Euclidean TSP is NP-hard [Papadimitriou, 1977]

Hamiltonian Cycle Problem: Given an unweighted graph G, does G contain a cycle that visits every vertex exactly once?

**Hardness of Approximation**

The general TSP is NP-hard to approximate

Claim: If P ≠ NP, then there is no polynomial time c-approximation algorithm for TSP.

Proof: Suppose A is a polynomial time c-approximate algorithm for TSP, we use A to solve Hamiltonian cycle problem.

Transform X:
Create G’ from G=<V, E> such that G’ has all edges. Assign weights to edges in G’ as follows:

\[ w(u, v) = \begin{cases} 
1 & \text{if } (u, v) \in G \\
1 + |V| & \text{if } (u, v) \notin G
\end{cases} \]

(i.e. using a non-existing edge is (much) worse than using an existing one)

Transform Y:
If |TSP(G’)| ≤ c*|V| then output “Yes”, return “No” otherwise.

Correctness of reduction:
Edges not in the original graph are so costly that there is a gap between cost of tour if G contains a Hamiltonian cycle (cost = u) and cost of tour if G doesn’t (cost > c*|V|)

**New topic: Online Algorithms**

For input sequences p_1p_2…pn (very large), an online algorithm must produce an output given p_1p_2…pi (without seeing pi+1…pn) for each i.

Example:
Page replacement in cache
P_1p_2…pn is a sequence of page requests made by a program, k is a cache (number of pages). At i-th page request, pi, the cache contains some k pages. If pi is not in cache (page fault), some page must be evicted from cache to make room for pi then pi is added to cache. The cost of a page replacement algorithm A on a sequence p_1p_2p_3…pn is f_A(p_1p_2p_3…pn) = the number of faults on p_1p_2…pn
Online algorithm must decide what page to evict without knowing the future request.

Examples of page replacement algorithms:

1. Least Recently Used (LRU):
   Evict page whose most recent request occurred furthest in the past (the least recently used page).

2. Least Frequently Used (LFU):
   Evict page that has been requested least often.


4. FIFO (First In First Out):
   Evict page that has been in cache longest.

How do we decide the “best” online algorithm?

1. Worst case performance
   \[
   \max \left( f_{LRU}(p_1, p_2, \ldots, p_n) = n, \quad f_{LFU}(p_1, p_2, \ldots, p_n) = n, \quad f_{FIFO}(p_1, p_2, \ldots, p_n) = n \right)
   \]

2. Average case performance
   m = total number of pages possibly requested. Expected number of page faults on sequence of randomly, uniformly, independently chosen pages:
   \[
   E[f_{LRU}(p_1, p_2, \ldots, p_n)] = (1-k/m)*n
   \]

3. Competitive Analysis
   How does online algorithm’s performance compare to best offline algorithm?
   Definition: An online algorithm A is c-competitive if exist b such that for all \(p_1, p_2, \ldots, p_n:
   \[
   f_A(p_1, p_2, \ldots, p_n) \leq c \cdot f_{OPT}(p_1, p_2, \ldots, p_n) + b
   \]
   where b is an arbitrary constant
   Theorem: LRU and FIFO are k-competitive where k is the cache size.
   Theorem: If A is a deterministic online algorithm for paging, then c >= k.