2}) \begin{pmatrix} 1 \\ 0 \end{pmatrix} \;=\; -\frac{1}{2},$$

which is smaller, so it's better for B. Actually, if A plays this strategy, then $(1, 0)$ happens to be an optimal counterstrategy – so in this case, for a fixed mixed strategy $\boldsymbol{p}$ of player A, there is an optimal pure strategy e_j minimizing the game value.

This is not a coincidence, the Loomis theorem says it's always the case: for any game matrix C and fixed mixed strategy $\boldsymbol{p}$, there is a column index j such that

$$\min_{\boldsymbol{q}} \boldsymbol{p} C \boldsymbol{q}^T \;=\; \min_{j} \boldsymbol{p} C e_j^T,$$

and similarly for any fixed $\boldsymbol{q}$ there is an optimal counterstrategy of the form e_i.

This makes the actual computation of the value of the game easier, as we now seek for the maximizer (minimizer, resp.) of the equation

$$\max_{\boldsymbol{p}} \min_{j} \boldsymbol{p} C e_j^T \;=\; \min_{\boldsymbol{q}} \max_{i} e_i C \boldsymbol{q}^T.$$

[6] TODO: fix font sizes

Exercise

Let us determine the optimal strategies for the game matrix $\begin{pmatrix} 2 & 4 \\ 5 & 3 \end{pmatrix}$, and the value of the game.

Solution

For determining the optimal strategy for Player A, we can write $\boldsymbol{p} = (p_1, p_2)$ explicitly, so we are trying to maximize

$$\max_{(p_1,p_2)} \min_j (p_1, p_2) \begin{pmatrix} 2 & 4 \\ 5 & 3 \end{pmatrix} e_j^T$$

If we multiply the first two matrices, we get

$$\max_{(p_1,p_2)} \min_j (2p_1 + 5p_2, 4p_1 + 3p_2) e_j^T$$

Now, thanks to the Loomis theorem, we only have to consider the unit vectors e_j^T. Multiplication by these guys from the right means picking an entry, so the above formula further equals

$$\max_{(p_1,p_2)} \min\{\ 2p_1 + 5p_2,\ 4p_1 + 3p_2\ \}.$$

From this point on, there is a general approach, namely: if we denote the value of the game by a new variable t, then we have an objective function t which we want to maximize (as we're devising a strategy for player A). We know that this t is the minimum of the linear functions appearing in the brackets (if the matrix has m columns, then there'll be m such terms). So, t is upperbounded by each of these expressions. Thus, we can write a linear system of inequalities:

$$\begin{aligned} t &\leq 2p_1 + 5p_2 \\ t &\leq 4p_1 + 3p_2 \\ 0 &\leq p_1 \\ 0 &\leq p_2 \\ 1 &= p_1 + p_2 \\ \max\ & t \end{aligned}$$

This is an LP problem that can be written mechanically from the matrix: make the product of C by the vector $(p_1, \ldots, p_n)$ as above, state that the new variable t is upperbounded by all the entries of the product (that's the first m inequalities), and state also that the p_i's are forming a probability distribution (last $n + 1$ rows), and maximize t.

(If one happens to compute a strategy for player B, then one has to make the product $C\boldsymbol{q}^T$ instead, write that the new variable t is lowerbounded by each entry of the product, that $\boldsymbol{q}$ is a probability distribution and minimizing t.)

If one feeds the above system to some LP solver (chances are, even your favorite spreadsheet program has some built-in LP solving functionality), one gets that the game's value is $t = 3.5$, and the optimal strategy for A is $(0.5, 0.5)$.

(The optimal strategy for B is $(0.25, 0.75)$, by the way.)

Now in the particular case when we have only two variables, we can do the job easier (though that's not really an application for analyzing a randomized online algorithm, but anyways...). In the above case, we can use the variable substitution $p_2 = 1 - p_1$ and get a linear system over one single variable: we should maximize $\min\{\ 5 - 3p_1,\ 3 + p_1\}$, where $0 \le p_1 \le 1$. Now if there are only two such inequalities, and we already ruled out the case of a pure Nash equilibrium, then chances are, the maximum will be at the point where these two lines meet (otherwise the optimal solution would be at either $p_1 = 0$ or at $p_1 = 1$, both being a pure strategy), which means we only have to make them equal and we're set: $5 - 3p_1 = 3 + p_1$, that is, $p_1 = 0.5$ and the common value is 3.5, just as before.

But having a 2×2 is not really a thing which we encounter if we apply game theory to our topic... in most cases we have an infinite matrix to begin with.

Application of game theory

Suppose we have an online problem for which all the possible randomized algorithms can be expressed as a probability distribution over the set of all deterministic algorithms.

For example, our example algorithm for ski rental that purchased the skis with probability 0.5 on the $\frac{3}{4}B$th day, and with probability 0.5 on the Bth day, is basically a randomized algorithm that runs $A_{\frac{3}{4}B}$ with probability 0.5, and A_B with probability 0.5.

Now let us construct the following matrix C: each row of C corresponds to an input of the problem, while each column of C corresponds to a deterministic online algorithm for the problem. This matrix typically is of infinite size on both dimensions.

Let the entry $C_{i,j}$ be the competitive ratio of algorithm j being run on input i, that is, the ratio $A_j(i)/\mathrm{Opt}(i)$.

As an example, consider the (trumpets) ski rental problem with $B = 4$. We are studying now this problem since both the inputs and the sensible algorithms are super easy: row i corresponds to the case when the length of the season is i, and column j corresponds to the algorithm A_j (which rents the skis for the first $j - 1$ days, then buy them). We could make an additional column for A_∞ as well (which we already know to perform badly) but we'll omit that detail for now. (Actually, it makes no sense to give a chance for A_∞ being run, it's so bad.)

The top-left 6×6 part of this matrix is

N	A_1	A_2	A_3	A_4	A_5	A_6	...
1	**4/1**	1/1	1/1	1/1	1/1	1/1	...
2	4/2	**5/2**	2/2	2/2	2/2	2/2	
3	4/3	5/3	**6/3**	3/3	3/3	3/3	...
4	4/4	5/4	6/4	**7/4**	4/4	4/4	
5	4/4	5/4	6/4	7/4	**8/4**	5/4	...
6	4/4	5/4	6/4	7/4	8/4	**9/4**	
⋮		⋮		⋮			⋱

For example, the entry for row $N = 1$ and column A_1 is $4/1 = 4$ since A_1 buys the skis for a price of 4 on the first day, while the optimal cost is $\min\{4, 1\} = 1$ if the season consists of a single day. Also, row 4, column 3 is 6/4 as in row 4 (and below), the optimum is 4, and A_3 rents for two days, and buys on the third day, for a total cost of 6.

In each column, the maximum entry is shown in boldface.

Now if we (as some sort of player B) choose an algorithm A_j, then its competitive ratio is defined as the maximum possible[7] value $\frac{A_j(\boldsymbol{\sigma})}{\mathrm{Opt}(\boldsymbol{\sigma})}$. This means that if our competition (as some sort of player A) picks the row maximizing the entry in column j, then his optimal strategy as player A in this infinite matrix game against our fixed strategy A_j is exactly the problem of determining the competitive ratio of A_j! For example, if we pick the column of A_3 as the minimizing player B, then we can ensure that the ratio is at most $6/3$ and indeed, if the maximizing player A picks the row $N = 3$, then the value of that specific run is exactly $6/3$, the competitive ratio of A_3 for this version of the ski rental problem with $B = 4$.

That is, if we want to construct the best possible deterministic online algorithm for this problem, that's exactly the same thing as computing the optimal pure strategy j^* as player B in this infinite matrix game! This approach works for every online problem (but we might of course bounce into problems if the enumeration of all the possible algorithms is not so easy for the problem in question). The reason why A_4 is the optimal online algorithm for this instance of the ski rental problem is that this is the value j^* which attains the minimum in the expression $\min_j \max_i \frac{A_j(i)}{\mathrm{Opt}(i)}$.

Then, if a randomized algorithm can be given as a probability distribution $\boldsymbol{q} = (q_1, q_2, \ldots)$ over the deterministic algorithms (e.g. we have already seen the case when $q_{3B/4} = q_B = 0.5$ and $q_j = 0$ for all the other js), then the expected competitive ratio of an algorithm for a given input i is $e_i C \boldsymbol{q}^T$. For the worst possible case, we get (as the minimizing player) the term $\max_i e_i C \boldsymbol{q}^T$ – so an optimal randomized algorithm is given as the minimizer of the expression

$$\min_{\boldsymbol{q}} \max_i e_i C \boldsymbol{q}^T,$$

which is (by the Fundamental Theorem of Game Theory and the Loomis Theorem) exactly the value of the matrix game given by the above constructed matrix!

Now if we do the math as before (not being scared of taking products of infinite matrices and vectors), we have to compute the product $C\boldsymbol{q}^T$, and make each entry a lower bound of the newly introduced variable t, stating that those $\boldsymbol{q}$s are forming a probability distribution and minimize t:

$$\begin{aligned}
4q_1 + q_2 + q_3 + q_4 + q_5 + q_6 + \ldots &\leq t \\
2q_1 + 5q_2/2 + q_3 + q_4 + q_5 + q_6 + \ldots &\leq t \\
4q_1/3 + 5q_2/3 + 2q_3 + q_4 + q_5 + q_6 + \ldots &\leq t \\
q_1 + 5q_2/4 + 3q_3/2 + 7q_4/4 + q_5 + q_6 + \ldots &\leq t \\
q_1 + 5q_2/4 + 3q_3/2 + 7q_4/4 + 2q_5 + 5q_6/4 + \ldots &\leq t \\
q_1 + 5q_2/4 + 3q_3/2 + 7q_4/4 + 2q_5 + 9q_6/4 + \ldots &\leq t \\
\ldots &\leq t \\
q_1 + q_2 + q_3 + q_4 + q_5 + q_6 + \ldots &= 1 \\
q_j &\geq 0 \\
\min t &
\end{aligned}$$

Now we have an infinite number of constraints and an infinite number of variables. In this form the system cannot be explicitly solved but in several cases, one can hope for reducing it to an equivalent form of finite size (e.g. by deducing some of the probabilities to have a value 0 in the optimal solution, hence being able to remove that particular column).

[7] again, I should say the "supremum" here... stay tuned

Consider rows i and $i+1$ for an arbitrary $i \geq 4$. In these rows, the optimum cost is 4 (by $i \geq 4$). All the algorithms till A_i buy the skis on day i at latest, hence the first i entry of these two rows coincide. In the columns of index $j \geq i+1$, A_j keeps renting in the first $i+1$ days, so $C_{i+1,j} = C_{i,j} + 1/4$ (division by 4 is due to that 4, the optimum is the denominator in both rows). Finally, in column i, we have $C_{i+1,j} = C_{i,j} + 4/4 = C_{i,j} + 1$ as A_{i+1} happens to buy the skis on day $i+1$, spending B more money. (See e.g. rows 4 and 5 of the matrix: a sudden increment of 4/4 happens under A_5, and for the columns right to that, there are increments of 1/4 between the two rows). This means that $C_{i+1,j} \geq C_{i,j}$ for each possible j when we have $i \geq B$.

Translated to the LP language, as the variables q_j are nonnegative, this means that if the constraint generated from row $i+1$ is satisfied, then so is the constraint generated by row i. That is, row 4 can be removed due to the presence of row 5; then, row 5 can be removed due to the presence of row 6, and so on. At the end we get that all the rows from row 4 and above can be replaced by a single inequality, which has $C_{i,i}$ as the coefficient of q_i when $i \geq 4$ and $C_{4,i}$ when $i < 4$.

Thus, we have now the following system of finite constraints

$$\begin{aligned}
4q_1 + q_2 + q_3 + q_4 + q_5 + q_6 + \ldots &\leq t \\
2q_1 + 5q_2/2 + q_3 + q_4 + q_5 + q_6 + \ldots &\leq t \\
4q_1/3 + 5q_2/3 + 2q_3 + q_4 + q_5 + q_6 + \ldots &\leq t \\
q_1 + 5q_2/4 + 3q_3/2 + 7q_4/4 + 8q_5/4 + 9q_6/4 + \ldots &\leq t \\
q_1 + q_2 + q_3 + q_4 + q_5 + q_6 + \ldots &= 1 \\
q_j &\geq 0 \\
\min t &
\end{aligned}$$

but still, we have an infinite number of variables. However, assume $\boldsymbol{q}$ is a solution for this system. As we are minimizing t, its value will be the minimum of the left-hand side of the first four constraints. Then we can transform this solution to another one $\boldsymbol{q}'$ in which $q'_4 = \sum_{j=4}^{\infty} q_j$, and $q'_j = q_j$ for $j < 4$. That is, we "collate" all the probabilities for running A_j with $j \geq 4$ and run A_4 instead with the summed probability.

Then, the left-hand sides of the first three constraints do not change, while the last one either decreases or remains the same. Also, $\boldsymbol{q}'$ is still a probability distribution. Setting t to be the minimum of the corresponding linear constraints we get that the value of the objective function is either improved or remains the same. Hence, we can set $q_j = 0$ for each $j \geq 5$ and get a finite LP system:

$$\begin{aligned}
4q_1 + q_2 + q_3 + q_4 &\leq t \\
2q_1 + 5q_2/2 + q_3 + q_4 &\leq t \\
4q_1/3 + 5q_2/3 + 2q_3 + q_4 &\leq t \\
q_1 + 5q_2/4 + 3q_3/2 + 7q_4/4 &\leq t \\
q_1 + q_2 + q_3 + q_4 &= 1 \\
q_j &\geq 0 \\
\min t &
\end{aligned}$$

Feeding it to our favorite LP solver, we get that the optimum value of $\frac{256}{175}$ is attained at $q_1 = \frac{27}{175}$, $q_2 = \frac{36}{175}$, $q_3 = \frac{48}{175}$, and $q_4 = \frac{64}{175}$.

Putting it back into our context, we have just proved that the best possible randomized online algorithm for the ski rental problem when the price of the skis is $B = 4$ is the following: we

plan to buy on the first day with a probability of $\frac{27}{175}$, on the second day with a probability of $\frac{36}{175}$, on the third day with a probability of $\frac{48}{175}$, and on the fourth day with a probability of $\frac{64}{175}$. This way, we get a $\frac{256}{175}$-competitive algorithm. (That's around 1.4629 by the way – compare it to the best possible deterministic one of $2 - 1/4 = 1.75$.)

The Yao Principle

Usually it's not that easy to solve the infinite system of inequalities generated for an online problem. Instead, we use the following chain of inequalities: for arbitrary probability distributions $\boldsymbol{q}_0$ and $\boldsymbol{p}_0$ over the deterministic inputs and the determistic algorithms we have

$$\min_j \boldsymbol{p}_0 C e_j^T \;\leq\; \max_{\boldsymbol{p}} \min_j \boldsymbol{p} C e_j^T \;\leq\; \min_{\boldsymbol{q}} \max_i e_i C \boldsymbol{q}^T \;\leq\; \max_i e_i C \boldsymbol{q}_0^T.$$

In particular, the part

$$\min_j \boldsymbol{p}_0 C e_j^T \;\leq\; \min_{\boldsymbol{q}} \max_i e_i C \boldsymbol{q}^T$$

states that if we give a probability distribution $\boldsymbol{p}_0$ over the set of inputs, that is, a *randomized input*, against which the best possible deterministic algorithm has a competitive ratio of K (or even worse), then this K is a lower bound of the competitive ratio of the best possible *randomized algorithm* (against a deterministic input, say, but that is the same as that against a randomized input, due to the Loomis theorem). This is called the Yao principle.

Let us see an application for the principle.

Exercise

Show that the best possible randomized online algorithm for the ski rental problem cannot have a better competitive ratio than 5/4.

Solution

We apply the Yao Principle. To this end, we have to specify a randomized input.

Let N, the length of the season, be $B/2$ with a probability of $\frac{1}{2}$ and $3B/2$ with a probability of $\frac{1}{2}$.

(Note: this choice is rather arbitrary. In principle, each choice would give us some lower bound for the competitive ratio. In this case, this is a "lucky choice" as in the end, it turns out to give a bound of 5/4 as needed. We'll give a more elaborated example in the next exercise in the context of the scheduling problem.)

Now we have to determine for each deterministic online algorithm A_j the expected competitive ratio $E\left(\frac{A(I)}{\mathrm{Opt(I)}}\right)$. If our chosen probability has a finite support, then this is simply the weighted sum of the competitive ratios, in this case:

$$E\left(\frac{A(I)}{\mathrm{Opt(I)}}\right) \;=\; \frac{1}{2}\frac{A(B/2)}{\mathrm{Opt}(B/2)} \;+\; \frac{1}{2}\frac{A(3B/2)}{\mathrm{Opt}(3B/2)}.$$

Recall that for the ski rental problem, $\mathrm{Opt}(N) = \min\{N, B\}$ so we have $\mathrm{Opt}(B/2) = B/2$ and $\mathrm{Opt}(3B/2) = B$, which means the expected competitive ratio of a deterministic algorithm A is

$$E\left(\frac{A(I)}{\mathrm{Opt(I)}}\right) \;=\; \frac{1}{2}\frac{A(B/2)}{B/2} \;+\; \frac{1}{2}\frac{A(3B/2)}{B}.$$

We know that each sensible deterministic online algorithm has the form A_j for some $j \geq 1$ which rents for $j-1$ days, then buys on the jth day. Now (and this is the usual scenario) we have to do a case analysis on these possible algorithms, and to give a lower bound for their competitive ratio.

There are three cases: either $j \leq B/2$, or $B/2 < j \leq 3B/2$, or $3B/2 < j$.

- If $j \leq B/2$, then the algorithm purchases the skis before the $B/2$th day, so that $A_j(B/2) = j - 1 + B$ and $A_j(3B/2) = j - 1 + B$ which makes the expected ratio at least

$$\begin{aligned} E\left(\frac{A(I)}{\text{Opt(I)}}\right) &= \frac{1}{2}\frac{A_j(B/2)}{B/2} + \frac{1}{2}\frac{A_j(3B/2)}{B} \\ &= \frac{1}{2}\frac{j-1+B}{B/2} + \frac{1}{2}\frac{j-1+B}{B} \\ &\geq \frac{1}{2}\frac{B}{B/2} + \frac{1}{2}\frac{B}{B} && \text{(as } j \geq 1\text{)} \\ &= \frac{3}{2}. \end{aligned}$$

- $B/2 < j \leq 3B/2$, then for the first case, the algorithm keeps renting for the whole length $N = B/2$ of the season, thus $A_j(B/2) = B/2$. For the second case, the algorithm buys eventually before reaching day $3B/2$, making $A_j(3B/2) = j - 1 + B$. Thus in that case, the expected ratio becomes at least

$$\begin{aligned} E\left(\frac{A(I)}{\text{Opt(I)}}\right) &= \frac{1}{2}\frac{A_j(B/2)}{B/2} + \frac{1}{2}\frac{A_j(3B/2)}{B} \\ &= \frac{1}{2}\frac{B/2}{B/2} + \frac{1}{2}\frac{j-1+B}{B} \\ &\geq \frac{1}{2} + \frac{1}{2}\cdot\frac{B/2+B}{B} && \text{(as } j > B/2\text{)} \\ &= \frac{5}{4}. \end{aligned}$$

- Finally, if $3B/2 < j$, then the algorithm keeps renting in both cases, thus $A_j(B/2) = B/2$ and $A_j(3B/2) = 3B/2$. Plugging in these values, we get that the ratio is at least

$$\begin{aligned} E\left(\frac{A(I)}{\text{Opt(I)}}\right) &= \frac{1}{2}\frac{A_j(B/2)}{B/2} + \frac{1}{2}\frac{A_j(3B/2)}{B} \\ &= \frac{1}{2}\frac{B/2}{B/2} + \frac{1}{2}\frac{3B/2}{B} \\ &= \frac{5}{4}. \end{aligned}$$

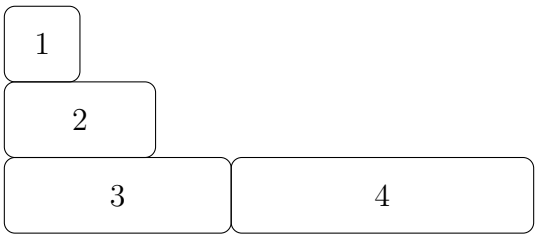
Thus we indeed got that $5/4$ is a lower bound in each of the cases, proving the statement.

The scheduling problem

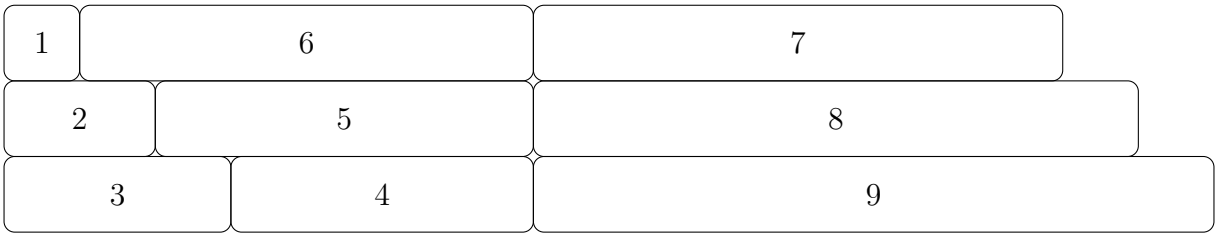
In the scheduling problem, we have a number m of machines, onto which we have to assign jobs. Each job has a processing time $p_i \geq 0$ so the input has the for $\boldsymbol{\sigma} = (p_1, \ldots, p_n)$. The action for request i is $\tau_i \in \{1, \ldots, m\}$, the index of the machine we assign the job to. The load of the jth machine is $\ell(j) = \sum_{\tau_i = j} p_i$, the total processing time of the jobs assigned to machine j, and the makespan of the generated schedule is $L = \max_j \ell(j)$, the maximal load. The objective function is the makespan.

As an example, if $m = 3$ and the input sequence is $(1, 2, 3, 4, 5, 6, 7, 8, 9)$, then a possible schedule is given by the responses $(1, 2, 3, 3, 2, 1, 1, 2, 3)$, meaning that the first job is scheduled onto the first machine, the second job is scheduled onto the second machine, the third job onto the third machine, the fourth is scheduled onto the third as well, the fifth one onto the second and so on.

We can visualize this schedule as one machine being drawn in a single line, and the jobs are being drawn in order, as rectangles, a job with a processing time p_i being a rectangle of size $p_i \times 1$. For example, the first four responses give us the following scheduling:



At this moment, the load of the first machine (top) is 1, the load of the second is $\ell(2) = 2$ and the load of the third is $\ell(3) = 7$, making the makespan to be 7. The whole schedule is visualized as



and the makespan of this one is $3 + 4 + 9 = 16$.

There are some obvious lower bound for the optimal makespan:

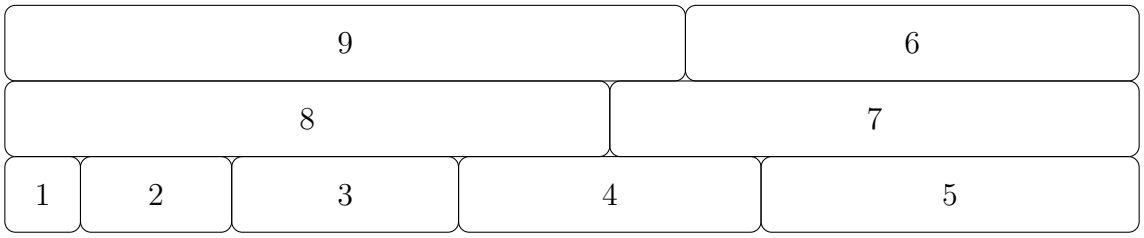
Proposition

Let us consider the optimal makespan L^* for the input $(p_1, \ldots, p_n)$ of the scheduling problem with m machines. Then

$$\max p_i \leq L^*$$
$$\frac{\sum p_i}{m} \leq L^*$$

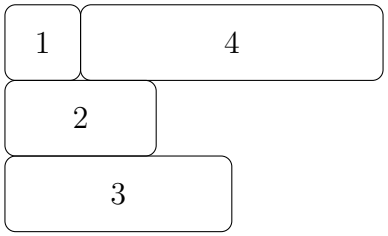
Indeed: the first bound comes from the fact that we have to schedule the largest job, making the load of its machine to be at least $\max p_i$, while the second states that $\sum p_i \leq m \cdot L^*$ which is clear since $\sum_i p_i = \sum_j \ell(j) \leq m \cdot \max_j \ell(j) = m \cdot L^*$.

These bounds give us $9 \leq L^*$ and $\frac{1+2+\ldots+9}{3} = 15 \leq L^*$. So if we happen to find a schedule with a makespan of 15, then it's optimal. And we can:

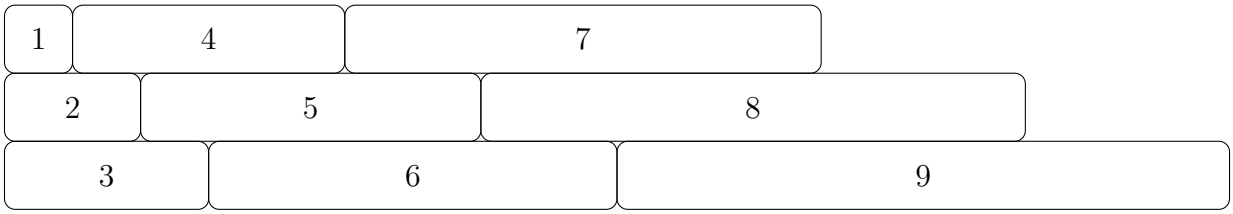


As an online deterministic algorithm, let us consider the following algorithm called LIST: we assign the next job in queue to the machine with the least actual load. (In case of a tie, let us schedule it to the one with the least possible index, for being deterministic.)

If we run LIST on the same input, we start with



(for this prefix the optimum solution would have a cost of 4 by the way) and we end up with



having a cost of 18. So for this particular instance, the competitive ratio is $\frac{18}{15} = 1.2$.

Another application of the Yao principle

Exercise

Give some nontrivial lower bound for the competitive ratio of the best possible randomized algorithm for the scheduling problem, with $m = 2$ machines.

Solution

Such statements can be attacked by the Yao principle. We have to pick some randomized input and study all the possible deterministic algorithms' outputs on them, lower-bounding the competitive ratio in each case, and return with the minimum of these lower bounds.

For starters, let us consider the following randomized input: we get the input $(1, 1)$ with a probability of $\frac{1}{2}$ and the input $(1, 1, 2)$ with a probability of $\frac{1}{2}$.

In general, when we apply the Yao principle, in the first it is advisable to

- pick exactly two input sequences $\boldsymbol{\sigma}_1$ and $\boldsymbol{\sigma}_2$ with some nonzero probabilities, $\frac{1}{2}$-$\frac{1}{2}$ suffices for the first try,
- such that $\boldsymbol{\sigma}_1$ is a prefix of $\boldsymbol{\sigma}_2$,

- while the optimal solution $\boldsymbol{\tau}_1$ greatly differs from the prefix of the optimal solution $\boldsymbol{\tau}_2$.

Since we are dealing against deterministic online algorithms, whatever $\boldsymbol{\tau}'_1$ is the output of such an algorithm A for the input $\boldsymbol{\sigma}_1$, that is also the prefix of its output for $\boldsymbol{\sigma}_2$. So generally, if an algorithm performs well on $\boldsymbol{\sigma}_1$, then it'll perform bad on $\boldsymbol{\sigma}_2$ and vice versa, giving on average some expected competitive ratio strictly greater than one.

Back to the current exercise, we determine the optimal costs for both cases: for $(1, 1)$, the optimal cost is 1 (we have to schedule the two jobs onto different machines), and for $(1, 1, 2)$, the optimal cost is 2 (we have to schedule the first two jobs onto the same machine and the third job to the other one). As the optimal output prefixes differ, it's a good candidate for showing a lower bound.

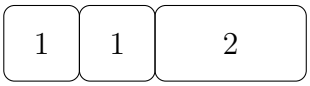
So the expected competitive ratio becomes

$$\frac{1}{2} \cdot \frac{A(1,1)}{1} + \frac{1}{2} \cdot \frac{A(1,1,2)}{2}.$$

Now we have to analyze all the possible online deterministic behaviors against this specific input set. If we choose the two inputs $\boldsymbol{\sigma}_1$ and $\boldsymbol{\sigma}_2$ such that $\boldsymbol{\sigma}_1$ is a prefix of $\boldsymbol{\sigma}_2$, then it suffices to take into account all the possible outputs for $\boldsymbol{\sigma}_2$ as their prefix of the appropriate length gives the output for $\boldsymbol{\sigma}_1$.

So we have a number of cases for an online algorithm A:

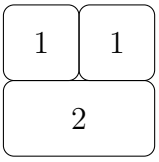
- If A schedules $(1, 1, 2)$ as

  then $A(1, 1, 2) = 4$ and $A(1, 1) = 2$, making the expected ratio

  $$\frac{1}{2} \cdot \frac{2}{1} + \frac{1}{2} \cdot \frac{4}{2} = 2.$$

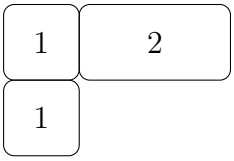
- If A schedules $(1, 1, 2)$ as

  then $A(1, 1, 2) = 2$ and $A(1, 1) = 2$ (as the first two jobs get scheduled onto the same machine), making the ratio

  $$\frac{1}{2} \cdot \frac{2}{1} + \frac{1}{2} \cdot \frac{2}{2} = \frac{3}{2}.$$

- If A schedules $(1, 1, 2)$ as

then $A(1,1,2) = 3$ while $A(1,1) = 1$, making the ratio

$$\frac{1}{2} \cdot \frac{1}{1} + \frac{1}{2} \cdot \frac{3}{2} = \frac{5}{4}.$$

Amongst these ratios, $\frac{5}{4}$ is the least one, so we proved that there is no randomized online algorithm for $m = 2$ machines having a competitive ratio better than $\frac{5}{4}$.

Exercise

Can you improve the lower bound you just got?

Solution

What we see in the previous calculation is that the minimum value is taken in a single case (and there were only finitely many cases). For many problems, this indicates that by changing the probability (while retaining the support of the input distribution) we can do better.

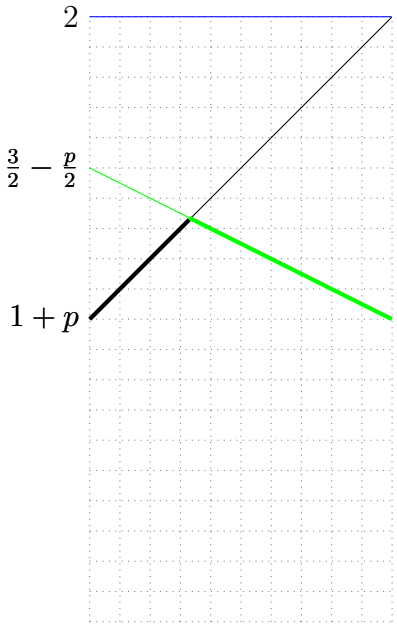
So what happens if we parametrize our input and say that $(1,1)$ is given with a probability of p, while $(1,1,2)$ is given by a probability of $1-p$?

Everything remains the same, we only have to change the $\frac{1}{2}$s to the corresponding symbols and get that

- in the first case, the expected ratio is $p \cdot 2 + (1-p) \cdot 2 = 2$,
- in the second one, it's $p \cdot 2 + (1-p) \cdot 1 = 1 + p$,
- and in the third case, it's $p \cdot 1 + (1-p) \cdot \frac{3}{2} = \frac{3}{2} - \frac{p}{2}$

Thus, we have to maximize $\min\left\{2, 1+p, \frac{3}{2} - \frac{p}{2}\right\}$ over $0 \leq p \leq 1$.

We can solve this by plotting the three functions



The minimum of these three functions is shown in boldface. Thus, the minimum gets maximized when $\frac{3}{2} - \frac{p}{2} = 1 + p$, that is, $p = \frac{1}{3}$. This choice (giving $(1, 1)$ with a chance of $\frac{1}{3}$ and $(1, 1, 2)$ with a chance of $\frac{2}{3}$) gives us the competitive ratio $1 + p = \frac{4}{3}$ as a lower bound. Since it's larger than the $5/4$ we had before, we improved the lower bound.

Or, instead of plotting the functions, we can solve this by a linear solver: the corresponding LP problem, if we introduce the new variable t for storing the maxmin is

$$\begin{aligned} 2 &\geq t \\ \frac{3}{2} - \frac{p}{2} &\geq t \\ 1 + p &\geq t \\ 0 \leq p &\leq 1 \\ \max t & \end{aligned}$$

Feeding it to our favorite LP solver gives us the optimal solution $p = 1/3$ and $t = 4/3$. Though in this case it might be an overkill to use that.

Exercise

Can you improve your bound even more?

Solution

We already optimized our distribution for the support consisting of $(1, 1)$ and $(1, 1, 2)$. In this case we might try to add a third input, which extends $\boldsymbol{\sigma}_2$ even further. This approach might or might not work, but we'll elaborate this. Say we extend our set of inputs by $\boldsymbol{\sigma}_3 = (1, 1, 2, 4)$. (On the very scientific base of "why not".)

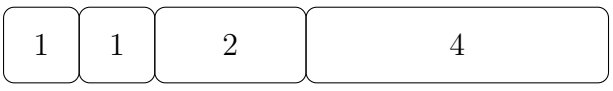
For this new input, the optimal cost is 4. Let p_1 be the probability of giving $(1, 1)$, p_2 be the chance of giving $(1, 1, 2)$ and $p_3 = 1 - p_1 - p_2$ being the chance for $(1, 1, 2, 4)$.

Then, the expected cost of an algorithm A is

$$p_1 \frac{A(1,1)}{1} \; + \; p_2 \frac{A(1,1,2)}{2} \; + \; p_3 \frac{A(1,1,2,4)}{4}.$$

As $\boldsymbol{\sigma}_3$ is the longest one, with all the others being a prefix of it, it suffices to analyze all the possible outputs for $(1, 1, 2, 4)$:

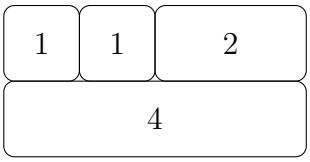
- If $(1, 1, 2, 4)$ gets scheduled as



  that gives us $A(1, 1) = 2$, $A(1, 1, 2) = 4$ and $A(1, 1, 2, 4) = 8$, yielding an expected ratio of

$$p_1 \frac{2}{1} \; + \; p_2 \frac{4}{2} \; + \; p_3 \frac{8}{4} = 2.$$

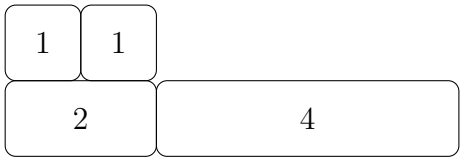
- If $(1,1,2,4)$ gets scheduled as



then $A(1,1)=2$, $A(1,1,2)=4$ and $A(1,1,2,4)=8$. Then the ratio is

$$p_1\frac{2}{1} \ + \ p_2\frac{4}{2} \ + \ p_3\frac{4}{4} = 2p_1+2p_2+p_3.$$

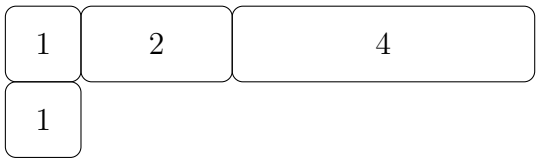
- If $(1,1,2,4)$ gets scheduled as



(or if 4 gets scheduled on the other machine) then $A(1,1)=2$, $A(1,1,2)=2$ and $A(1,1,2,4)=6$. Then the ratio is

$$p_1\frac{2}{1} \ + \ p_2\frac{2}{2} \ + \ p_3\frac{6}{4} = 2p_1+p_2+\frac{3}{2}p_3.$$

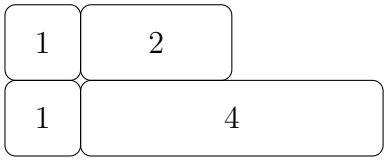
- If $(1,1,2,4)$ gets scheduled as



then $A(1,1)=1$, $A(1,1,2)=3$ and $A(1,1,2,4)=7$. Then the ratio is

$$p_1\frac{1}{1} \ + \ p_2\frac{3}{2} \ + \ p_3\frac{7}{4} = p_1+\frac{3}{2}p_2+\frac{7}{4}p_3.$$

- Finally, if $(1,1,2,4)$ gets scheduled as



then $A(1,1)=1$, $A(1,1,2)=3$ and $A(1,1,2,4)=5$. Then the ratio is

$$p_1\frac{1}{1} \ + \ p_2\frac{3}{2} \ + \ p_3\frac{5}{4} = p_1+\frac{3}{2}p_2+\frac{5}{4}p_3.$$

Right now, firing up the LP solver method does not seem to be an overkill at all. The generated LP problem is:

$$\begin{aligned} t &\le 2 \\ t &\le 2p_1 + 2p_2 + p_3 \\ t &\le 2p_1 + p_2 + \frac{3}{2}p_3 \\ t &\le p_1 + \frac{3}{2}p_2 + \frac{7}{4}p_3 \\ t &\le p_1 + \frac{3}{2}p_2 + \frac{5}{4}p_3 \\ p_1 + p_2 + p_3 &= 1 \\ p_i &\ge 0 \\ \max\ t & \end{aligned}$$

We might note that the first constraint is redundant as the second one is always smaller, and the fourth is also redundant due to the presence of the fifth. Either way, if we enter this system or the reduced one into an LP solver, we get the optimal solution at $p_1 = 1/3$, $p_2 = 2/3$, $p_3 = 0$ and $t = 4/3$.

Which means that with this extension of the support, we could not improve the ratio.

Exercise

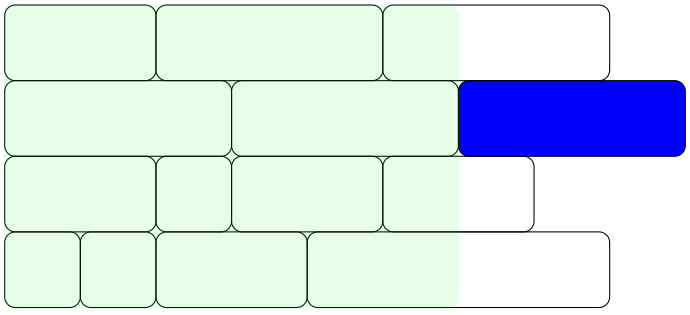
Do the math (definitely with the help of an LP solver for the last phase) for the input sequence $(1, 1, 2, 4, 8)$ (that is, p_1 for $(1, 1)$, p_2 for $(1, 1, 2)$, p_3 for $(1, 1, 2, 4)$ and p_4 for $(1, 1, 2, 4, 8)$.

You should end up with a result of $\frac{7}{5} = 1.4$, which is an improvement.

The LIST algorithm is $\left(2 - \frac{1}{m}\right)$-competitive

The above 1.4 we get after solving the four-variable system is not so far[8] from an upper bound: in this section we show that the LIST algorithm is actually $\left(2 - \frac{1}{m}\right)$-competitive, which is 1.5 if $m = 2$.

Let us consider a schedule produced by the LIST algorithm. Let the ℓth job be one of those finishing latest (on the figure below, this job is filled in blue).



[8] Well... being far is relative, I admit

Now if S_ℓ is the starting time of this job, then we can be sure in that when the LIST algorithm assigns job ℓ to its machine, all the machines have a load at least S_ℓ (since if any of them would have a load less than S_ℓ, LIST would place the job onto that machine). Hence the area of size $m \times S_\ell$ is already completely filled when we place job j onto the track, that is,

$$m \cdot S_\ell \leq \sum_{i<\ell} p_i$$

and of course, since job ℓ finishes last, if L denotes the cost of the schedule produced by LIST, then we also have

$$L = S_\ell + p_\ell.$$

We also have the lower bounds

$$\frac{\sum_{i=1}^{n} p_i}{m} \leq L^* \qquad\qquad p_\ell \leq L^*$$

for the cost of the optimal solution L^*.

Putting all the pieces together, we get that

$$L = S_\ell + p_\ell \leq \frac{\sum_{i<\ell} p_i}{m} + p_\ell = \frac{\sum_{i\leq\ell} p_i}{m} + \frac{m-1}{m} p_\ell \leq L^* + \frac{m-1}{m} L^* = \left(2 - \frac{1}{m}\right) L^*,$$

proving the upper bound.

Exercise

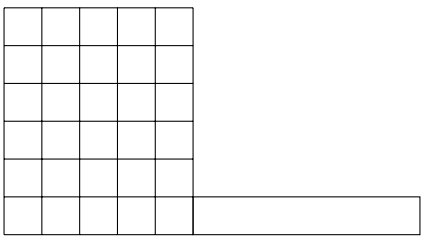
Show that if $p_j \leq \frac{\sum_{i=1}^{n} p_i}{2m}$ holds for each job p_j, then LIST is $\left(\frac{3}{2} - \frac{1}{2m}\right)$-competitive.

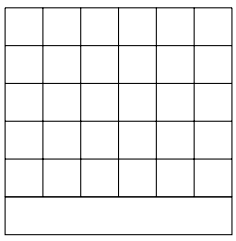
Solution

Since $\frac{\sum_{i=1}^{n} p_i}{m} \leq L^*$, this condition ensures $p_\ell \leq \frac{L^*}{2}$ in the proof above, making the chain to

$$L = S_\ell + p_\ell \leq \frac{\sum_{i<\ell} p_i}{m} + p_\ell = \frac{\sum_{i\leq\ell} p_i}{m} + \frac{m-1}{m} p_\ell \leq L^* + \frac{m-1}{m} \cdot \frac{1}{2} L^* = \left(\frac{3}{2} - \frac{1}{2m}\right) L^*.$$

For proving the lower bound, consider an input sequence of $m \times (m-1)$ jobs of unit cost, followed by a job of a cost of m. For the case of $m = 6$, the output of LIST (left) and the optimal solution (right) are depicted below:





In general, LIST produces a solution of cost $2m - 1$ while the optimal cost is m, hence this example shows the matching lower bound $\frac{2m-1}{m} = 2 - \frac{1}{m}$.

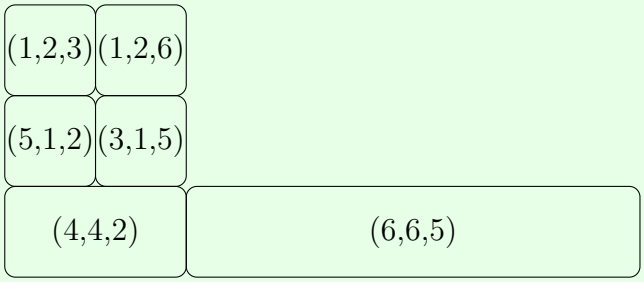
Scheduling variant: unrelated machines

A possible extension of the scheduling problem is the following: we have m machines, jobs are arriving, this time job i defines a processing time $p_{i,j}$ for each machine $1 \leq j \leq m$. For example, if we job with processing time vector $(1, 2, 4)$ arrives, this means that on the first machine it takes 1 time unit to process the job, if we choose the second, then it takes 2 time units, and on the third it takes 4 time units.

As an example, if $m = 3$ and the input sequence is

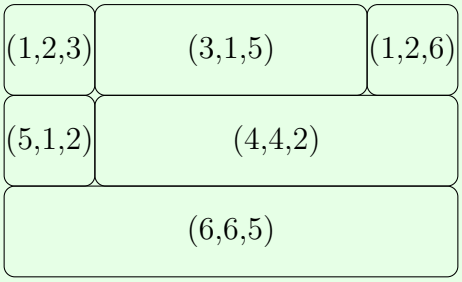
$$((1, 2, 3), (5, 1, 2), (4, 4, 2), (3, 1, 5), (1, 2, 6), (6, 6, 5))$$

then a possible schedule is



with a total cost of 7 (we get this one if we assign job i in a greedy way to the machine for which $p_{i,j}$ is minimal).

We can do better:



is an optimal solution with a total cost of 5.

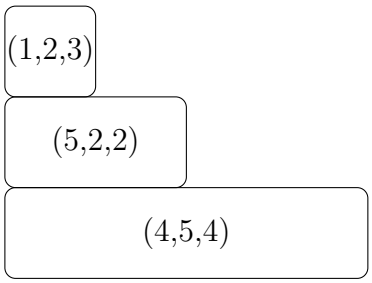
For this variant, two lower bounds for the optimal value L^* are:

$$\max_i \min_j p_{i,j} \leq L^* \qquad \frac{\sum_i \min_j p_{i,j}}{m} \leq L^*$$

The first one comes from the fact we have to schedule each single job, and job i takes at least $\min p_{i,j}$ time units to process (this happens in the example above with the job $(6, 6, 5)$: 5 is thus a lower bound for the optimal cost). For the second, the nominator sums the total time units we have to spend with work for processing all the jobs in the lucky case when each job can be scheduled to the machine with solves it fast (this is what we have done in the first example: that way, the total time needed was 11), and if we are extremely lucky, then it happens so that there are no gaps and all the machines will have the same load (we were not that lucky in the first example, there were huge gaps on the first two machines), proving the other lower bound.

For this variant, the LIST algorithm is the following: we check what the load would be on each machine if we would assign the current job there; and we choose the machine having the least load.

For example, consider the following configuration:



If now we get the job $(4, 2, 1)$, then the load of machine 1 would increase to 5, the load of machine 2 would increase to 4, and the load of machine 3 would increase to 5. Since 3 is the least amongst these possibilities, we schedule $(4, 2, 1)$ onto machine 2. (Observe that this is not the machine which could finish the job in the least time.)

Proposition

The LIST algorithm is m-competitive for this variant of the scheduling problem.

Proof: Upper bound

Let us consider the run of the LIST algorithm on a queue of m jobs.

For each $1 \leq i \leq n$, let $L(i)$ be the makespan after processing the first i jobs. We prove by induction that

$$L(k) \leq \sum_{i \leq k} \min_j p_{i,j}$$

holds for each k. For $k = 0$, both sides are zero, so it's fine. Now assume the claim holds for k and let us step to $k + 1$. We claim that $L(k + 1) \leq L(k) + \min_j p_{k+1,j}$. Indeed: let j^* be the index so that p_{k+1,j^*} is minimal. Then, the load on the machine j^* is at most $L(k)$ before scheduling job $k + 1$ (as $L(k)$ is the maximum load at that moment), so the load on that machine would be at most $L(k) + \min_j p_{k+1,j}$, and possibly even less on some other machine. So after scheduling job $k + 1$ the load on its machine is at most $L(k) + \min_j p_{k+1,j}$. If this is a machine at this moment having the maximal load, then we get $L(k + 1) \leq L(k) + \min_j p_{k+1,j}$. Otherwise, if this machine has a non-maximal load at this moment, then some other machine has; but that machine has the same load as before, so $L(k + 1) = L(k)$ in that case (which also entails $L(k + 1) \leq L(k) + \min_j p_{k+1,j}$, so the statement is proved).

Thus, plugging in n in place of k we get

$$L = L(n) \leq \sum_{i \leq n} \min_j p_{i,j} \leq m \cdot L^*,$$

as needed to get the upper bound.

Proof: Lower bound

Let $\epsilon > 0$ be some arbitrarily small value and let us consider the following sequence of jobs: $\boldsymbol{p}_1 = (1 + \epsilon, 1, m, m, \ldots, m)$, $\boldsymbol{p}_2 = (m, 1 + \epsilon, 2, m, m, \ldots, m)$, $\boldsymbol{p}_3 = (m, m, 1 + \epsilon, 3, m, m, \ldots, m)$, $\ldots$, $\boldsymbol{p}_{m-1} = (m, m, m, \ldots, 1 + \epsilon, m - 1)$, $\boldsymbol{p}_m = (m, m, \ldots, m, 1 + \epsilon)$.

In general, $p_{i,i} = 1 + \epsilon$, $p_{i,i+1} = i$ and the other entries are m.

For this input, $1+\epsilon$ is an optimal solution: we schedule job i to machine i for each i.

The LIST algorithm puts first job to the second machine as that has a cost of 1. For the second job, the second machine would now be loaded till $2+\epsilon$ while the third would be loaded only till 2, so job 2 gets assigned to machine 3. Then, job 3 gets assigned to machine 4 (loads are $3+\epsilon$ vs. 3), and so on, while in the final step, job m gets assigned to the first machine, the makespan becoming m, showing a lower bound of m for competitiveness as ϵ tends to zero.

Sheduling variant: Related parallel machines

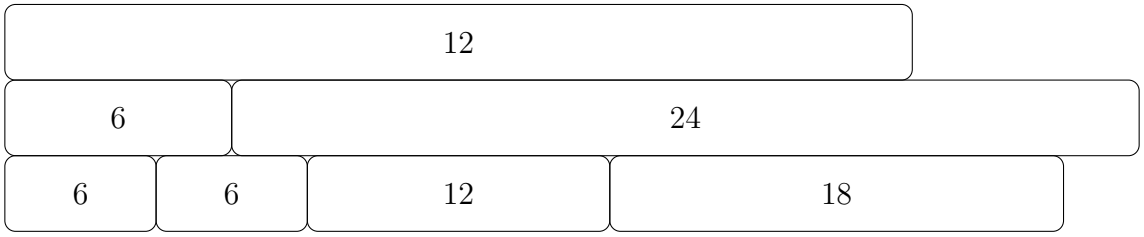
In this variant, we have m machines, each machine j having a speed $v_j > 0$. The jobs themselves again have a single processing cost p_i. Processing job i on machine j takes $\frac{p_i}{v_j}$ time. Again, the LIST algorithm schedules the job to the machine which would have the least load after getting the job.

Exercise

Run the LIST algorithm for the case of $m = 3$, $\boldsymbol{v} = (1, 2, 3)$, and input sequence $6, 6, 6, 12, 24, 18, 12$. Can you give an optimal solution?

Solution

We end up with the following configuration after running LIST (the bottom machine being the fastest):



First we schedule 6 onto the fastest machine, its load being $6/3 = 2$ then, then the second 6 goes on the second machine, its load becoming $6/2 = 3$, then the third job of size 6 goes again to the third machine, its load becoming 4, then the 12 goes again to the third machine, its load becoming 8 (on machine 2 the load would be 9 while it would be 12 on the first), and so on, the makespan is now 15.

This is, in fact, optimal: as all the number are divisible by 6, if we want to go below 15, then on the first machine we can only process 12 units, so that's already optimal. Now if on the second machine we go below 15, then we can process at most 24 units, in which case the load on the third machine would grow to at least 16. So this 14 is an optimal solution.

We again have two lower bounds for the optimum:

$$\frac{\max p_i}{\max v_j} \le L^* \qquad\qquad \frac{\sum p_i}{\sum v_j} \le L^*$$

Indeed: the largest job of size $\max p_i$ has to be processed on a machine, which takes at least $\frac{\max p_i}{\max v_j}$ time even on the fastest one, showing us the first bound, and for the second one, observe

that the total amount of work all the machines can do is $\sum v_j$ per time unit, thus if all of them are evenly loaded, the total makespan is the one given in the second bound.

We will show in this section that LIST is $\Theta(\log m)$-competitive in this setting.

Proof: Lower bound

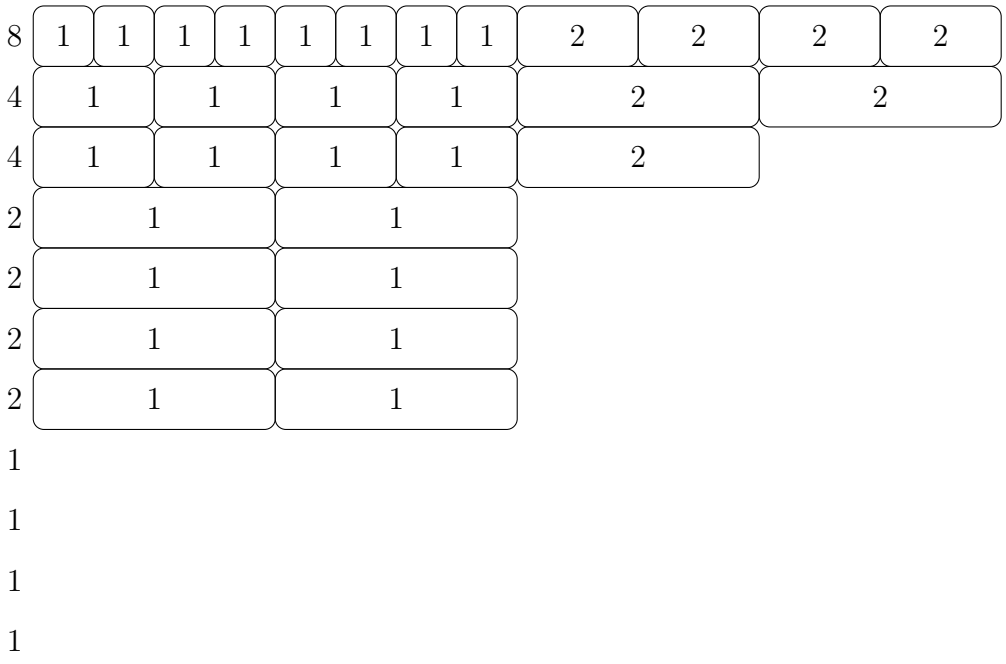
For the lower bound we construct an input of the following form:

- First, we construct the input with respect to a parameter k. Now the number m of machines and the competitive ratio c will both be functions of k so that as $k \to \infty$, m will also tend to infinity (thus the argument works for an arbitrary number of machines) and c will be $\Omega(\log m)$.
- Jobs arrive in increasing order, so that $p_1 \leq p_2 \leq \ldots p_m$ for some m. Intuitively, these arrangements tend to ruin LIST's performance in general.
- There are m machines with speeds $v_1 = p_m$, $v_2 = p_{m-1}$, $v_3 = p_{m-2}$, $\ldots$, $v_m = p_1$. This way, the optimal cost is one: we have to assign p_i to machine $m - i + 1$.
- It's easier to analyze the behavior of the LIST algorithm if everything is an integer and divisibility is not a problem. So the size of the jobs (and the speed of the machines) will be powers of two.

Thus, we will have M_0 machines of speed $2^0 = 1$, some number M_1 of machines of speed 2^1, in general, M_i machines of speed 2^i for $0 \leq i \leq k$. Also, let $M_k = 1$.

We construct the numbers of machines to make sure that when the subsequence of jobs of 2^i begins, then for each $j > i$, the machines of speed 2^j have a load of exactly i, and for each $j \leq i$, the machines of speed 2^j have a load of exactly j.

For an example, see the Figure below for $k = 3$ (with the number M_i of the machines being incorrect yet). The Figure shows the moment after we scheduled all the jobs of sizes 1 and 2, and the jobs of size 4 (which will be scheduled to the first machine only) are yet to come.



In order for the construction to work, we have to ensure that the jobs of size 1 exactly load the machines of speed at least 2, the jobs of size 2 exactly load the machines of speed at least 4 and so on. Since the jobs of size 2^i should be the same as M_i to make the optimal cost one, we have $M_{k-1} = 2 \cdot M^k$ (since the 2 jobs of size 2^{k-1} can fill the machine of speed 2^k for a load of one), $M_{k-2} = 4 \cdot M_k + 2 \cdot M_{k-1}$ and so on; in general,

$$M_{k-t} = 2^t \cdot M_k + 2^{t-1} \cdot M_{k-1} + \ldots + 2^2 \cdot M_{k-1+2} + 2^1 \cdot M_{k-t+1}.$$

Simpligfíing the terms but the last one by 2 we get

$$M_{k-t} = 2 \cdot \left(2^{t-1} \cdot M_k + 2^{t-2} \cdot M_{k-1} + \ldots + 2^1 \cdot M_{k-1+2}\right) + 2^1 \cdot M_{k-t+1},$$

which is $2 \cdot M_{k-t+1} + 2 \cdot M_{k-t+1} = 4 \cdot M_{k-t+1}$ if $t > 1$ since in that case the inner factor also has this general form.

So we get that if we have $M_k = 1$ machine of speed 2^k, $M_{k-1} = 2$ mahcines of speed 2^{k-1}, $M_{k-2} = 8$ machines of speed 2^{k-2}, and so on (with the number of machines being 32, 128 and so on, each time quadrupling the previous number), then for that input the LIST algorithm has a cost of k, while the optimum is 1. Expressing k in terms of the number m of machines we get that $m = 1 + 2 + 2 \cdot 4 + 2 \cdot 4^2 + 2 \cdot 4^3 + \ldots + 2 \cdot 4^{k-1} = 1 + 2\frac{4^k - 1}{4-1}$ and so $k = \Theta(\log m)$, thus for this particular input sequence, LIST is an $\Omega(\log m)$-competitive algorithm.

For the asymptotically matching bound we prove a couple of lemmas first. Again, let L^* stand for the cost of the optimal solution and L stand for the cost of the solution produced by LIST.

Proposition

After running LIST, the fastest machine has a load of at least $L - L^*$.

Proof

Let us consider the machine having the highest load. If this is the fastest machine, then its load is L, which is of course at least $L - L^*$ and we are done. Otherwise, let p_ℓ be the last task on this machine, let v_j and $v_{\max}$ respectively stand for the speed of this machine and the fastest one, and let

Clearly, we have $\frac{p_j}{v_{\max}} \leq L^*$ (since the left-hand-side is always a lower bound for the optimal cost).

Since LIST chose machine j, we get that $L \leq \text{load(max)} + \frac{p_j}{v_{\max}} \leq \text{load(max)} + L^*$, rearranging this constraing we indeed get $\text{load(max)} \geq L - L^*$.

Proposition

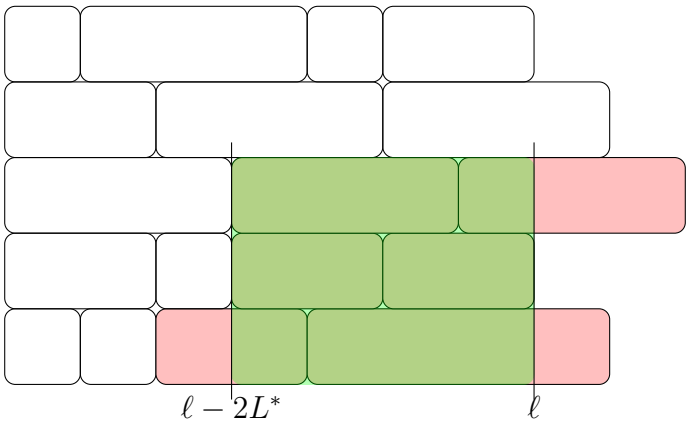
If after running LIST all the machines of speed at least v have a load at least ℓ, then all the machines of speed at least $v/2$ have a load at least $\ell - 4L^*$.

Proof

If the minimum load of the "fast" machines (let the machines of speed at least v be called "fast", and the machines of speed between $v/2$ and v be called "mid" machines) ℓ is at most $4L^*$, then there is nothing to show (as the claim states that the load is nonnegative

in that case, which clearly holds). So assume $\ell \geq 4L^*$. In this case, $\ell - 2L^*$ is a positive number.

Let us consider all the jobs that are scheduled on a fast machine, start before ℓ and end after $\ell - 2L^*$, that is, have a nonempty intersection with the "rectangle" indicated on the Figure below. The jobs in question are marked; the last three row of the Figure are the fast machines.



It is clear that in an optimal solution at least one of these jobs is scheduled on a non-fast machine (as if all of these were scheduled to a fast machine, then the makespan is necessarily at least $2L^*$, since these jobs completely fill the green rectangle). So let the jth job be one such job. Since job j of size p_j is scheduled in some solution (having makespan L^*) to a machine of speed at most v, we get that $\frac{p_j}{v} \leq L^*$. Thus, on a machine of speed at least $v/2$, the processing of job j takes at most $2 \cdot L^*$ time units. Now since LIST scheduled this job onto a fast machine instead, on which the load became at least $\ell - 2L^*$ (since job j intersects the work rectangle indicated on the Figure), we get that on each of the mid-speed machines having load ℓ' and speed $v' \geq v/2$, the inequality $\ell' + \frac{p_j}{v'} \geq \ell - 2L^*$ holds. Thus, knowing $\frac{p_j}{v'} \leq 2L^*$ and rearranging we get $\ell' \geq \ell - 4L^*$, as claimed.

Having these two Lemmas we are able to prove a $\Theta(\log m)$-competitivity:

Proposition

LIST is $\Theta(\log m)$-competitive on parallel related machines.

Proof

In this proof, let us call the machines of speed at least $\frac{v_{\max}}{m}$ fast, and the others slow.

By the first lemma we know that after running LIST, the machine(s) of speed $v_{\max}$ have a load of at least $L - L^*$. Applying the second lemma we get that the machines of speed at least $\frac{v_{\max}}{2}$ have a load of at least $L - 5L^*$. Applying the second lemma again, machines of speed at least $\frac{v_{\max}}{4}$ have a load of at least $L - 9L^*$, in general, machines of speed at least $\frac{v_{\max}}{2^k}$ have a load of at least $L - (4k+1)L^*$. Iterating this speed-halving $\lceil \log m \rceil$ times, we get that machines of speed at least $\frac{v_{\max}}{2^{\lceil \log m \rceil}} \leq \frac{v_{\max}}{m}$ have a load of at least $L - (1 + 4\lceil \log m \rceil)L^*$ – that is, the load is at least $L - (1 + 4\lceil \log m \rceil)L^*$ on each fast machine, so the total work carried out by these machines only is at least

$$\Big(\sum_{j:j \text{ fast}} v_j \Big) \Big(L - (1 + 4\lceil \log m \rceil)L^* \Big).$$

Now let us compute the total work W carried out by the machines in an optimal schedule. In this case, each machine has a load of at most L^*, and thus the total work is at most $\Big(\sum_j v_j\Big)L^*$. We can split the sum into two sums, separately summing the work carried out by the slow and the fast machines respectively and we get

$$W \;\leq\; \Big(\sum_j v_j\Big)L^* \;\leq\; \Big(\sum_{j:j \text{ slow}} v_j\Big)L^* + \Big(\sum_{j:j \text{ fast}} v_j\Big)L^*.$$

Now as the slow machines have speed at most $\frac{v_{\max}}{m}$ by the definition of being slow and there at most m slow machines (as there are m machines at all), we get that $\Big(\sum_{j:j \text{ slow}} v_j \leq \frac{v_{\max}}{m} \cdot m = v_{\max}$ so

$$W \;\leq\; v_{\max} \cdot L^* + \Big(\sum_{j:j \text{ fast}} v_j\Big)L^*.$$

Of course the fastest machine is fast, so $v_{\max} \leq \Big(\sum_{j:j \text{ fast}} v_j\Big)$, yielding

$$W \;\leq\; 2 \cdot \Big(\sum_{j:j \text{ fast}} v_j\Big)L^*.$$

Putting the lower and the upper bounds for the total work we get

$$\Big(\sum_{j:j \text{ fast}} v_j\Big)\Big(L - (1 + 4\lceil \log m \rceil)L^*\Big) \;\leq\; 2 \cdot \Big(\sum_{j:j \text{ fast}} v_j\Big)L^*,$$

rearranging which we get

$$L \;\leq\; \big(3 + 4\lceil \log m \rceil\big)L^*,$$

showing $\Theta(\log m)$-competitiveness.

Scheduling variant: the time model

In this variant, each job j has a processing time p_j and an arrival time t_j. Time ticks in a continuous way and we have no knowledge about the job (p_j, t_j) before t_j. A sample input for $m = 2$ machines is $(2, 0),\ (2, 0),\ (3, 1)$, that is, two jobs of processing time 2 are known already in the beginning and a job of processing time 3 arrives after 1 unit of time passes. Clearly, we can decide at time point 0 to schedule both of the jobs, then when job $(3, 1)$ arrives, we are forced to wait till time point 2 and then we can start $(3, 1)$, yielding a makespan of 5. The optimal solution for this case would schedule only one of the $(2, 0)$ jobs, wait till time point 1, then schedule $(3, 1)$ to the free machine and $(2, 0)$ to time point 2, as soon as the first $(2, 0)$ finishes. Hence the optimal solution has a makespan of 4. (See the Figure below.)

So it might have a benefit to wait in this setting. Clearly, waiting too long is another costly mistake. This particular example shows that for $m = 2$ machines, those algorithms that schedule both of the two $(2, 0)$ jobs before time point $t = 1$ cannot have a competitive ratio

beating 1.25. On the other hand, if an algorithm waits till $t = 1$ time units before scheduling the second $(2, 0)$ task, it will have a cost of at least 3 on the single input $(2, 0),\ (2, 0)$, for which the optimal solution has a makespan of 2, making the competitve ratio of at least 1.5. This proves that there is no deterministic online algorithm having a competitive ratio better than 1.25 for $m = 2$ machines.

Also, if we have m machines, then m jobs of $(2, 0)$ are either all scheduled before $t = 1$ (in which case one job $(3, 1)$ makes the ratio at least 1.25) or not (in which case the ratio becomes at least 1.5), so this result holds for an arbitrary number of machines.

Modifying this approach again, we can play against all the determistic algorithms with inputs tailored to them: say, if the last $(2, 0)$ is scheduled at $t < 1$, then we can throw in a job $(4-t, t)$ as well instead of the $(3, 1)$. This would make the ratio at least $\frac{6-t}{4} = 1.5 - \frac{t}{4}$ for this input, while the $(2, 0)$ jobs alone give a ratio of $\frac{2+t}{2} = 1 + \frac{t}{2}$. Parametrizing this the usual way, we get that the best possible lower bound is achieved where $1.5 - \frac{t}{4} = 1 + \frac{t}{2}$, that is, $t = \frac{2}{3}$ for which the ratio becomes $4/3$.

Hence we proved:

Proposition

There is no deterministic online algorithm for any number m of machines for the scheuling problem in the time model, which has a competitive ratio better than $4/3$.

By a somewhat more involved bad input construction, and a lengthier case analysis one can prove also the following:

Proposition

There is no deterministic online algorithm for any number m of machines for the scheuling problem in the time model, which has a competitive ratio better than 1.3473.

One competitive algorithm in this setting is the so-called INTV algorithm:

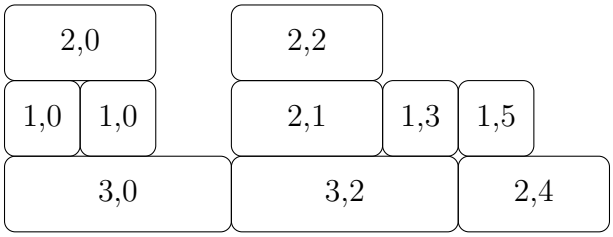
1. (Collecting phase.) We wait till the least time point T such that all the machines are idle and there are unscheluded jobs.
2. (Distribution.) Let X be the set of unscheluded jobs at time point T. Let us compute an optimal scheduling for X starting at time point T, assign them onto the machines, and go back to the collecting phase.

As an example, let $m = 3$ and let us consider the input sequence

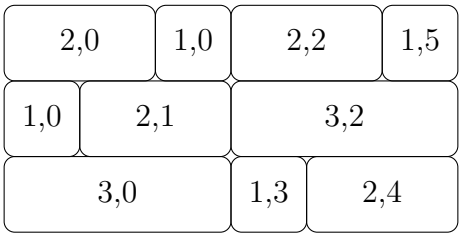
$$(1, 0),\ (1, 0),\ (2, 0),\ (3, 0),\ (2, 1),\ (2, 2),\ (3, 2),\ (1, 3),\ (2, 4),\ (1, 5).$$

At time point $T = 0$, there are four unscheduled jobs and all the machines are idle. So we compute an optimal schedule for these four jobs (see the Figure below), and wait till $T = 3$, when all the machines become idle again. Till $T = 3$, the jobs $(2, 1),\ (2, 2),\ (3, 2)$ and $(1, 3)$ are accumulated, so at $T = 3$ we schedule them optimally to the three machines. Now the current makespan is $T = 6$, so we wait till $T = 6$ and then we distribute the jobs $(2, 4)$ and $(1, 5)$ as

well.



So in this case, INTV achieves a makespan of 8. Clearly, $\max\{p_j + t_j\}$ is a lower bound for the optimum: as there is a job $(2, 4)$, we cannot finish processing it before $T = 6$. In fact, this is possible, so 6 is the optimum, see the Figure below.



However, INTV has a serious drawback: computing an optimal offline scheduling at the distribution points is a hard[9] problem. Instead of insisting to an optimal solution locally, in practice we use an α-approximation algorithm for some constant α: such an algorithm can be implemented to run in a short[10] amount of time and the solution computed by the algorithm has at most α times the cost as the optimal one.

For the offline variant of the scheduling problem, consider the following algorithm Longest Processing Time, or LPT:

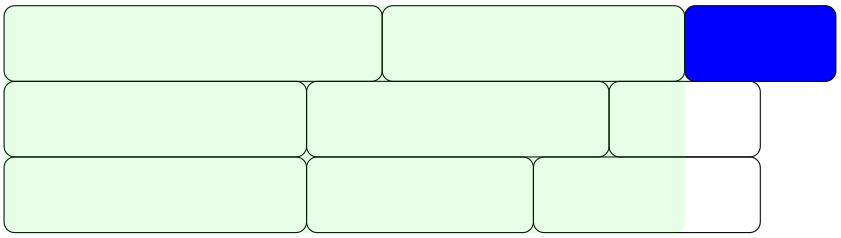
- We first sort the input sequence in decreasing order with respect to processing time,
- then we apply the LIST algorithm on the input in this particular order.

Proposition

LPT is a 4/3-approximation algorithm.

Proof

Let $m > 1$ be the number of machines. Assume to the contrary that $p_1 \geq p_2 \geq \ldots \geq p_n$ is some input sequence (already sorted in decreasing order) such that $L > \frac{4}{3}L^*$ holds, where L^* denotes the cost of the optimal solution and L denotes the cost of LPT. Let us choose this sequence to be a shortest possible one.



Then, LPT schedules p_n so that its ending time is L and p_n is the only such job: otherwise,

[9] a so-called "**NP**-hard"

[10] read as: polynomial instead of exponential

for the sequence $p_1, \ldots, p_{n-1}$ we would get a possibly smaller optimal cost while the same LPT cost, so that would be a shorter counterexample.

Then, by the same reasoning as before (see the Figure above) we get that

$$L^* \geq \frac{\sum_{i=1}^{n} p_i}{m} \geq \frac{m(L - p_n) + p_n}{m} \geq L - p_n.$$

Together with the assumption $L > \frac{4}{3}L^*$ this yields $p_n > L^*/3$. Since p_n is the smallest job, we get that all the jobs are of size at least $L^*/3$.

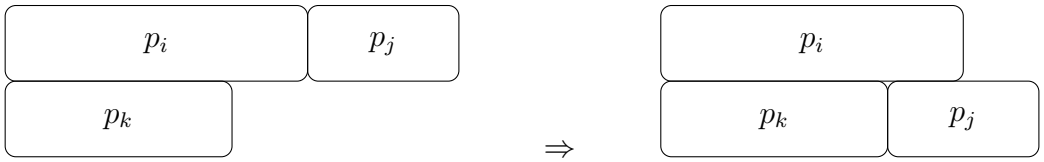
Now let us consider an optimal schedule. We will transform it so that in each step the makespan remains unchanged and after a number of steps, we arrive to the schedule produced by LPT, which implies that LPT produces an optimal solution if $p_n > L^*/3$.

In an optimal solution (having a makespan of L^*), there can be at most two jobs on each machine (as three jobs of size greater than $L^*/3$ would make the load larger than L^*).

Let us consider the following two transformations:

1. If on a machine there are two jobs p_i and p_j with $p_i < p_j$, let us change their order:

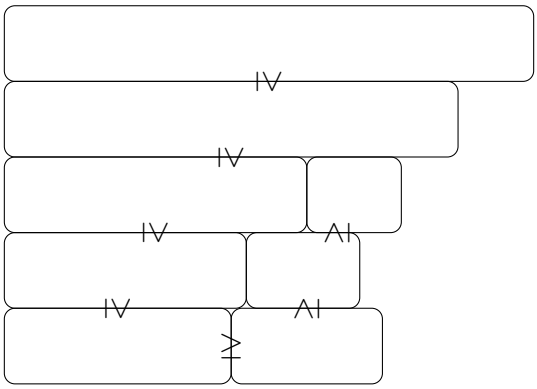
2. If on a machine there are two jobs $p_i \geq p_j$ and on another one there is a single job p_k with $p_k < p_i$, let us put p_j to the other machine instead:



3. If on a machine there are two jobs $p_i \geq p_j$ and on another one there are two jobs $p_k \geq p_\ell$ with $p_i \geq p_k$ and $p_\ell < p_j$, then let us swap p_j and p_ℓ:

Neither of the above transformations increases the maximum load on the machines involved, so the solution remains optimal after applying any of them. Actually, it can be shown that after a finite number of steps (possibly reordering the machines as well) one

arrives to a schedule depicted on the figure below:



It is easy to see that this schedule is the one produced by the LPT algorithm: clearly, LPT distributes the first m (largest) jobs to the machines, one job for each machine, then the last $n-m$ jobs fill the machines "bottom-up". Thus, if $p_n > L^*/3$, then LPT produces an optimal solution, so L cannot be larger than $\frac{4}{3}L^*$ is this case either.

On the performance of INTV we can show the following:

Proposition

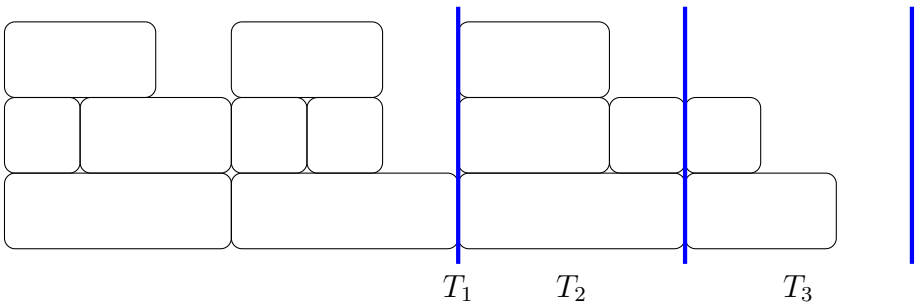
If we use an offline α-approximation algorithm in the distribution phase of INTV, then we get a $(2 \cdot \alpha)$-competitive algorithm for online scheduling in the time model.

Hence,

- INTV is 2-competitive
- if we use LPT during the distribution phase, then the resulting INTV+LPT algorithm is 8/3-competitive.

Proof

Let us consider an output of the INTV algorithm in which we use an offline α-approximation heuristic in the distribution phase. Let T_3 stand for the length of the last phase, T_2 stand for the length of the previous one, and let T_1 be the start of the phase before the last (see the Figure below).



Then, the cost of the INTV algorithm is $T_1 + T_2 + T_3$. Observe that all the jobs that got scheduled in the last phase arrive after T_1. Thus, only to schedule these jobs we have to use a makespan of at least $T_1 + T_3'$ where T_3' is the optimal cost of scheduling all these jobs without timestamps (that is, in the list model). As the algorithm uses an α-approximating

offline sub-scheduler, we have $T_3 \leq \alpha \cdot T'_3$.

Similarly, only for scheduling those items that got scheduled in the phase before the last one, we have to use a makespan of at least T'_2, the optimal offline cost of scheduling them without timestamps and again, we have $T_2 \leq \alpha \cdot T'_2$.

Putting these together we get

$$\begin{aligned} L^* = \; & T_1 + T_2 + T_3 \;\leq\; (T_1 + T_3) + T_2 \;\leq\; (T_1 + \alpha \cdot T'_3) + \alpha \cdot T'_2 \\ \leq \; & \alpha \cdot (T_1 + T'_3) + \alpha \cdot T'_2 \;\leq\; 2\alpha \cdot L. \end{aligned}$$

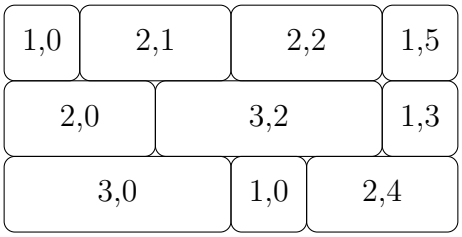
Another, and even better and conceptually simpler algorithm is the so-called ONLINE LPT algorithm which does not work in phases:

- As soon as there is an idle machine and at least one unscheduled job, let us schedule a longest job onto an idle machine.

Running the ONLINE LPT for our previous example

$$(1,0),\ (1,0),\ (2,0),\ (3,0),\ (2,1),\ (2,2),\ (3,2),\ (1,3),\ (2,4),\ (1,5)$$

on $m = 3$ machines again (see the Figure below), at time $T = 0$ we have three idle machines so we schedule the three longest jobs $(3,0)$, $(2,0)$ and one of the $(1,0)$s, then we wait. At $T = 1$ one machine becomes idle and we have an $(1,0)$ job and a $(2,1)$ job to be scheduled so we put the longer one, $(2,1)$, to the idle machine and wait. At $T = 2$ the second machine becomes idle and we have three jobs, $(1,0)$, $(2,2)$ and $(3,2)$, so we assign $(3,2)$ to the second machine. Then at $T = 3$ we have the jobs $(1,0)$, $(2,2)$ and $(1,3)$ and two machines, so we put $(2,2)$ and $(1,0)$ to the two idle machines. At $T = 4$ the longest job $(2,4)$ is assigned to the idle machine and finally, at $T = 5$ we put the last two jobs onto the two idle machines, yielding (for this particular input) an optimal solution.



The performance of ONLINE LPT is much harder to analyze but is possible:

Proposition

The ONLINE LPT algorithm is 3/2-competitive.

The List Access problem

The (most basic version of the) List Access problem is the following: we have a (singly-linked) list of items of length m. For convenience, we set the list initially to be $(1, 2, 3, \ldots, m)$. A request is a number $p \in \{1, \ldots, m\}$. The cost of serving p is the current position of p in the list (this models the traversal cost of the list when seeking for p).

Before handling the next request, we are allowed to do the following operations: i) we can swap two adjacent elements in the list for a cost of 1, and ii) we can move the last requested element p into the front of the list, for free.

The objective is to minimize the total combined cost paid.

For example, when $m = 3$ so the initial list is $(1, 2, 3)$, and the request sequence is $3, 2, 3, 2$, then

- if we do not move the elements at all, then we pay 3 for querying 3 since that's on the third position, and pay 2 for the item 2, being the third element of the list. As both elements are queried twice, we pay $3 + 2 + 3 + 2 = 10$ in total.
- A better approach is to move the 3 to front after its first query. Then, after the first query our list is $(3, 1, 2)$ and we payed a cost of 3 so far. Now if we move 2 to the front as well after serving the next query we end up with the list $(2, 3, 1)$ and we spent 6 so far, which is at this point more than the original cost of 5 so far; but it pays off as handling the next two requests $3, 2$ we only pay an additional cost of $2 + 1 = 3$, yielding a total cost of 9.
- An even better approach is to use non-free swaps in the beginning: swapping the first two elements we get the list $(2, 1, 3)$, then swapping the last two elements we have $(2, 3, 1)$ and so far we have a cost of 2. Now serving all the requests we pay an additional cost of $2 + 1 + 2 + 1 = 6$, thus we only spend 8.

The reader can verify by a case analysis that these values are optimal for the given set of operations, thus it might worth to spend a cost for the swaps.

Of course one may introduce more modifier operations, like "sorting the list for some reasonable cost", "reverse the list" etc for some nonnegative cost. However, for all of these cases it is easy to show a lower bound:

Proposition

In any model of the List Access problem in which we are able to reverse the list for some possibly nonzero cost, there is no deterministic online algorithm with a competitive ratio better than 2.

Proof

For any deterministic online algorithm A, the worst possible input is the one which always requests the last element of the list. For such an input of length n, given an initial list of length m, the total cost achieved by A is at least $m \cdot n$.

We show that there are two (deterministic) algorithms A_1 and A_2 such that if we run both of them on some input of length n, given an initial list of length m, then the sum of their total cost is "around" $m \cdot n$ as well – meaning that on any possible input, at least one of them has a cost "around" $\frac{m \cdot n}{2}$, giving an upper bound on the optimal cost, thus a lower bound 2 for the competitive ratio of A.

The two algorithms are not that mystic: A_1 does not change the order of the elements in the list, only pays p for each request p. The other one, A_2 first reverses the list, say for a cost of $f(m)$ (in our basic model $f(m) = O(m^2)$, we only have to apply a bubble sort), after which it does not change the order of the elements, and thus pays $m - p$ for each incoming request p.

Thus, after an initialization cost of $f(m)$, the two algorithms pay m in total per request, hence after n requests the sum of their total cost is $f(m) + n \cdot m$, hence one of them pays at most $\frac{f(m)+n\cdot m}{2}$. Setting the number of requests n to be $n = f(m)$ (say), we get a lower bound of $\frac{2\cdot f(m)\cdot m}{f(m)\cdot(m+1)} = \frac{2m}{m+1} = 2 - \frac{2}{m+1}$ which converges to 2 if m tends to infinity, proving the claimed lower bound.

Now let us consider the following algorithm Move To Front, or MTF:

- Whenever we get a request, let us move the accessed element to the front.

Note that if we run MTF on $(3, 2, 3, 2)$ we spend a cost of 10, but in general, MTF performs surprisingly well:

Proposition

MTF is 2-competitive.

Proof

We use the so-called potential method for proving this one. To this end, let us start from the initial configuration $(1, 2, \ldots, m)$, and let A be some algorithm (online or not) that also processes the requests. Let $MTF(t)$ and $A(t)$ be the total cost of MTF and A, respectively, having processed the first t requests. That is, $MTF(0) = A(0) = 0$.

We define a "potential function" $\Phi(t)$ as follows: let $\Phi(t)$ be the number of pairs of elements in the list that are in different order in the two lists of MTF and A, having processed the first t requests. For example, if after processing the first three requests the list of MTF is $(3, 1, 4, 2, 5)$ while the list of A is $(1, 4, 5, 3, 2)$, then the value of $\Phi(3)$ is 3: the pairs that are in different order are $(1, 3)$, $(2, 5)$ and $(3, 4)$.

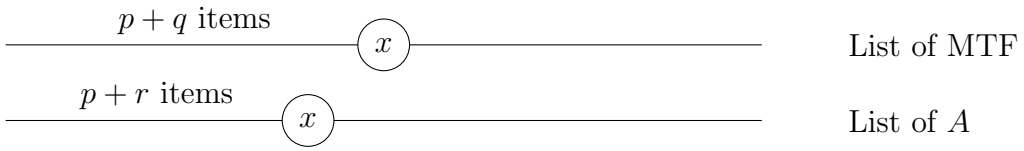
We will show by induction on t that

$$MTF(t) \; + \; \Phi(t) \; \leq \; 2 \cdot A(t)$$

holds for each t, proving the claim, since A is arbitrary and Φ is nonnegative, thus $MTF(I) \leq \mathrm{Opt}(I)$ holds for each input I then.

For $t = 0$ the claim holds, since $\mathrm{MTF}(0) = \Phi(0) = A(0) = 0$.

Now assume the $(t + 1)$th request comes for an element x. Let p be the number of those items preceding x in both lists; q be the number of those items preceding x in the list of MTF, but not in the list of A; and r be the number of those items preceding x in the list of A only. (See the figure below.)



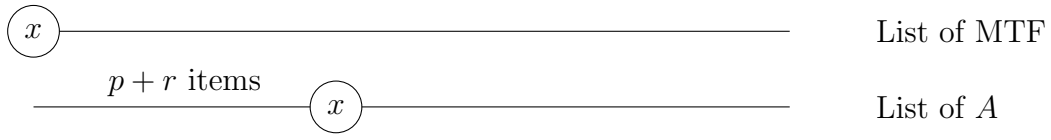
Now let MTF serve x first. Then,

- the cost of MTF increases by $p + q + 1$,

- and at the same time, the potential Φ increases by $p - q$: the q items originally preceding x in the list of MTF only now come after x in that list as well, this decreases the potential by q, while the p items originally preceding x in both lists will have a different order in the two lists now.

Thus, the sum MTF+Φ increases by $(p + q + 1) + (p - q) = 2p + 1$.

After this, let A serve x as well. At this point, x is the first item of the list of MTF:



Now the access cost of x is $p + r + 1$ for A which is at least $p + 1$. Thus at this point we have

$$\begin{aligned} MTF(t+1) + \Phi(t+1) &= MTF(t) + \Phi(t) + 2p + 1 \ \leq 2 \cdot A(t) + 2p + 1 \\ &\leq 2 \cdot (A(t) + p + 1) \ \leq 2 \cdot A(t+1), \end{aligned}$$

so the invariant holds at this point. Now A can do the following:

- Moving x to the front for free. This way $A(t+1)$ does not change, while the potential decreases by $p + r$, and so the invariant still holds.
- Swapping two adjacent elements of the list costs A one (thus increases the right-hand side by 2), while changes (either increases or decreases) the potential, that is, the left-hand side by one (as only the relative order of the particular pair involved changes), and so the invariant still holds.

Thus, the invariant MTF+$\Phi \leq 2 \cdot A$ holds for any A, including the optimal solution as well, showing 2-competitiveness of MTF.

Adding randomization to the mix, one can come up by the following algorithm BIT:

- Initially, we assign uniformly at random a bit to each element of the list.
- When a request arrives to an element p, we flip the bit associated to p.
- If we flipped the bit so that it became 1, we move the element to the front; otherwise we don't modify the list.

Without proof we state that this randomized algorithm performs better than any deterministic one:

Proposition

BIT is 1.75-competitive.

As a note, the best known randomized algorithm is the so-called "COMB" (short for "combined") algorithm, which runs BIT with a probability of 80% and another simple 2-competitive deterministic algorithm called TIMESTAMP with a probability of 20%. The competitive ratio of COMB is 1.6.

The Bin Packing problem

In the Bin Packing problem, the input is a sequence $a_1, a_2, \ldots, a_n$ of real numbers $0 < a_i < 1$, interpreted as sizes of items arriving. We want to store these items into unit-size bins, trying to minimize the number of bins used. Formally, for the request a_i we have to output a bin index $\tau_i \in \{1, 2, \ldots\}$ such that for each bin index $b \geq 1$, the sum $\sum_{\tau_i = b} a_i$ is at most 1. We want to minimize the number $\max_i \tau_i$ of used bins.

For this problem, we are interested in the asymptotic competitive ratio

$$\limsup_{n \to \infty} \left\{ \frac{c_A(I)}{\mathrm{Opt}(I)} : \mathrm{Opt}(I) \geq n \right\}$$

of an algorithm A, that is, we only care about the ratio if the optimal solution's cost tends towards infinity.

We start by proving a lower bound:

Proposition

There is no deterministic online algorithm for Bin Packing with an asymptotic competitive ratio better than 4/3.

Proof

As usual, for a deterministic online algorithm A we can give a bad enough input I and show that the competitive ratio of A is at least 4/3 on at least one prefix I' of I. Since we now want a lower bound for the asymptotic competitive ratio, we also have to ensure that $\mathrm{Opt}(I') \geq n$ for a parameter n.

So let us fix n and consider the following sequence I, consisting of $2n$ items of size 0.4, then an additonal $2n$ items of size 0.6.

After running A on the complete sequence, for each used bin there are four possibilities:

- A bin can contain only one item of size 0.4. Let k_1 be the number of these bins. Observe that these bins were already used after processing the first $2n$ items.
- A bin can contain two items of size 0.4. Let k_2 be the number of these bins. These bins were already used after processing the first $2n$ items.
- A bin can contain an item of size 0.4 and an item of size 0.6. Let k_3 be the number of these bins. These bins were already used after processing the first $2n$ items.
- A bin can contain only one item of size 0.6. Let k_4 be the number of these bins. These bins were not used yet after processing the first $2n$ items.

So A uses in total $k_1 + k_2 + k_3 + k_4$ bins. The optimal cost is $2n$ for the whole sequence (the optimum pairs one item of size 0.4 and one item of size 0.6). We also know that $k_1 + 2k_2 + k_3 = 2n$ (that's the total count of items of size 0.4 inside the bins) and $k_3 + k_4 = 2n$ (that's the total count if items of size 0.6 at the end).

Also, considering only the first $2n$ items, A already used $k_1 + k_2 + k_3$ bins after processing the $2n$ items of size 0.4. For this prefix, the optimal cost is n (the optimum puts two items of size 0.4 into each bin).

Hence, the competitive ratio of A on inputs with an optimal cost of at least n is at least

$$\max\left\{\frac{k_1+k_2+k_3+k_4}{2n},\ \frac{k_1+k_2+k_3}{n}\right\}$$

for some numbers k_1, k_2, k_3 and k_4 such that $k_1+2k_2+k_3=2n$ and $k_3+k_4=2n$.

Converting this to an LP problem we get the following set of constraints:

$$\begin{aligned} k_1+2k_2+k_3 &= 2n \\ k_3+k_4 &= 2n \\ k_1+k_2+k_3+k_4 &\le t\cdot 2n \\ k_1+k_2+k_3 &\le t\cdot n \\ \min\ t & \end{aligned}$$

Here of course the variables k_i are nonnegative integers. It turns out that instead of solving this MILP (Mixed Integer Linear Programming, as there are both integral and real-valued variables) problem for each n and computing the limit of t as n tends to infinity, one can simply switch to the real-valued relaxation of the problem where the variables $x_i \ge 0$ are denoting the values $\frac{k_i}{n}$ and after dividing each constraint by n we get:

$$\begin{aligned} x_1+2x_2+x_3 &= 2 \\ x_3+x_4 &= 2 \\ x_1+x_2+x_3+x_4 &\le 2\cdot t \\ x_1+x_2+x_3 &\le t \\ \min\ t & \end{aligned}$$

Feeding these values to our favourite LP solver gives us the optimal solution $t=\frac{4}{3}$ (where $x_1=0$, $x_2=2/3$, $x_3=2/3$ and $x_4=4/3$). This corresponds in general to the solution where till the end we put two items of size 0.4 to $2n/3$ bins, a $0.4+0.6$ pair to $2n/3$ bins and a single item of size 0.6 to $4n/3$ bins – this can be achieved by, say, packing two items of size 0.4 into a single bin, then make a singleton bin, and iterate this till we run out of the items of size 0.4 – this way we use $4n/3$ bins instead of the optimal n up till this point, then as the items of size 0.6 arrive, we fill the bins containing only one item fitst, then open a new bin for each of the remaining items. This online algorithm indeed produces an asymptotic competitive ratio of 4/3 on this particular set of inputs.

The above approach can be generalized. Let us fix a sequence $s=(a_1,\alpha_1),\ (a_2,\alpha_2),\dots,(a_m,\alpha_m)$ with $0<a_i<1$ being item sizes and $0<\alpha_i$ being coefficients. For each n, such a sequence represents the input sequence of $\alpha_1\cdot n$ items of size a_1, followed by $\alpha_2\cdot n$ items of size a_2, etc, finally $\alpha_m\cdot n$ items of size a_m. In the proof above, we worked with the sequence $(0.4,2),\ (0.6,2)$. For each such fixed sequence s there are finitely many possibilities of putting items into unit-size bins: each option can be represented by an integer vector $(p_1,\dots,p_m)$ with $0\le p_i$ being an integer for each i: this vector represents a bin which, after processing the whole input, contains p_1 number of items of size a_1, p_2 items of size a_2 etc. (We distinguish between a_i and a_j here even if their size is the same.) So let us define the set of packing patterns as the set $P=\{(p_1,\dots,p_m)\in\mathbb{N}_0^m:\ \sum p_ia_i\ge 1\}$. Although $(0,0,\dots,0)$ can be seen as a packing pattern, we do not consider it to be one, so we exclude it from P.

In our proof above we had four packing patterns in P: $(1,0)$, $(2,0)$, $(1,1)$ and $(0,1)$. To each pattern $p\in P$ we introduce a real-valued variable $x_p\ge 0$ as before, the intended semantics of

x_p being that we use $x_p \cdot n$ bins of type p after processing the whole sequence.

With these patterns one can easily formalize that the total number of items of type i is $\alpha_i \cdot n$: for each $1 \leq i \leq m$ we introduce the constraint

$$\sum_{(p_1,\ldots,p_m)\in P} p_i x_i \;=\; \alpha_i.$$

Also, for each prefix $(a_1, \alpha_1),\ (a_2, \alpha_2), \ldots, (a_j, \alpha_j)$ we compute the optimal cost Opt_j (or an upper bound for that). Let β_j stand for the value $\frac{\mathrm{Opt}_j}{n}$ (note that if we cannot give a linear bound for Opt_j, then we don't end up with an LP problem). Now after processing this prefix of the input, exactly those bins are used by the algorithm that have a nonzero entry $p_k > 0$ for some $k \leq j$, so we can write the following constraint for each j:

$$\sum_{(p_1,\ldots,p_m)\in P, p_1+\ldots+p_j>0} x_p \;\leq\; \beta_j \cdot t$$

and by minimizing t we end up by a lower bound for the asymptotic competitive ratio of the Bin Packing problem.

By choosing a complicated enough sequence s, a lower bound of 1.5401 is known for the possible asymptotic competitive ratio for Bin Packing with this method.

Probably the simplest, and still competitive algorithm is the following one called Next Fit (NF):

- We manage a single open bin. When the next item arrives, we put it into the open bin if it fits; otherwise, we close the open bin, open a new one and put the item in that.

As an example, if the input sequence is $0.9, 0.2, 0.9, 0.2, 0.9, 0.2, 0.9, 0.2, 0.9, 0.2$, then Next Fit opens a new bin for each incoming item. An optimal packing would pack the five items of size 0.2 into a single bin and the five items of size 0.9 into separate bins, thus the optimal solution uses six bins, making the competitive ratio $\frac{10}{6}$ for this input.

To give a lower bound for the asymptotic competitive ratio of Next Fit, let us fix the integer n and consider the sequence $1 - \frac{1}{2n}$, $\frac{1}{n}$, $1 - \frac{1}{2n}$, $\frac{1}{n}$,... of $2n$ items in total. Again, Next Fit uses a separate bin for each of these items, making its cost $2n$, while the optimal solution would put all the small items into a single bin, so the optimal cost is $n + 1$. Hence, the optimal cost of this sequence tends to infinity as n does so; the ratio $\frac{2n}{n+1}$ converges to 2, so the asymptotic competitive ratio of Next Fit is at least 2.

However, it is at most 2 as well: for any pair of consecutive bins we have that the total size of the items in the two bins is at least 1 (otherwise we would not have opened a new bin). Hence, if Next Fit uses n bins, then the total size of items in these bins is at least $\lfloor \frac{n}{2} \rfloor$, which is clearly a lower bound for the optimum – or, to put it into another form, if the optimum is n, then Next Fit uses at most $2n + 1$ bins. As $\frac{2n+1}{n}$ also converges to 2, we get:

Proposition

The asymptotic competitive ratio of Next Fit is 2.

In some cases it is reasonable to put an upper bound for the number of open bins (like when we have an actual storage room with limited capacity). Given an integer $k \geq 1$, the HARMONIC(k) algorithm maintains k open bins, with each bin storing items whose size falls within a specific interval. Within each size category, the algorithm manages a Next Fit:

- When the ith item (of size a_i) arrives, let us compute its category:
  - if $\frac{1}{2} < a_i \leq 1$, then the category of the item is 1,
  - if $\frac{1}{3} < a_i \leq \frac{1}{2}$, then the category of the item is 2,
  - if $\frac{1}{4} < a_i \leq \frac{1}{3}$, then it is 3, etc,
  - finally, if $a_i \leq \frac{1}{k}$, then its category is k.
- Now if the item fits in the open bin of its category, we pack it there; otherwise, we close the bin of that category, open a new one and pack the item there.

For example, if the input is 0.4, 0.2, 0.3, 0.6, 0.4, 0.4, 0.2, 0.3 and $k = 4$, then we proceed as follows: as $\frac{1}{3} < 0.4 \leq \frac{1}{2}$, the first item goes into the Category 2 bin, the item 0.2 being smaller than 1/4 goes into the Category 4 bin, then 0.3 goes to the category 3 bin, then the 0.6 goes into the category 1 bin (at this point we are already using four bins for the four items but that's not a problem for the asymptotic competitive ratio), then the 0.4 still fits into the Category 2 bin (which now holds items of total size 0.8), the next 0.4 does not fit into its Category 2 bin, so we close that bin, open a new one for Category 2 and put this item there. Then, 0.2 still fits into its Category 4 bin, and 0.3 fits into its Category 3 bin. (See the Figure below for the result.)

Sure, for this particular example only 3 bins are enough (we can pack the open bins of Category 1 and 2 into a single bin, and also the rest can go into a single bin as well) but let us analyze the asymptotic competitive ratio of the HARMONIC(k) algorithm.

Proposition

The asymptotic competitive ratio of HARMONIC(k) is at most 1.69103 for a large enough k.

Proof

We apply here the so-called weight function method. In general, the weight function method works as follows: for each item size x, we define a weight $w(x) \geq x$ such that

- the total weight which fits into a single bin is upperbounded by some constant U and
- HARMONIC(k) puts a total weight of at least L into each closed bin.

Clearly, as for each fixed k, HARMONIC(k) leaves at most k bins open. So if the total weight of the items is W, then the optimum is at least W/U by the first property, and the cost of HARMONIC(k) is at most $W/L + k$ by the second one, making the asymptotic competitive ratio at most U/L.

For HARMONIC(k), our particular weight function is

$$w(x) = \begin{cases} 1 & \text{if } \frac{1}{2} < x \le 1 \\ \frac{1}{2} & \text{if } \frac{1}{3} < x \le \frac{1}{2} \\ \frac{1}{3} & \text{if } \frac{1}{4} < x \le \frac{1}{3} \\ \frac{1}{4} & \text{if } \frac{1}{5} < x \le \frac{1}{4} \\ \dots & \\ \frac{k}{k-1}x & \text{if } x \le \frac{1}{k}. \end{cases}$$

Let us first show that HARMONIC(k) puts a total weight of at least $L = 1$ into each closed bin. Clearly, if the category of a bin is some $j < k$, then exactly j items go into the bin before it gets closed: as items in those category have a weight of $1/j$, the total weight is indeed 1 in these bins.

Now for a closed bin of category k, the total size of the items is at least $\frac{k-1}{k}$, since we close such a bin when some item of size at most $\frac{1}{k}$ does not fit. As these items of size x have a weight $\frac{k}{k-1}x$, their total weight is at least $\sum \frac{k}{k-1}x = \frac{k}{k-1}\sum x \ge \frac{k}{k-1} \cdot \frac{k-1}{k} \ge 1$.

Now let us prove some upper bound on the possible sum $\sum w(x)$ if $\sum x < a$ for some constant a. For this, we define the density $\varrho(x) = \frac{w(x)}{x}$ of an item of size x. Clearly, since $w(x) \ge x$, the density is always at least one. Also, if an item is of size $x \le \frac{1}{j}$ for some $j < k$, then its density is less than $\frac{j+1}{j}$.

Clearly, if we use an item of size $x > 1/2$, then it's better to use an item of size $x = \frac{1}{2} + \epsilon$ for some small enough $\epsilon > 0$. In this case, the total weight in the bin becomes at least $1 + \frac{k}{k-1} \cdot \frac{1}{2}$ (if we fill the rest with small items). Otherwise, the maximum density available by any item is smaller than $3/2$. So the best option is to use one item of size $x = \frac{1}{2} + \epsilon$. For the remaining space, if we use an item of size $x = \frac{1}{3} + \epsilon$, then the total weight in the rest of the bin becomes at least $\frac{1}{2} + \frac{k}{k-1}\frac{1}{6}$. Otherwise, the maximum available density becomes $\frac{4}{3}$, thus the total weight in the rest of the bin is at most $\frac{2}{3}$. Thus the best option is to use one item of size $x = \frac{1}{3} + \epsilon$.

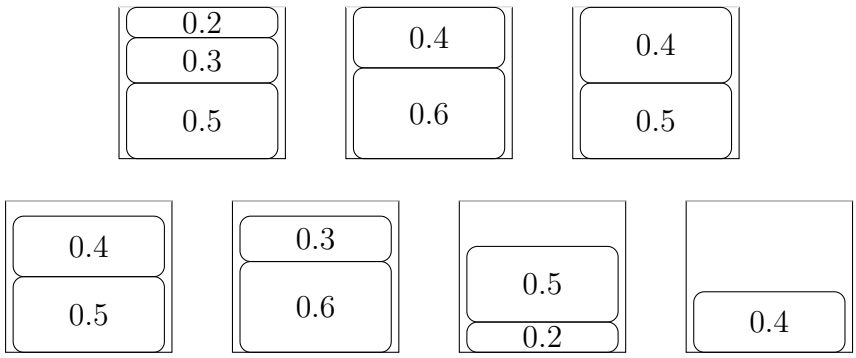
Now the remaining space in the bin is $\frac{1}{6} - 2\epsilon$. If we use an item of size $x = \frac{1}{7} + \epsilon$, then the weight we can pack in is at least $\frac{1}{6} + \frac{k}{k-1}\frac{1}{42} > \frac{8}{42}$, while if we don't do that, then the maximum possible density is $\frac{8}{7}$, yielding a possible weight of at most $\frac{8}{42}$.

In the next step the best option becomes to use an item of size $x = \frac{1}{43} + \epsilon$ (provided $k \ge 43$ of course, otherwise just simply use a small item) and so on: the total weight of items in the bin will be at most $1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{42} + \ldots \approx 1.69103$.

If there is no upper bound on the number of the open bins, then one can come up with the following two algorithms First Fit and Best Fit:

- Both algorithms manage open bins, never close a bin.
- If the next item arrives, then they only a new bin only if the new item does not fit into any of the already used bins.
- Otherwise, First Fit puts the item into the first possible bin; Best Fit picks the bin among the possible ones which is most fully loaded.

As an example, running the algorithms First Fit and Best Fit on the input 0.5, 0.6, 0.3, 0.4, 0.2, 0.5, 0.4 we end up with the following configurations respectively:



So on this particular input, First Fit produces an optimal packing while Best Fit does not, but of course it can happen the other way around.

Both of these algorithms are asymptotically better than Next Fit:

Proposition

The asymptotic competitive ratio of both First Fit and Best Fit is 1.7.

Proof

Again, we can use the weight function method here. Our current weight function is:

$$w(x) = \frac{6}{5}x \; + \; \begin{cases} 0.4 & \text{if } \frac{1}{2} < x \\ 0.1 & \text{if } \frac{1}{3} < x \leq \frac{1}{2} \\ \frac{3}{5}x - 0.1 & \text{if } \frac{1}{6} < x \leq \frac{1}{3} \\ 0 & \text{if } x \leq \frac{1}{6} \end{cases}$$

First we prove some upper bound of total weight that can be pushed into a single bin. Clearly, as the weight is nonnegative, we can assume that $\sum x = 1$. Then, the sum of the $\frac{6}{5}x$ parts of the weight function becomes $\frac{6}{5} = 1.2$ and so we have to show that the sum of the second parts, also called the "penalty", cannot exceed 0.5. Clearly, we only have to consider those items of size greater than $\frac{1}{6}$ (since the smaller ones have a penalty of zero), and the number of these combinations is finite:

- We can use one item of size $x > \frac{1}{2}$. Then the penalty is 0.4.
- We can use one item of size $x > \frac{1}{2}$ and one item of size $\frac{1}{3} \leq y \leq \frac{1}{2}$. Then the total penalty is 0.5.
- We can use one item of size $x > \frac{1}{2}$ and one item of size $\frac{1}{6} < y \leq \frac{1}{3}$. Then the total penalty is $0.4 + \frac{3}{5}y - 0.1 = 0.3 + \frac{3}{5}y \leq 0.3 + \frac{1}{5} = 0.5$.
- We can use one item of size $x > \frac{1}{2}$ and two items of sizes $\frac{1}{6} < y, z \leq \frac{1}{3}$. Then the total penalty is $0.4 + \frac{3}{5}(y + z) - 0.2 = 0.2 + \frac{3}{5}(y + z) \leq 0.2 + 0.3 = 0.5$ since $y + z \leq 0.5$.
- If we do not use an item of size $x > 0.5$, then the penalty of each item is at most 0.1. Since each item considered is larger than $\frac{1}{6}$, we can use at most five items, thus the total penalty is at most 0.5.

Thus, 1.7 is an upper bound for the possible total weight that can be fit into a single bin.

By a longer case analysis one can prove that if we run either FF or BF, then except for a constant number of bins, each bin gets packed by a total weight of at least 1. Some interesting cases:

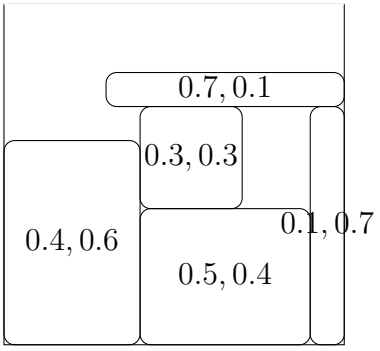
- Those bins containing at least one item of size $x > \frac{1}{2}$ have a weight of at least $\frac{6}{5} \cdot \frac{1}{2} + 0.4 = 1$ by this single item, so these bins are fine.
- If there are at least two items of size at least $\frac{1}{3}$, then the total weight in the bin is at least $\frac{6}{5} \cdot \frac{2}{3} + 0.2 = 0.8 + 0.2 = 1$, so these bins are fine.
- Those bins containing items of a total size of at least $\frac{5}{6}$ have a total weight of at least 1, so these bins are fine.
- So we only have to consider those bins of total size at most $\frac{5}{6}$. In the case of First Fit, the first such bin might contain a weight less than one. However, the other such bins can only contain items of size more than $\frac{1}{6}$ (otherwise we would put them into the first such bin).
- There cannot be more than two bins of total size at most $\frac{1}{2}$ according either to First Fit or Next Fit. So that's an option for a second such bin of weight less than one, but the other bins contain a weight more than $\frac{1}{2}$.
- If there is a bin containing a total size less than $\frac{2}{3}$, then all the later bins contain items of size more than $\frac{1}{3}$, thus their total size it more than $\frac{2}{3}$. Hence there is at most one such bin.
- So we have to consider only bins of total size at least $\frac{2}{3}$ but at most $\frac{5}{6}$, which contain only items with size between $\frac{1}{6}$ and $\frac{1}{2}$, and with at most one item larger than $\frac{1}{3}$.

Continuing in this fashion we get the claimed result.

Bin packing variants: vector, box, and strip packing

The strip packing is a 2-dimensional generalization of the bin packing. Here, the input is a sequence of pairs (w_i, h_i) with $0 \leq w_i, h_i \leq 1$, representing rectangles of a given width and height. We have to pack these rectangles without overlaps onto a strip of width 1 of infinite height; the objective is to minimize the maximal used height.

As an example, if the input is the sequence $(0.4, 0.6)$, $(0.3, 0.3)$, $(0.5, 0.4)$, $(0.1, 0.7)$, $(0.7, 0.1)$, then a possible solution of cost 0.8 is depicted on the Figure below.

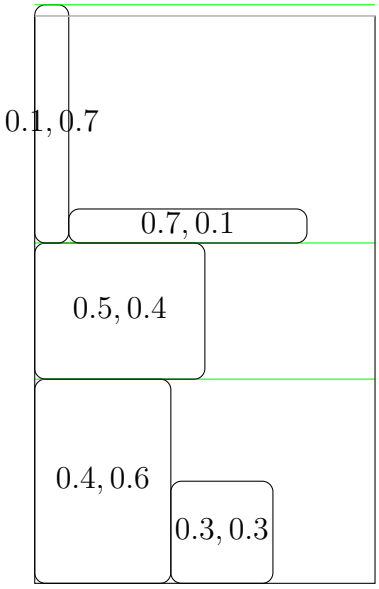


A possible way to define shelves inside the strip: a shelf has some height, fixed upon its construction. Initially, there are no shelves. When a rectangle arrives, a shelf-based algorithm has to decide whether to put the new rectangle onto a shelf (whose height is at least as large as the height of the box and which has a large enough remaining width), or to open a new shelf for the rectangle. In the latter case, the algorithm should determine the height of the new shelf as well.

For example, one possible algorithm is the following:

- If the new item does not fit on any of the existing shelves, let us open a new shelf, having height exactly the same as the height of the new box.
- Otherwise, put the item onto the first shelf on which it fits.

Running this algorithm (that can be seen as a generalization of First Fit) on the previous example we get the solution depicted on the Figure below. Shelves are denoted by green horizontal lines.

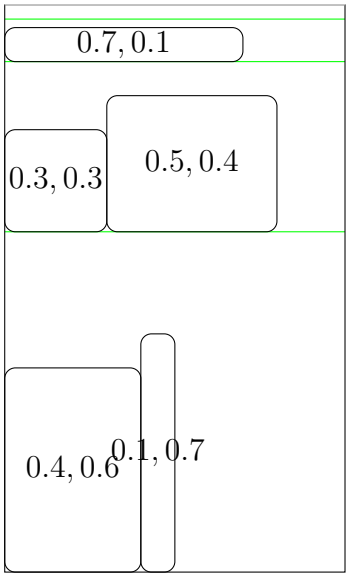


Clearly, this algorithm can leave large empty areas: in fact, it is not competitive as shown by the input $(0.5+\frac{1}{2n},\frac{1}{2n})$, $(0.5+\frac{2}{2n},\frac{1}{2n})$, $(0.5+\frac{3}{2n},\frac{1}{2n})$,...,$(0.5+\frac{n}{2n},\frac{1}{2n})$. On this input, each box would get a fresh shelf, the total cost being larger than $n/2$, while in an optimal solution all the boxes would fit in a single row, making the optimal cost 1.

A better class of algorithms is called Next Fit Shelf, or NFS, which is parametrized by a constant $0 < r < 1$. The algorithm creates shelves of height exactly r^i with i being an integer; note that $1 = r^0 > r > r^2 > r^3 > \ldots$. For each such height, there will be at most one open shelf of that given height. When an item of size (w, h) arrives, then the algorithm determines the smallest possible shelf height on which it fits, that is, the integer i with $r^{i+1} < h \leq r^i$. If there is no open shelf of this height r^i, or if the new box does not fit on the open shelf of this height, the algorithm opens a new shelf of height r^i (closing the previous open shelf if such a shelf exists), otherwise the item is placed on the open shelf.

As an example, for $r = 0.5$, the possible shelf heights are 1, 0.5, 0.25, 0.125 and so on. For the same input sequence $(0.4, 0.6)$, $(0.3, 0.3)$, $(0.5, 0.4)$, $(0.1, 0.7)$, $(0.7, 0.1)$ the run of $\mathrm{NFS}_{0.5}$ is the following (depicted on the Figure below). The first item has height 0.6 which needs a height of 1, so we open a new shelf of height 1 for this item. Then, the next item of height 0.3 fits on a shelf of height 0.5, so we open a new shelf of this height. The item of height 0.4

also fits on a shelf of height 0.5 so we place it next to the item $(0.3, 0.3)$ since it has enough remaining width to do that. Then, the item $(0.1, 0.7)$ is put on the shelf of height 1, and the item $(0.7, 0.1)$ gets a new shelf of height 0.125.



On this particular example, the algorithm has a cost of 1.625. Should a new item $(0.3, 0.4)$ arrive, it would be put onto a shelf of height 0.5. Since there is not enough width on the open shelf of that height, the algorithm would close that open shelf and open a new one of height 0.5 for the new item.

Though the advantages of this algorithm might not be apparent inspecting this input, it is still a constant competitive one:

Proposition

The algorithm NFS_r is $\left(\frac{2}{r} + \frac{1}{r(1-r)}\right)$-competitive and is asymptotically $\frac{2}{r}$-competitive.

Proof

Key facts: the total height of all the open shelves cannot be too large as the sum of the geometric progression is upperbounded; closed shelves are treated similarly to Next Fit: their height is used by at least a fraction of r (otherwise we would have used a smaller shelf for the item in question), and two consecutive shelves of the same height have to be filled up to a total width of at least one.

The k-server problem

In the k-server problem, we have a metric space M, that is: a (finite or infinite) set M of points, equipped by a distance d satisfying the following properties: $d(x, y) \geq 0$; $d(x, x) = 0$ and for each $x \neq y$, $d(x, y) > 0$; and the triangle inequality $d(x, y) + d(y, z) \geq d(x, z)$. A server configuration is a k-element multiset $C \subseteq M$. Initially, the servers are in some initial configuration C_0. A request is a point p of M. If we are in a configuration C and we get the request $p \in M$, then a response is a new configuration C' with $p \in C'$. The cost of this response is $d(C, C')$, the minimum total cost of moving the servers from the configuration C to C'.

For example, assume $k = 2$, our metric space is the line with $d(x, y) = |x - y|$, and we are in the configuration $\{1, 4\}$, at which point we get the request 3. Then it's a possibility to move our servers into the configuration $\{1, 3\}$ for a cost of 1. Another option would be to move into the configuration $\{3, 4\}$ for a cost of 2. A third one is to move into the configuration $\{3, 10\}$ for a cost of $2 + 6 = 8$, though this one seems to be not too well justified.

We call an algorithm (either online or not) lazy if it satisfies the following property: if we get a request p in the configuration C, then

- if $p \in C$, then the new configuration becomes C,
- otherwise, the new configuration has the form $C - \{q\} \cup \{p\}$ for some $q \in C$.

That is, if it's not needed to move a server, we don't move them; otherwise, we choose one server to serve the request, and we move only that one.

Proposition

Any algorithm, can be effectively transformed into a lazy one which cannot have a worse cost on any input than the original algorithm. If the original algorithm is online, then so is the new one.

For the k-server problems, we usually work with the weak competitive ratio: A is weakly c-competitive if for any initial configuration C_0 there exists some constant β such that $c_A(I) \leq c \cdot \mathrm{Opt}(I) + \beta$.

The greedy algorithm is the lazy one which moves the server q that is closest to the request. In most of the metric spaces, this is not a competitive algorithm. To see this, let our metric space be the line, $k = 2$ and the initial configuration be $\{0, 4\}$. Upon the request sequence $1, 0, 1, 0, \ldots, 1, 0$ the greedy algorithm moves the first server back and forth between the two points, and the cost diverges to infinity, while the optimal cost for a long enough sequence would be 3, simply moving the server from 4 to 1 in the first step.

We've already seen a(n almost) special case of the k-server problem before. A metric space is called uniform if the distance between different points is always 1. Then, the paging problem becomes a k-server problem on a uniform metric space with the single difference of the initial filling of the cache, which increases the cost by k. We've also seen that for the paging problem there is no deterministic online algorithm having a competitive ratio better than k. This holds in any metric space:

Proposition

If M has at least $k + 1$ points, then there is no deterministic online algorithm which is better than weakly k-competitive.

Proof

The proof is similar to the one given for the List Access problem's lower bound. Key points: having $k + 1$ points we can always request the point which is not covered by the servers. Also, we can define k algorithms with a grand total cost being equal to the cost of this run, meaning that the optimum is at most the cost of the algorithm, divided by k.

So let A be a deterministic online algorithm and let the initial server configuration be $C_0 = \{p_1, p_2, \ldots, p_k\}$ and p_{k+1} be a $(k+1)$th point. For each t, let $q_t = \{p_1, \ldots, p_{k+1}\} - C_{t-1}$. The request sequence is now $(q_1, q_2, \ldots, q_n)$. Clearly, the first query q_1 is served by A from

q_2 since that's the empty point after $t = 1$. Similarly, the second query is served from q_3 and so on, so the total cost of A on this input is $\sum_{i=1}^{n-1} d(q_i, q_{i+1})$.

Now we define the lazy online algorithms $A_1, \ldots, A_k$ as follows. In the first step, A_i serves $q_1 = p_{k+1}$ from p_i. For the remaining queries, if q_t is not covered, then it gets served by q_{t-1} (so the algorithms differ only in their first step).

Let C_t^i be the configuration of A_i after serving the tth request. Then formally, $C_0^i = \{p_1, \ldots, p_k\}$ for each i, $C_1^i = \{p_1, \ldots, p_{k+1}\} - \{p_i\}$ and

$$C_{t+1}^i = \begin{cases} C_t^i & \text{if } q_{t+1} \in C_t^i \\ C_t^i - \{q_t\} \cup \{q_{t+1}\} & \text{otherwise} \end{cases}$$

As the algorithms are lazy, we get that each C_t^i contains exactly k pairwise different points. Clearly, $q_t \in C_t^i$ for each $t \geq 1$ and i. We also claim that $C_t^i \neq C_t^j$ if $i \neq j$ and $t \geq 1$. This holds for $t = 1$ by the definition of the first step. Now assume this holds for t and let us move to $t + 1$. So let $C_t^i \neq C_t^j$.

- If $q_{t+1} \in C_t^i \cap C_t^j$, then $C_{t+1}^i = C_t^i \neq C_t^j = C_{t+1}^j$.
- If $q_{t+1} \in C_t^i$ and $q_{t+1} \notin C_t^j$, then $C_{t+1}^i = C_t \ni q_t$ and $C_{t+1}^j = C_j^j - \{q_t\} \cup \{q_{t+1}\}$, and this latter set does not contain q_t, so these two sets are different.
- If q_{t+1} is not a member of either C_t^i or C_t^j, then $C_t^i - \{q_t\} \neq C_t^j - \{q_t\}$ and also $C_{t+1}^i \neq C_{t+1}^j$, so these two sets are different as well.

Thus all the possible k-element subsets of $\{p_1, \ldots, p_{k+1}\}$ containing q_t occur as configurations of the form C_t^i. Amongst these sets there is only one which does not contain q_{t+1}, this one pays $d(q_t, q_{t+1})$ when serving q_{t+1}, for the others the request is covered and thus free to serve. Hence the total cost payed by the k algorithms is $d(q_t, q_{t+1})$ for each step after the very first one, which has a total cost of $\sum_{i=1}^{k} d(p_i, p_{k+1})$. Thus, the total cost of these k algorithms is $\sum_{i=1}^{k} d(p_i, p_{k+1}) + \sum_{i=1}^{n-1} d(q_i, q_{i+1})$. Hence, the best one amongst them pays at most $\frac{1}{k}$ times this cost and so

$$\text{Opt} \leq \frac{\sum_{i=1}^{k} d(p_i, p_{k+1}) + \sum_{i=1}^{n-1} d(q_i, q_{i+1})}{k}$$

thus the limit of the cost of the algorithm and the optimal cost tends to at least k as the length of the request sequence tends to infinity, meaning A cannot perform better than being weakly k-competitive.

On the line, the following algorithm Double Coverage, or DC, performs surprisingly well:

- Suppose our current configuration is C and the next request is p.
- If $p \in C$, then the new configuration is C as well.
- If $p < \min C$ or $p > \max C$, then we move the server which is closest to the request.

- Otherwise, there are two servers $x, y \in C$ between which p falls, say $x < p < y$. Let d be the minimum of $d(p, x)$ and $d(p, y)$. We move both x and y towards p, their new positions being $x + d$ and $y - d$, respectively.

Observe that DC is not a lazy algorithm.

As an example, let $k = 3$ and the initial server configuration be $\{1, 4, 10\}$. Assume the input is the request sequence $6, 9, 0, 3, 7$. Then the server configurations in order are $\{1, 6, 8\}$ (moving 4 and 10 two units towards the request 6), $\{1, 6, 9\}$ (8 serves 9), $\{0, 6, 9\}$ (1 serves 0), $\{3, 3, 9\}$ (both 0 and 6 move towards 3, three units) and $\{3, 5, 7\}$ (one of the 3s and the 9 is moved towards 7, two units). The total cost of DC in this case is $4 + 1 + 1 + 6 + 4 = 16$. The cost given by the greedy algorithm is 8.

Proposition

The algorithm DC is weakly k-competitive on the line.

Proof

We again use the potential method. Let $s_1 \leq s_2 \leq \ldots \leq s_k$ be the server positions according to the DC algorithm and let $x_1 \leq x_2 \leq \ldots \leq x_k$ be the server positions according to some lazy algorithm A. Initially, $s_i = x_i$ for each i.

We define the following potential function Φ:

$$\Phi \;=\; k \cdot \sum_{i=1}^{k} |x_i - s_i| \;+\; \sum_{i<j} (s_j - s_i).$$

We claim that

$$DC(t) \;+\; \Phi(t) \;\leq\; k \cdot A(t) \;+\; \Phi(0)$$

holds for each $t \geq 0$ where again , $DC(t)$ and $A(t)$ respectively denote the total cost of DC and A, after processing the first t requests. The claim holds for $t = 0$.

Let the $(t + 1)$th request be for the point q and let it be served by A first. Assuming A moves a server x_i to q for a cost of d, the quantity $A(t)$ increases by d, so the right-hand side increases by $k \cdot d$. Clearly, $DC(t)$ does not change at this point, nor the second term of the potential function and the term $\sum_{i=1}^{k} |x_i - s_i|$ can change by at most d, thus the left-hand side can increase by at most $k \cdot d$ as well. So at this point, the inequality still holds.

Now let DC serve q. We show that the left-hand side cannot increase by a case analysis.

- If q lies strictly between two servers $s_j < q < s_{j+1}$, then let d be the distance $\min\{q - s_j, s_{j+1} - q\}$. In this case the cost of this step is $2d$, the DC term increases by this amount.
- For the second term of the potential function, the distance between s_j and s_{j+1} decreases by $2d$. For all the other servers s_k, the total distance from s_j and s_{j+1} from s_k remains the same: one of the moving servers gets closer to s_k, the other one gets further to s_k by the same amount. So the second term of the potential function decreases by $2d$.

- For the first term of the potential function, one of the servers s_j and s_{j+1} moves a distance of d and gets to the point q on which there is a server present: by this, the sum $\sum_{i=1}^{k} |x_i - s_i|$ decreases by d. The other server can increase this sum by at most d, so the first term of the potential function cannot increase.
- Hence, if q lies strictly between two servers of DC, the inequality still holds. Otherwise, if q falls outside the servers of DC, then DC moves a single server (either s_1 or s_k) for a cost of d. This way the first term of the potential function decreases by $k \cdot d$, the cost of DC increases by d, while the second term of the potential function increases by $(k-1)d$. In total, the left-hand side does not change in that case either, and the inequality still holds.

An algorithm which is conjectured to be weakly k-competitive for any metric space is the so-called Work Function Algorithm. This algorithm requires the computation of the offline optimum for each prefix of the input sequence. Let us denote for each possible server configuration C and integer t by $w_t(C)$ the offline optimal cost of serving the first t requests, ending up in the configuration C.

The computation of $w_t(C)$ can be done by induction on t:

- $w_0(C_0) = 0$ and for each $C \neq C_0$, $w_0(C) = \infty$.
- If the tth request is p, and $p \notin C$, then $w_t(C) = \infty$.
- Otherwise, $w_t(C) = \min_{C'} \left\{ w_{t-1}(C') + d(C', C) \right\}$.

For an example, if we have the points $0, 2$ and 5 in the metric space with $d(x, y) = |x - y|$, the initial server configuration is $\{0, 5\}$ and the request sequence is $2, 0, 2$, then the computation table is the following:

	$\{0,2\}$	$\{0,5\}$	$\{2,5\}$
	∞	0	∞
2	3	∞	2
0	3	4	∞
2	3	∞	6

For example, $w_2(\{0, 5\})$ is computed as follows: in the previous row, $\{0, 2\}$ has a cost of 3, and the cost of moving the servers from $\{0, 2\}$ to $\{0, 5\}$ is 3, producing a total cost of 6. The second option is moving from $\{2, 5\}$ to $\{0, 5\}$ for a cost of 2. As $w_1(\{2, 5\}) = 2$, this option produces a total cost of $2 + 2 = 4$. This being the minimum, $w_2(\{0, 5\})$ becomes 4.

The work function algorithm is a lazy algorithm, which works as follows: if the tth request is p is not in the current configuration C, then we choose the server q to serve p which minimizes the sum $w_t(C - \{q\} \cup \{p\}) + d(q, p)$.

Let us see the work function algorithm acting on this request sequence $2, 0, 2$.

- The initial configuration is $\{0, 5\}$. If we move the server from 0 to serve the request 2, the cost is $w_1(\{2, 5\}) + d(0, 2) = 2 + 2 = 4$. If we move the server from 5 to 2, the cost is $w_1(\{0, 2\}) + d(5, 2) = 3 + 3 = 6$. So we choose 0 to serve the request and we are in the configuration $\{2, 5\}$.

- The next requested point is 0. If we move 2 to 0, the cost is $w_2(\{0,5\})+d(2,0) = 4+2 = 6$. Moving 5 to 0 has the cost $w_2(\{0,2\}) + d(5,0) = 3+5 = 8$ so we move 2 to 0 and we are in the configuration $\{0,5\}$.
- The next requested point is 2. If we move 0 to 2, the cost is $w_3(\{2,5\})+d(0,2) = 6+2 = 8$. If we move 5 to 2, the cost is $w_3(\{0,2\}) + d(5,2) = 3+3 = 6$. So we move 5 to 2 and we are in the configuration $\{0,2\}$.

Proposition

The work function algorithm is $(2k-1)$-competitive for any metric space (and is shown to be k-competitive for lots of cases).