1 Exercise 1

Suppose we have a set of requests over the real continuous time. Teach request arrives at an arbitrary time. An algorithm may provide a service point at any time. The response time of a request is the time from its arrival until the next service point. The cost of an algorithm (which we want to minimize) is:

$$\Big(\sum_{requests} ResponseTimeOfRequest\Big) + \#ServicePoints$$

We note that a service point services all waiting responses simultaneously.

1.1 A 2-Competitive Deterministic Algorithm

We use the parameter $SumCurrentResTime$ to note, at any point in time, the sum of the response times of all the waiting requests if a service point is set in the current time.
We view the following deterministic online algorithm:
Every time that $SumCurrentResTime = 1$, the algorithm provides a service point.
We show that the above algorithm is 2-Competitive:
We divide the time sequence into segments, according to the service points provided be the algorithm. For each service point, we define the segment of the service point to be from the time of the first request that the service point services, until the times that the service point is provided. We note that the segments are **seperate**.
For every such segment we place one service point, and there for:

$$\#ServicePoints = \#Segments$$

We note that by the definition of the segments, there is no request outside of a segment. Also, since the service point is set in the time when $SumCurrentResTime = 1$, we get that the sum of the response times of the requests in each segment is 1. Hence,

$$\sum_{requests} ResponseTimeOfRequest = \sum_{s\in S}\sum_{r\in s} ResponseTimeOfRequest(r) = \sum_{s\in S} 1 = \#segments$$

When S is the set of all segments, and $r \in s$ are all requests that occur durring the time of the segment. Therefore, the cost of the above algorithm is $cost(ON) = 2\#segments$.
We claim that every algorithm that serves all of the requests (and specifically the optimal algorithm) must have a cost of at least 1 to service the requests of each segment (from the segments described above). If the claim holds, than the cost of OPT is at least $\#segments$, $cost(ON)/cost(OPT) \le 2$, and the above algorithm is 2-competitive.
For each segment as discribed above, eiter OPT has a service point in that time interval (which increases the cost by one), or the distance between the first request in the interval and the first service point after it is greater than 1 (which increases the cost by more than 1). In any case, we get that in order to service the requests in each segment, OPT must pay at least 1. Therefore the cost of OPT is at least 1.

1.2 A lower bound of $2-\epsilon$

Given an online algorithm ON the request sequence we give will make a request at time ϵ after each service point that ON provides.
We denote the time of the request made after the i'st service point of ON is provided as r_i.
We denote the cost of servicing the all the requests until the firt service point that ON provided as C. We suppose that ON services the last request imidietly (otherwise the cost of ON will only increase). Therefore, the total cost of ON is $(C + n + t - (n-1)\epsilon)$.
If $t > n$, we choose an offline algorithm that provides a service point in each time r_i. The cost of this algorithm is $n + C$ and must be at least the cost of the optimal algorithm. Hence:

$$\frac{cost(ON)}{cost(OPT)} \ge \frac{C+n+t-(n-1)\epsilon}{n+C} = \frac{2n}{n} - \epsilon' = 2 - \epsilon'$$

When we denote $n' = n + c$, and assuming that C is small enough comparing to n and t.
If $t < n$:
We note that $t = sum_{1=2}^{n} r_i - r_{i-1}$. We can seperate this sum into the sum of the intervals between the odd

requests to the requests before them, and the sum of the intervals between the even requests to the requests before them. We note that one of the two sums must be at most t. Assume, without loss of generality, that the sum of the intervals between the even requests and the requests before them is at most t. The ofline algorithm will provide a service point at each even request, and therefore its cost will be at most $\frac{n}{2} + \frac{t}{2} + C - (n-1)\epsilon$. We therefore yield that:

$$\frac{cost(ON)}{cost(OPT)} \geq \frac{C + n + t - (n-1)\epsilon}{\frac{n}{2} + \frac{t}{2} + C - (n-1)\epsilon} = 2 - \epsilon'$$

1.3 Each service point can serve up to k Requests

Assume that for some fixed $k > 1$, each service point can serve up to k Requests.
We say that the competitive ratio shown in 1.a. still holds.
We slightly change the online agrotihm so that the algorithm provides a service point when ever $SumCurrentResTime = 1$ or the number of unserved requests equals k (what ever comes first). We now define the segment of the service point to be from the time of the first request that the service point services, until the time of the first request after the service point is provided (not including the time of the first request after).
It is easy to see that $cost(ON) \leq 2\#segments$. We now claim that the cost of each segment for OPT is at least one. If the segment is ended because $SumCurrentResTime = 1$, than the same arguments from 1.a. hold, and the segment costs at least 1 to OPT.
If the segment is ended because the number of unserved requests equals k than OPT must provide a service point durring that period of time (since after the segment the number of unserved requests is $k+1$, and at least one request will not be answered), and therefore the cost of the segment i also 1.
We yield that $cost(ON)/cost(OPT) \leq 2$, and the above algorithm is 2-competitive.

We say that the competitive ratio lower bound shown in 1.b. still holds. This is obviously true since except for the cost C that both algorithms pay at the beginning, for requests made before the first service point, the cost of both the ofline and the online algorithm does not change by the limitation of serving at most k requests. This is since both algorithms service at most 2 requests in every service point (but the first one), and we know that $k \geq 2$. For ON, it is since we choose the sequence to be one request after each service point, and for of as in both cases we choose each service point to serve at most 2 requests (either we put a service point on each request, or on each second request.

2 Exercise 2

We consider the caching problem, where the pages have different sizes, and the cost of bringing a page to the memory is equal to its size. We note that in this problem removing a part of a page P from the memort cause P to become unavailable. This means that in the next request for P, we will have to bring the entire page to the memory. Hence there is no point in removing parts of pages from the memory.
We mark s_{min} as the size of the smallest page, and s_{cache} as the size of the cache. We are asked to show an algorithm which is s_{cache}-competitve. Denote $k = \frac{s_{cache}}{s_{min}}$. We will show an algorithm which is k competitive, and is therfore also s_{cache}-competitve (as $k \leq s_{cache}$).

2.1 A Deterministic K-Compatible Algorithm

We view the Greedy Dual Size algorithm, when $cost(p) = size(p)$. We first note that in our case $\frac{cost(p)}{size(p)} = 1$.

```
1  initialize $L \leftarrow 0$
2  while ($P \leftarrow GetNextRequest) \neq NIL$ do
3      if $P$ is already in memory then
4          $H(P) \leftarrow L + 1$
5      else
6          while there is not enough room in the memory for $P$ do
7              $L \leftarrow min_{q \in M} H(q)$
8              evict q for which $H(q) = L$ from the memory
9          bring $P$ to memory
10         $H(p) \leftarrow L + \frac{cost(p)}{size(p)}$
```

We shall prove that the above algorithm (denote GDS) is k-competitive by eximining the flow of the optimal algorithm and of the DGS algorithm on a general input sequence of requests. After each request, first the optimal algorithm serves the request, and than the GDS serves the request.
In this analysis we charge each algorithm for the documents that it evicts, instead of charging it for the documents that it evicts. Note that the two cost measures have a difference of at most an additive constant.

Observation 1 *For every document q in the cache, it holds that:*

$$L \leq min_{p \in M} H(p) \leq H(q) \leq L + \frac{cost(q)}{size(q)} = L + 1$$

Proof: We can easily prove the observation using induction on the request sequence.
In the first request there is no page in the memory, $L = 0$, and the page inserted (mark p) is given a cost $H(p)$.
We assume that at the end of servicing the last request the statement held. This means that there are only pages at the cache for which $H(p) \in L, L+1$. For the new page p that we bring to the cache it holds that $H(p) = L + 1$ (lines 4 and 10). In line 7 we change L to be the minimum $H(p)$ for p in the cache, and there for it stays that for every p in the cache, $H(p) \in L, L+1$. There for the lemma hold at the end of servicing this request. ■

Notations
Denote L_{final} as the value of L at the end of the lagorithm (after serving the last request).
Mark s_{min} and s_{cache} as the size of the smallest task and of the cache, accordingly. Note that $k = \frac{s_{cache}}{s_{min}}$.

Lemma 2 *The cost of all pages evicted by the optimal algorithm is at most $s_{min} L_{final}$*

Proof: We first note that L is increased only in cases where we evict a page from the cach (line 7). Since the optimal algorithm is ran before the GDS algorithm than the page brought to the cache at that point is already in the cache of the optimal algorithm (for it has satisfied the request). Since there is no place in the chace of the GDS algorithm for the document, there is a page in the cache of the GDS algorithm which is not in the optimal algorithm. Hence, whenever L increases there is a page (denote p) in the cache of the GDS algorithm which is not in the cache of the optimal algorithm.
We also note that the GDS algorithm only brings a page to the cache if it is requested. This means that p was

once evicted from the cache of the optimal algorithm.
We view the time in which p is in the cache of the GDS algorithm, and is not in the cache of the optimal algorithm. We will atribute the increase in L durring this time to the cost of evicting p at the begining of the period for the optimal algorithm. Denote S as the group of all pages which are in the cache of the GDS algorithm and not in the cache of opt at some point in time. Therefore, if we show that L did not increase in more than $1 \leq size(p)/size_{min}$ during that time, then we get that the optimal cost is at least $size_{min}L_{final} \leq size_{min}sum_{p\in S}size(p)/size_{min}$.
We note that the period described above ends either when p is evicted from the GDS cache, or when p is brought back to the optimal cache. Denote L_{start} as the value of L at the begining of the time period. From observation we get that $L \leq H(q) \leq L+1$. if the period ends by Opt bringing p back, then H(p) does not change durring that time (as p is not evicted), and therefore from observation L does not increase in more than 1. Oterwise, $H(p)$ does not increase until p is evicted, and since $H(p)$ is that upper bound of L, L is increased by at most 1 durring the time stated. ■

Lemma 3 *The cost of all pages evicted by the GDS algorithm is at most* $s_{cache}L_{final}$

Proof: For each page p evicted by the GDS algorithm we view the time interval that p is in the cache. Let $L1$ and $L2$ be the values for L when it is brought in and evicted from the cache, respectively. We notice that $H(p) = L_1 + 1$ and $H(p) = L_2$ at the beginning and the end of the period, respectively. Since $H(p)$ can only increase when p is in the cache, we get that $L_2 - L_1 \geq 1 = cost(p)size(p)$. Therefore $size(p)L_2 - L_1 \geq cost(p)$ at the time interval. Since the size of all the pages in the cache in a given moment can not be greater than s_{cache} we achieve that the some of $size(p)L_2 - L_1$ is at most $s_{cache}L_{final}$, and therefore the cost of all pages evicted by the GDS algorithm is at most $s_{cache}L_{final}$. ■

Combining the above lemma we yield that the competitive ratio is: $\frac{costoftheGDSalgorithm}{costoftheGDSalgorithm} = \frac{s_{cache}}{s_{min}}$, which means that the above algorithm is k-competitive.

2.2 For all i we have $x_i = r$

In this case the problem becomes the same as the paging problem shown in class with a cache that can fit $\lfloor \frac{k}{r} \rfloor$ pages:
The number of pages that can fit fully into the cache in this case is $\lfloor \frac{k}{r} \rfloor$.
As stated in the question, removing a part of a page P from the memory will cause P to become unavailable, which means that in the next request for P, we will have to bring the entire page to the memory. Therefore, there is no added value to the remaining place in the cache, when it is filled with $\lfloor \frac{k}{r} \rfloor$ pages.
Hence, we can view this problem as the problem of serving requests for pages, when we have room for $\lfloor \frac{k}{r} \rfloor$ pages in the cache.
Since the cost of moving a page to the cache is constant for all of the pages (r), and we are trying to minimize the total cost, it is the same as minimizing the number of page-faults.
Hence, this is exactly the problem of minimizing the number of page faults for the paging problem which a cache that has room for $\lfloor \frac{k}{r} \rfloor$ pages.
Therefore, as we have seen in class, the Least Recently Used algorithm is $\lfloor \frac{k}{r} \rfloor$-compatitive for the problem of paging with a cache that can fit $\lfloor \frac{k}{r} \rfloor$ pages. This means that the Least Recently Used algorithm is $\lfloor \frac{k}{r} \rfloor$-compatitive for this problem.

3 Exercise 3

Supose that we are at any point in an island of an unknown plannar shape, and can walk in any continues curve in the island. Our goal is to reach the seashore using a curve of minimum length. We will know we are done when we reach the seashore.
We assume that we are at least one unit of distance from the seashore, and hence non additive constant is needed), and show a deterministric constant competitive algorithm:
Denote the distance between our starting position and the closest point at the seashore to that position as L_{OPT}. The optimal algorithm will go straight from the starting position to the closest point at the seashore, and therefore $cost(OPT) = L_{OPT}$.
We move in a series of step were in each step i we move from a cicle of radius x_{i-1} around our beginning position, to a cicle of radius x_i around our beginning position, in the fastest way (in the direction perendicular to the tangent), and cycle that circle (of radius x_i) looking for the seashore. We mark the beginning point as a circle of radius $x_0 = 0$, for convinience. Assume that $x_n < L_{OPT} \leq x_{n+1}$, than we must find the seashore by the end of the cycle in the $n+1$ step.
The cost of step i equals the cost of walking between the two circles $(x_i - x_{i-1})$ plus the cost of cycling the circle with radius x_i $(2\pi x_i)$. Therefore, we get that:

$$cost(ON) = \sum_{i=1}^{i=n+1} ((x_i - x_{i-1}) + 2\pi x_i) = x_{n+1} + \sum_{i=1}^{i=n+1} 2\pi x_i$$

And since $x_n < L_{OPT}$, we yield:

$$\frac{cost(ON)}{cost(OPT)} = \frac{x_{n+1} + 2\pi \sum_{i=1}^{i=n+1} x_i}{x_n}$$

We choose $x_i = q^i$, and get:

$$\frac{cost(ON)}{cost(OPT)} = \frac{q^{n+1} + 2\pi \frac{q(q^{n+1}-1)}{q-1}}{q^n} = q + \frac{2\pi(q^{n+1}-1)}{(q-1)q^{n-1}}$$

This expresion is maximal when $\frac{(q^{n+1}-1)}{(q-1)q^{n-1}}$ is maximal. Thefore, we want to find:

$$\sup_q \frac{(q^{n+1}-1)}{(q-1)q^{n-1}} = \sup_q \frac{q^2}{q-1} - \frac{1}{(q-1)q^{n-1}} = \sup_q \frac{q^2}{q-1}$$

Where in the last step the quotient with n drops since it is negligible for a large n. As we have seen in class, this expresion is maximal when $q = 2$, and we get $\frac{q^2}{q-1} = 4$, and hence:

$$\frac{cost(ON)}{cost(OPT)} \leq 2 + 2\pi 4 = 2 + 8\pi \approx 27.1327$$

The algorithm therefore cycles a circle with radius 2^i at step i, and is 27.1327-compatitive.

4 Exercise 4

4.1 MF is 5-Completitive

As in the proof for the regular list update problem, we view the potential function:

$$\Phi = \#(x,y) \parallel \{x\ apears\ before\ y\ in\ one\ list\ and\ after\ y\ in\ the\ other\ list\}$$

. We also show that $MF(\sigma) + \Phi \leq OPT(\sigma)$, by showing that in every step: $\Delta MF(\sigma) + \Delta\Phi \leq \Delta OPT(\sigma)$.
We examine the cost of MF is much larger than the cost of OPT. Assume X is the $a+1$ item in OPT list and the $b+1$ item in MF list. If OPT does not move three items than, as shown in class:

$$\Delta OPT(\sigma) = a$$

$$\Delta MF(\sigma) = b$$

. Also at most $2a - b$ pairs are effected by MF moving X. If OPT moves X, then even less pairs are effected.
In the current problem OPT can also move three items located before X to any place up until $a+1$. We view the effect of moving one item in such a manner. Only pairs which contain that item will be effected, and more strongly, since the item is moved within the range $[1, a+1]$, the movenemt will only effect its position comparing to the other items in $[1, a+1]$. This means that it can change the order of at most a pairs in OPT, and therefore increase $\Delta\Phi$ by at most a. Since three item can be moved, we get the $\Delta\Phi$ can increase by at most $3a$ from OPT moving three items. Hence, we yield that the total change in Φ can be at most $\Delta\Phi = (2a - b) + 3a$, and:

$$\Delta MF(\sigma) + \Delta\Phi = b + (2a - b) + 3a = 5a \leq \Delta OPT(\sigma)$$

.

4.2 A Lower Bound of $4 - O(\frac{1}{n})$

We view following ofline algorithm (denoted as OPT):
At every step, in addition to serving the request, the algorithm views the next 3 item requested in the sequence, and check for each of them if the following condition holds:
It is placed before the current requested item in the list, and after the first three places in the list.
The algorithm switches the items (of the next three requested items) that sutisfy the request with the items placed in the first three places in the list.
Given a deterministic online algorithm ON, we view the following request sequence and show that for this requence it holds that:

$$\frac{cost(ON)}{cost(OPT)} \geq 4 - O(\frac{1}{n})$$

At a given state of the list in OPT and in ON we view the 13 items that are last in On. We fist request the item of this 13 items which is positioned last in the list of OPT. Now ON will switch the position of up to 3 of the remaining 12 items for free. We then request one of the at least 9 items not switched by ON. Again, ON can switch up to 3 items from the 8 remaining for free, and in the next step we request one of the at least 5 items not switched by ON. And now ON can once again switch up to 3 items for free, which means that there is at least one item remaining which we did not choose yet, and was not moved for free by ON, and we request this item.
The sequence is there for a sequence that makes the four requests described above each time.
For conviniense, we denote the time interval of the request sequence as $4t$.
The cost of servicing the described four requests for OPT is at most $n+6$, as in each time, in servicing the first request, OPT moves the three items of the following requests to the beginning of the list, and therefore servicing the will cost 6. Hence, the total cost of OPT is at most $t(n+6)$.
The cost of ON:
We note that for each of the four items requested, the sum of the exchanges done on the item by ON and the position of the item in the list of ON when it is requested is at least $n - 13$. Therefore, the cost of each item requested to ON is at least $n - 13$, and the total cost of ON must be greater than $4t(n - 13)$.
Therefore:

$$\frac{cost(ON)}{cost(OPT)} \geq \frac{4(n-13)}{n+6} = 4 - \frac{76}{n+6} = 4 - O(\frac{1}{n})$$

And we yield that any deterministic online algorithm must be at least $4 - O(\frac{1}{n})$ competitive.

5 Exercise 5

We consider a variant of the sky rental problem, where after $(1+\gamma)M$ times of renting the skis, we get them for free ($\gamma > 0$).

5.1 Best Deterministic strategy

Denote t as the number of days in which we use the skis.
We view OPT and ON the optimal and online algoirthms, respectivly. The best deterministic strategy differs, depending on γ:
If $\gamma \geq 1$, then the algorithm buys the skis after M days (as the original algorithm).
If $\gamma \leq 1$, then the algorithm it to never buy the skies (always rent the skis, until t is over or we get the skis for free).
We first note that OPT always chooses to not buy the skis if $t < M$, and to buy the skis at the first day if $t \geq M$, no matter what γ is (as long as $\gamma > 0$). Therefore, the cost of OPT is always $cost(OPT) = min(t, M)$.
If $\gamma \leq M$: $cost(ON) = min(t, (1+\gamma)M)$, since we do not buy the skis. Hence, $\frac{cost(ON)}{cost(OPT)} = min(t, (1+\gamma)M)/min(t, M)$. The ratio is highest when $t \geq (1+\gamma)$, and we get $\frac{cost(ON)}{cost(OPT)} = (1+\gamma)$. Therefore, the competitive ratio of the algorithm is $(1+\gamma)$.
If $\gamma > M$, ON buys the skis after M days, and therefore:
if $t < M$: $cost(ON) = t = cost(OPT)$.
if $t \geq M$: $cost(ON) = 2M = 2cost(OPT)$.
Therefore, we get that the competitive ratio of the algorithm is 2.
We get that for every $\gamma > 0$, the competitive ratio of the algorithm described above is $min(2, (1+\gamma))$.

5.2 Lower Bound

For any given algorithm, we shall come-up with a series that it fails for.
Since the skis are given for free after time $(1+\gamma)M$, all the deterministic algorithms either don't buy the skies, or buy them at some time. Since we've already analysed the algorithm that does not buy the skis, it is enough to compute the minimal competitive ratio of an algorithm that buys the skis after time $T \leq (1+\gamma)$.
We view the serie of length T (specically chosen for this algorithm). As seen in class, we get:

$$\frac{cost(ON)}{cost(OPT)} = \frac{T+M}{min(T, M)} \geq 2$$

Therefore, the minimal competitive ratio of every deterministic algorithm is $min(2, (1+\gamma))$. Note that this is precisly the competitive ratio of the algorithm described above.

5.3 A randomized strategy

We note that if $\gamma \geq 1$ than we cannot benefit from renting the skis until $(1+\gamma)M$, as it is always better to buy the skis after M. Therefore, the options of the random algorithm and the optimal algorithm are identical to those in the problem which we have seen in class. Therefore we can use the algorithm which we studied and get a competitive ratio of $\beta = \frac{e}{1-e} \approx 1.58$.
If $\gamma < 1$:
We use a strategy similar to the one that we have studied in class, with one small diffrence. We note that we cannot benefit from buying the skis after time γM, and therefore we inspect only the probability of buying in the interval $[0, \gamma M]$.
For optimal performance with this approach, we wish that for every series of length t, we can hold a certain competitive-ratio β against OPT.
Let us denote by $F(t)$ the probability that we have not bought until time t.
For $0 \leq t \leq \gamma M$:

$$cost(t) = \int_0^t F(x)dx + [1 - F(t)]M$$

As shown in class, we get $F(t) = ce^{t/M} + \beta$, and since at time $t = 0$ the probability that we have bought must be zero ($F(0) = 1$), we get $F(t) = (1-\beta)e^{t/M} + \beta$. Since in the time $(1+\gamma)M$ we get the skis for free, it is left

to look on $\gamma M < t \leq (1+\gamma)M$:
Since the competitive ratio is required to be β, we get

$$cost(t) = \int_0^{\gamma M} F(x)dx + [1 - F(\gamma M)]M + F(\gamma M)[t - \gamma M] \leq \beta min(t, M)$$

We can find β according to the conditions at the extreme ($t = (1+\gamma)M$), considering that $F(t) = (1-\beta)e^{t/M} + \beta$:

$$M(1-\beta)e^{M\gamma/M} + \beta M\gamma - [M(1-\beta)e^0 + 0] + [1 - F(M\gamma)]M + F(M\gamma)M = \beta M$$

And we yield $\beta = \frac{e^\gamma}{e^\gamma - \gamma}$.
For $\gamma = 0.8$, we get $\beta 1.5612$. The probability function is therefore

$$F(t) = (1 - \frac{e^\gamma}{e^\gamma - \gamma})e^{t/M} + \frac{e^\gamma}{e^\gamma - \gamma}$$