Exercise: Packing and Paging

1. **NextFit algorithm.** Recall that we introduced the FirstFit algorithm for the BinPacking problem during the lecture. Now consider another algorithm **NextFit**:

   `NextFit` Algorithm

   When a new item arrives, put it into the *last* bin if the item fits into that bin. Otherwise, close the last bin, open a new bin, and put this new item to this new bin.

   (a) Claim that in the NextFit solution, the total size of jobs in any two consecutive bins is at least 1.

   If there are two consecutive bins together have items with total size less than 1, the items in the second bins can be assigned to the first bin without exceeding the bin capacity. According to the NextFit algorithm, the jobs in the second bin would be put in the first bin. Hence, the case won't happen.

   (b) Prove that NextFit $\leq 2 \cdot \text{OPT}$.

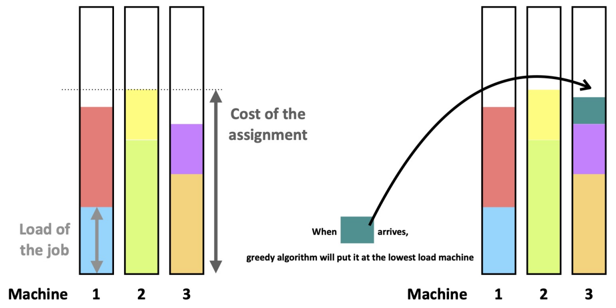
   Assume that NextFit uses k bins. By (a), any two consecutive bins (but the last one) have total item sizes at least 1. Hence, the total size of all items is at least $\lfloor \frac{k-1}{2} \rfloor$. Moreover, the optimal solution uses at least $\lfloor \frac{k-1}{2} \rfloor$ bins. Hence,

$$\begin{aligned} \lfloor \frac{k-1}{2} \rfloor &\leq \text{OPT} \\ \frac{k-1}{2} &\leq \text{OPT} + 1 \\ k - 1 &\leq 2 \cdot (\text{OPT} + 1) \\ \text{NextFit} = k &\leq 2 \cdot \text{OPT} + 3 \end{aligned}$$

2. **Online load balancing.** There is a finite set of m machines. A sequence of n jobs is arriving where each job J_i is specified by its processing load ℓ_i. Each job must be assigned to exactly one of the machines upon arrival. The total load of a machine is the sum of loads of the jobs assigned at it. The cost of an assignment is the maximum total load among the machines. The goal of the LoadBalancing problem is to find an assignment with the minimum cost.

   Consider the greedy algorithm (also see the illustration):

   `Assign each arriving job` r_i `to the machine with the lowest load (breaking ties arbitrarily).`



(a) Consider any job J_k. Claim that the cost of the optimal assignment is at least ℓ_k.

In the optimal solution, there is a machine where job J_k is assigned. Hence, the load (in the optimal solution) of the machine is at least ℓ_k

(b) Claim that the cost of the optimal assignment is at least $\frac{\sum_{j=1}^{n} \ell_j}{m}$
(Hint: Use the similar argument for bin packing optimal cost lower bound.)

To minimize the highest load, the best thing an algorithm can do is to put the load as even as possible. If there is an algorithm which can place the total load evenly on each of the machines (without worrying about if we have to cut a job into pieces), the cost of this algorithm is

$$\frac{\text{total load}}{\text{number of machines}} = \frac{\sum_{j=1}^{n} \ell_j}{m}.$$

The optimal algorithm has to assign the jobs without cutting them, so it cannot do better than this assignment. Hence,

$$\text{the cost of OPT} \geq \frac{\sum_{j=1}^{n} \ell_j}{m} .$$

(c) Show that the greedy algorithm is $2 - \frac{1}{m}$-competitive.

Assume the highest load machine M_i in the algorithm assignment has load $\ell_k + s$, where ℓ_k is the load of the last job J_k assigned to M_i and $s \geq 0$. Since the algorithm uses a greedy strategy, it always assigns the job to the current lowest load machine. Hence, when the job J_k arrives, the load of each of the machines is at least s (otherwise, the job J_k will be assigned to another machine with lower load instead of to the machine M_i). Therefore, the total load of the jobs is at least $(m-1) \cdot s + (\ell_k + s) = m \cdot s + \ell_k$.
By part (b), the cost of optimal is at least $\frac{\text{total load}}{m} \geq \frac{m \cdot s + \ell_k}{m} = s + \frac{\ell_k}{m}$. Hence, $s \leq \text{OPT} - \frac{\ell_k}{m}$ (where OPT denotes the cost of the optimal solution).
The cost of the algorithm

$$\ell_k + s \leq \ell_k + (\text{OPT} - \frac{\ell_k}{m}) \leq \text{OPT} + \ell_k \cdot (1 - \frac{1}{m}) = (2 - \frac{1}{m}) \cdot \text{OPT}$$

(where the second inequality comes from part (a) that $\text{OPT} \geq \ell_k$).

3. Consider the algorithm `LIFO` (Last-In-First-Out) for the Paging problem:

`LIFO` (Last-In-First-Out) Algorithm

When there is a page fault and the cache is full, replace the page that has been copied into the cache the most recently.

Answer the following questions:

(a) Given a sequence of n page requests, show that LIFO is at most $\Omega(\frac{n}{k})$-competitive.

In the worst case, LIFO incurs one page fault for each of the page requests. And the optimal solution has to copy each of the k pages into the cache. Therefore, for any sequence of requests R, $\frac{\texttt{LIFO}(R)}{\text{OPT}(R)} \leq \frac{n}{k}$.

(b) Show that the analysis in (a) is tight.

Consider the sequence of requests $R = 1, 2, 3, \cdots, k$ followed by n pairs of requests $k+1, k$. An offline algorithm can evict page $k-1$ when the page $k+1$ is requested for the first time and the total number of page faults is $k+1$. However, FirstFit replace page k when the page $k+1$ is requested and replace page $k+1$ when page k is requested. Hence, it incurs a page fault for every requests, and the total number of page faults is $k-1+n$. Therefore, when n is large enough, $\frac{\texttt{LIFO}(R)}{\text{OPT}(R)} \geq \frac{k-1+n}{k+1} \approx \Omega(\frac{n}{k})$.

4. Consider the algorithm CLOCK (CLOCK-replacement) for the Paging problem:

LIFO (Last-In-First-Out) Algorithm

For every page in the cache, there is a binary variable *mark* attached to it. Initially, when a page is copied into the cache, its *mark* bit is set to 0. Once a page that in the cache is accessed, its *mark* bit is set to 1

When there is a page fault and the cache is full, replace the first page that has its *mark* bit value equal to 0.

Given the phase partitioning, answer the following questions:

(a) Show that by the phase partitioning, the CLOCK algorithm incurs at most k page faults, where k is the cache size.

Since CLOCK evicts the page that has *mark* bit value 0, it never evicts a page that was requested earlier in the same phase. Hence, CLOCK incurs at most k page faults in a phase as there are at most k distinct pages in each phase.

(b) Show that CLOCK is an optimal online algorithm for the Paging Problem

We prove the optimality of CLOCK by claiming that for any i, OPT incurs at least 1 page fault among the period from the second request in phase i till the first request in phase $i+1$ Consider the cache just after the first request r in phase i. At this moment, there are $k-1$ pages in the cache, not counting r. Since the first request in phase $i+1$ is different from any of the pages in phase i, there are k distinct requests in the interval from the second request in phase i till the first request in phase $i+1$. Hence, any feasible algorithm has to evict one page to accommodate these requests, and so does OPT. Therefore, OPT has at least one page fault in this interval.

By this claim and (a) that CLOCK incurs at most k page faults in each phase, $\frac{\texttt{LRU}(R)}{\text{OPT}(R)} \leq \frac{p \cdot k}{p-1} = O(k)$ for any requests sequence R, where p is the number of phases in R. Since no online algorithm is at least $\Omega(k)$-competitive, CLOCK is an optimal online algorithm.