

Algorithms working in rounds

- In the past 9 weeks, we have dealt solely with algorithms that answer question in the following form: given an input, what is . . . of the input?
- By this, the input is completely specified, and then the algorithm is asked to answer a question about an input.
- In many real world applications, algorithms cannot afford to wait for the whole input before making decisions.
- Thus the algorithm runs in turns: some input event happens, and the algorithm must react to the input event by making decisions.
- Most data structure problems are of this form, as well as many real problems.

ALG/TITCS L09-1

The ski-rental problem

Suppose you are about to go skiing for the first time in your life. You have two options:

- You might buy skis, using, say, \$300. Then you can use it as many times as you like.
- You might rent skis, using, say, \$30. Then you will have to make such a choice the next time.

The problem seems simple: of course to rent it!

But what if you know you're going to ski for 50 times? Then buying will be a better choice, because to rent for 50 times it costs \$1500, much more than \$300.

But what if you don't even know how many times you will go ski? . . .

ALG/TITCS L09-2

Ski-rental with guarantee

It is clear that either strategy is not going to be optimal:

- If you always rent, it might happen that you like skiing very much and ski many times.
- If you always buy at the first time, it might happen that you won't ski after that, and thus waste the remaining \$270.

Noticing that we can waste only \$270 by buying, we can do the following:

For the first 9 times, you rent skis. At the 10-th time, you buy skis.

Is this algorithm "optimal"? No . . . it might happen that you don't ski after the 10-th time and waste \$270.

ALG/TITCS L09-3

The guarantee

However, note that the algorithm is optimal *if you go ski for at most 9 times*.

On the other hand, if you go ski for at least 10 times, *the optimal algorithm has to spend at least \$300*. You only waste \$270 for the rental in the first 9 time.

Thus the amount you spent is at most $(\$300 + \$270)/\$300 = 19/10$ times of the amount needed by someone who know exactly how many times he go ski.

We say that the algorithm is 19/10-competitive.

Soon we will see that this is the best possible on-line algorithm for the ski-rental problem.

ALG/TITCS L09-4

The ski principle

- Note that "locally", any choice of buying skis without future information is always a costly choice: it costs much more than to rent skis.
- But once you had already made a good number of "cheap" choices, you can afford to make a "costly" choice a few times: the cost of the costly choice can be attributed to the "cheap" choice.
- This is generally known as the "ski principle": an expensive choice can be a good choice, as long as it reduces the long-term cost, **and you have already paid a lot previously**.

ALG/TITCS L09-5

On-line algorithm, competitiveness

- On-line algorithms are algorithms that has to make irrevocable decisions (whether to rent or buy skis) before the input is complete (how many time you go ski).
- In **competitive analysis**, we compare the cost or benefit results from running the algorithm with the best cost or benefit achievable when the whole input is known before decisions need to be made—the **off-line optimal**.
- Like approximation algorithms, we want to have a *ratio bound* ρ , i.e., we want the algorithm gains no less than $1/\rho$ times the off-line optimal benefit, or requires no more than ρ times the off-line optimal cost, for all input.
- We say an on-line algorithm is ρ -competitive if it has ratio bound of ρ .

ALG/TITCS L09-6

Adversary

- Competitive analysis is a **worst-case** analysis: We do not talk about what is the probability that a particular input occurs.
- That is, the algorithm must perform reasonably well no matter what is the input. It cannot say things like “because the input is probably not like this, we don’t need to consider it.”
- When showing lower bounds of competitiveness of all algorithms for some problems, we usually use the idea of an **adversary**.
- An adversary is some bad guy who look at the internal working of an algorithm and decide what input to give to the algorithm.
- In fact, the word “adversary” usually extends to show bad cases for a given algorithm as well.

ALG/TITCS L09-7

Lower bound

To show a lower bound on the competitiveness of a problem, we usually argue like this:

At the beginning, the adversary gives the algorithm the following events. Depending on the action taken by the algorithm, the adversary gives the algorithm the following events. . . . (repeated) At the end, the algorithm {need at least this cost/gain at most this benefit}. The offline algorithm would do it like this . . . and thus obtains that {cost/benefit}. Thus any algorithm can only have this/that competitiveness.

Note that, given an algorithm, it takes only one bad input to get a lower bound of the competitiveness. But the bad input can depend on the algorithm. So we want to use the abstraction of the adversary to specify the bad input.

ALG/TITCS L09-8

Example: lower bound of the ski-rental problem

To show how the adversary work, consider the following lower bound for any algorithm \mathcal{A} for ski-rental:

- The adversary tells \mathcal{A} that you go ski.
- As long as you don’t buy skis, the adversary add an event telling \mathcal{A} that you go ski again.
- Once you buy skis, the adversary stops.
- If you rents for n times, then the cost of \mathcal{A} is $\$30n + 300$.
- An off-line algorithm needs only pay $\$ \min(30(n + 1), 300)$.
- The competitiveness is thus no better than $(30n + 300) / \min(30(n + 1), 300) < 19/10$.

Thus, instead of considering all the possible input, we focus on one input—that is given by the adversary depending on the algorithm.

ALG/TITCS L09-9

Summary

- For many real-world problems, an algorithm has to produce output before the input is completed.
- Without the knowledge of future information, an on-line algorithm has to make irrevocable decisions.
- In competitive analysis, the performance of such on-line algorithm is defined by the worst-case ratio between the performance of the algorithm and an off-line algorithm.
- To argue that all on-line algorithms for a problem has competitive ratios of at least some number, we usually use the notion of an adversary.
- Depending on the decisions made by the algorithm, the adversary tries to make the difference between the algorithm and the best off-line decisions to be very large, thus concluding that any algorithm has that competitive ratio.

ALG/TITCS L09-10

Simple Example 2: Unbounded search

- Suppose you are in the middle (“origin”) of a road of zig-zag shape along the east-west direction.
- You want to go toilet, and you know that there is one in both sides.
- But you don’t know which one is closer, and you don’t have a bound on how far it should be at.
- At each turn of the zig-zag road, you can look forward to the next turn and see whether there is a toilet. But that’s the only thing that you can see.
- At the beginning, you see no toilet. You want to minimize the distance you need to walk before finding a toilet.

ALG/TITCS L09-11

Some bad algorithm

Here are some algorithm that is not competitive:

- Go (East/West) until you find a toilet.
It is a correct algorithm, you will find a toilet. **But it can be arbitrarily bad:** if the next toilet is 2 turns towards the East and 100 turns towards the West, and you decided to go West at first, then . . .
- Go East to the next turn. If no toilet is found, go back to origin, and go west to the 2nd turn. If no toilet is found, go back to the origin, and go east to the 3rd turn. Continue like this until we find a toilet.

This time it is a good algorithm if the toilet is near, but if the toilet is far, it becomes quite bad: if the next toilet is 100 turns towards both the East and the West, then you should have walked $(1 + 2 + \dots + 100) = 5050$ turns when you see the first toilet—while the optimal algorithm just need walking for 100 turns.

ALG/TITCS L09-12

A better algorithm

Now consider the following algorithm:

Go East to the next turn. If no toilet is found, go back to origin, and go west to the 2nd turn. If no toilet is found, go back to origin, and go east to the 4th turn. If no toilet is found, go back and go west to the 8th turn, and so on, until we find a toilet.

This time the algorithm is 9-competitive.

Note that the *ski-principle* is in play: going back is costly, but if we had already walked for some distance, we don't mind to pay.

We use another strategy to show the competitiveness: to reduce any bad case to another bad case of a specific form. This allows us to avoid the clumsy calculations required to consider all the possible cases.

ALG/TITCS L09-13

Reducing the consideration

Suppose we have a bad case for the algorithm, in which the toilet is found at distance $2^k + p$ from the original position, where k is an integer, and $2 \leq p \leq 2^k$. Then the algorithm will not perform better if the toilet is in fact at distance $2^k + 1$.

Proof: Note that the algorithm turn around for the same number of times for either inputs. Thus making the toilet a distance $p - 1$ closer to the origin will save the algorithm only $p - 1$.

The same holds for the best off-line strategy (go directly to that toilet). If originally the ratio between the on-line algorithm and the best off-line strategy is $L/(2^k + p)$, now the ratio becomes $(L - (p - 1))/(2^k + 1) > L/(2^k + p)$.

Therefore, from now on we only need to consider inputs that the toilet is found at a distance $2^k + 1$ from origin, for some integer k .

ALG/TITCS L09-14

The competitiveness

Of course, we can assume that the on-line algorithm actually turn away from the toilet at point 2^k , since otherwise we can put the toilet to the other side. This does not affect the off-line cost, but the algorithm would cost much more.

Now we can find a guarantee. Note that the off-line strategy can find the toilet by walking for $2^k + 1$. Our algorithm walked for $2(1 + 2 + \dots + 2^{k+1}) + 2^k + 1$. We get the ratio bound

$$\frac{2(1 + 2 + \dots + 2^{k+1}) + 2^k + 1}{2^k + 1} < 2^3 + 1 = 9$$

Exercise: can we do better if we know that a toilet is within 10 turns away in both directions?

ALG/TITCS L09-15

The paging problem

Many problems in the computer are actually on-line problems. Here is an example.

We have k pages of cache available. Each of the pages is capable to hold a page of memory. We have requests of the following form: read the i -th memory page. If that page is already in the cache, we do nothing. Otherwise, we have to replace a page in the cache by that memory page, with a cost of 1 to make the memory load. Initially the cache contains nothing. As usual, we want to minimize the total cost of the algorithm.

Of course, if we have all the requests in hand, we can use the algorithm that we have done in the quiz to find the optimal strategy.

But how to do it on-line, i.e., **when we do not know the future requests**? We don't know when pages will be accessed again!

ALG/TITCS L09-16

The algorithm FWF (Flush When Full)

Here is a simple algorithm called Flush When Full:

When we have a request for a page not in cache, we see whether the cache is full. If not, we load the page in an empty cache page. Otherwise, we flush the cache (i.e., throw away all contents), and start over again.

This algorithm seems to be rather stupid. But it turns out that it gives a very good performance guarantee. Indeed, it is the best that can be achieved by a deterministic algorithm: k -competitive.

In practice, nobody uses FWF: it is much inferior than other algorithms like the Least-Recently-Used strategy. That the competitive ratio of LRU is not better than FWF is because **the analysis does not consider "locality of reference"**.

ALG/TITCS L09-17

The guarantee of the algorithm

We break the request sequence based on the time when FWF flushes its cache:

The first phase contains just the first request. The second phase contains all the requests after the first request, up to and including the first flush. The third phase contains all the requests after the first flush, up to and including the second flush, etc.

Let's call the last request of the i -th phase as r_i .

It is clear that FWF costs k for each phase except the first: there are k cache pages, each of them will be loaded only once. For the first phase, it costs 1.

On the other hand, any algorithm must load at least one page in each phase. This directly implies the competitive ratio of k .

ALG/TITCS L09-18

Why any algorithm needs one load in each phase?

- For the first phase, it is trivially true, since the cache was empty before.
- For phase i , $i > 1$: Note that each such phase accessed k pages other than the page requested by r_{i-1} .
- Since the last phase ends by the request r_{i-1} , the page requested by r_{i-1} is in the cache for any algorithm at the beginning of phase i .
- But if we don't count the page storing the page requested by r_{i-1} , there are only $k - 1$ pages in the cache.
- So one of the k different requests in the phase causes a page replacement for any algorithm.

ALG/TITCS L09-19

No algorithm is better than k -competitive

We can show that no algorithm is better than k -competitive:

The adversary uses only $k + 1$ pages of memory. It works as follows:

- Every time, look at the algorithm to check what page is in cache. Request the page that is not in the cache. Repeat this arbitrarily long.

Clearly, the adversary guarantee that every page request the algorithm to load a page, requiring 1 unit of cost.

Of course we know what is the optimal: replace the page that will not be requested again for the longest. The question is, how good it performs?

ALG/TITCS L09-20

The performance of optimal algorithm

- For the first k new requests, it costs k .
- But once the cache is full, after a page x is replaced, the next replacement can only occur when x is needed again.
- We know that x is the "last element" to be requested again, meaning that all the other $k - 1$ pages are requested again before the next replacement.
- So the optimal algorithm requests a page no more often than once-per- k requests.

Now it is clear that no algorithm is better than k -competitive.

ALG/TITCS L09-21