# CSE 431/531: Analysis of Algorithms (Summer 2023) Greedy algorithm

Chen Xu

June 13-20, 2023

# Updates on previous slide 6/6

Page 15 included a new example of master theorem case 3.

## Example 4

$T(n) = aT(n/b) + f(n), f(n) = 2^n$

- We would like to verify if the condition $a2^{n/b} \leq c2^n$ holds for some constant $c < 1$ and sufficiently large $n$.
- Take $c = 0.1, n_0 = \frac{-5-\log a}{1/b-1}$, for all $n \geq n_0$ we have

$$\begin{aligned} a2^{n/b} - c2^n &= 2^n(2^{\log a + n/b - n} - 0.1) \\ &\leq 2^n(2^{\log a + n_0/b - n_0} - 0.1) \\ &= 2^n(2^{-5} - 0.1) \\ &< 0 \end{aligned}$$

- Therefore the condition holds, we apply case 3, $T(n) = \Theta(2^n)$

# Outline

- General idea
- Fractional Knapsack problem
- Activity selection problem
- Offline caching
- ~~Huffman codes~~

# Decision problem vs optimization problem

- A decision problem is about yes/no.

## Subset Sum problem

**Instance:** Array $A$, value $v$
**Problem:** Does there exist a subset of $A$ such that it sums to $v$?

- An optimization problem is about values.

## Super Mario Bros problem

**Instance:** A SMB game $g$
**Problem:** How fast can you beat $g$?

# What is an optimum solution?

- A solution for a decision problem is a proof of evidence that a yes/no answer is valid. Usually the execution of a working algorithm is a proof. We call them constructive proof. There are non-constructive proofs as well. For example, the solution of **Subset-Sum** instance is a subset that was output by some algorithm and sums up to the value.
- There can be solutions for an optimization problem too. Among all solutions the optimum solution gives the **globally optimized value** according to the demand of the problem. The input sequence of your controller that gives the lowest time to clear the game is the optimum solution to the **SMB problem**.

# What is a sub-problem?

- A sub-problem of a problem usually solves the same problem as the origin but on a restricted/conditioned/transformed input from the origin.
- a sub-problem of **Subset Sum** problem can be:
  1. Solve on the same array but with a different target value
  2. Solve on all subsets of size 3 with the same target value
  3. Solve on all subsets that include the element $A_1$ with the target value $v - A_1$
  4. ...
- From a perspective of algorithm, calling a recursive function on the algorithm itself is a form of trying to solve the sub-problem.
- Let us denote the sub-problem $P_{sub}$ of a problem $P$ as $P \prec P_{sub}$. Called $P$ reduces to its sub-problem $P_{sub}$.

# What is an induction of sub-problems?

- An induction of sub-problems is when you have a well defined sequence of problems such that $P_0 \prec P_1 \prec P_2 \prec ... \prec P_n$.
- Take an example, the sum of array problem, define the sub-problem $Sum_i$ as sum the input array from $A_1$ to $A_i$. Then we have an induction sequence $Sum_n \prec Sum_{n-1} \prec Sum_{n-2} \prec ... \prec Sum_1$

# Mapping solutions to sub-problems

- We can map solutions of a problem to the solution of its sub-problems too. The exact mapping depends on the induction of problems.
- For example, a solution of **Subset Sum** problem $\{\{1,2,3,4,5,6\},10\}$ can be $\{1,2,3,4\}$.
  We can map this solution to $\{2,3,4\}$ of its sub-problem $\{\{2,3,4,5,6\},9\}$.
- A mapping is $M$-**local** if you can have an algorithm $M$ compute this mapping easily. The above mapping from $\{1,2,3,4\}$ to $\{2,3,4\}$ can be easily computed. You can write an algorithm that only drops the first element because your induction does so. We omit $M$ if your algorithm is trivial given the induction of problems. i.e. Part of your solution is fixed.
- It is relatively hard for you to map $\{1,2,3,4\}$ to $\{3,6\}$ which also equals the sum of $\{2,3,4\}$. Therefore this solution is not a local solution to $\{1,2,3,4\}$.

# What it means to be greedy

- We made sense of solution locality. And we have defined the induction of problems. We can therefore explore the two important properties of greedy algorithms.
  1. **Greedy Choice Property(GCP):** Your algorithm always maps the local solution to the optimal solution of the sub-problem. And your solution is always locally optimal. In short, **if optimal, then there is one achieved by our mapping**.
  2. **Optimal Substructure Property(OSP):** In the induction of the sub-problems, through the mapping, every solution for sub-problem is optimum. In short, **if optimum, then the induced is optimum**.
- To prove the correctness of a greedy algorithm, we may prove the above two properties.

# Fractional Knapsack Problem

## Fractional Knapsack Problem

**Input:** A set of items associated with values and weights $I = \{(v_1, w_1), (v_2, w_2), ..., (v_n, w_n)\}$. A weight limit $W_{max}$

**Output:** The quantity $x_i$ of each item $i$ to include, $0 \leq x_i \leq 1$, such that the total value $V = \sum_{i \in I} v_i \cdot x_i$ is maximized while $W = \sum_{i \in I} w_i \cdot x_i \leq W_{max}$

# Proposed greedy algorithm

## Greedy Fractional Knapsack

1: Calculate $d_i = \frac{v_i}{w_i}$ for each item $i$, where $v_i$ and $w_i$ are the value and weight of item $i$ respectively.
2: Sort the items by $d_i$ in descending order.
3: **for** $i = 1$ to $n$ **do**
4: **if** $W \geq w_i$ **then**
5: Add item $i$ completely in the knapsack.
6: $W = W - w_i$.
7: **else**
8: Add fraction $f = \frac{W}{w_i}$ of item $i$ in the knapsack.
9: $W = 0$.

Running time is $O(n \log n)$.

# An example

Suppose we have a knapsack with a maximum weight capacity of 50 units, and we have five items:

| Item | Value | Weight | Value-to-Weight Ratio | Fraction |
|---|---|---|---|---|
| 1 | 60 | 10 | 6 | 1 |
| 5 | 50 | 10 | 5 | 1 |
| 2 | 100 | 20 | 5 | 1 |
| 4 | 90 | 20 | 4.5 | 0.5 |
| 3 | 120 | 30 | 4 | 0 |

Table: Item Table

- We can take all of item 1, 5, and 2, half of item 4 and none of 3.
- The maximum value we can get is
  $60 * 1 + 50 * 1 + 100 * 1 + 90 * 0.5 + 120 * 0 = 255$.

# Proving the Greedy Choice Property

Let's talk about our mapping. we sorted all items by value-to-weight ratio in descending order.

- Our sub-problem induction is defined as $P_{I_k}$ where $I_k$ is the knapsack at step $k$, $P_{I_k}$ asks what is the solution to the remaining items.
- Our solution mapping is from $I_k$ to $I_{k+1}$ where $I_{k+1}$ catches the highest value-to-weight ratio item in the remaining pile.

Remember the **GCP** is that **if optimal, then there is one achieved by our mapping.** There are two ways of showing it. One is direct proof, the other one is by contradiction.

# Proving the Greedy Choice Property

Remember the **GCP** is that **if optimum, then there is one achieved by our mapping.** There are two ways of showing it. One is direct proof, the other one is by contradiction. Let's show it by contradiction. The contradition logic is **if optimum and it is not achieved by our mapping, then it is not optimum.**

- Assume there exists an optimum solution $I$ that doesn't include the item with the highest value-to-weight ratio, let $g$ be the item with the highest value-to-weight ratio, and let $o$ be the first item in the optimum solution that has a lower value-to-weight ratio than $g$.
- If we replace $o$ with $g$ in $I$, the total value increases and the total weight remains under the capacity (since both $o$ and $g$ are fractional, we can take the same weight of $g$ as $o$).
- Hence, we can obtain a better solution than $I$, which contradicts the assumption that $I$ is an optimum solution.

Exercise: Write the above proof mathematically!

# Proving the Optimal Substructure Property

Let's recall what our induction is:

- $P_{i_k}$ was induced from $P_{i_{k-1}}$
- We just took one best item $i_k$ at step $k$.

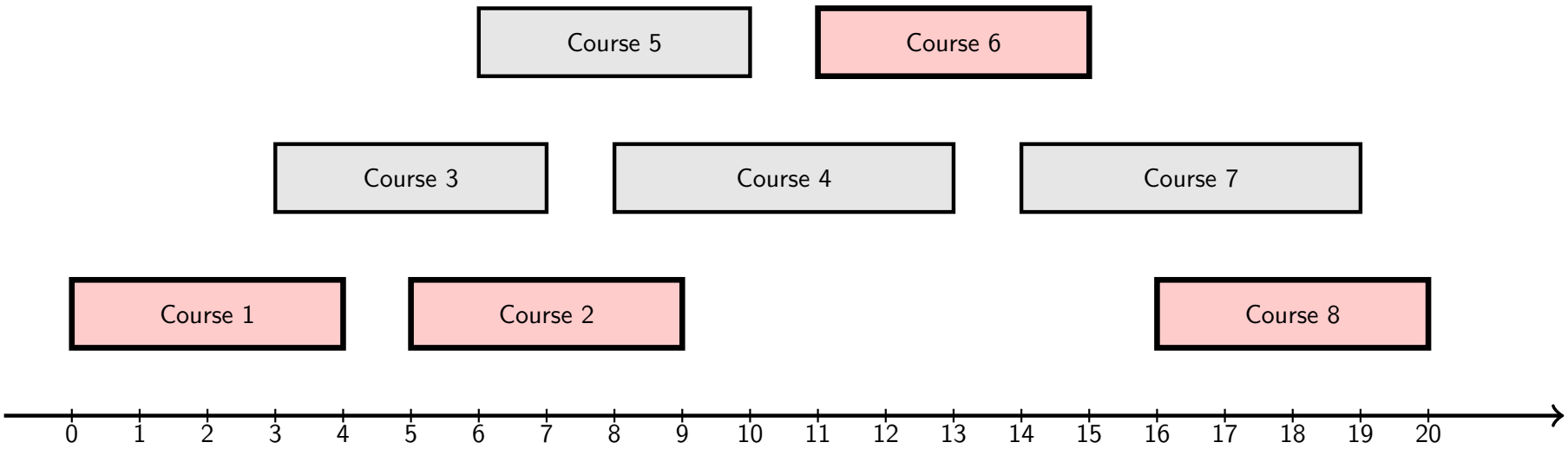
The **OSP** idea is to show that **if optimum, then the induced is optimum.** This means that we want to show that if we have the optimum solution is $I_k$ to the problem of $P_k$, then $I_k \setminus i_k$ is the optimum solution to $P_{I_{k-1}}$. Again, we have two ways of showing it. Let's show by contradiction.

- Let $I$ be the optimum solution with value $V$ to problem $P_{I_k}$ with knapsack capacity $W$ and fraction $X$.
- Let $I' = I \setminus i_k$ be a solution to $P_{I_{k-1}}$ with $X' = X - x_k$ fraction $V' = V - x_k * w_k$ value and weight $W' = W - w_k$.
- We assume that $I'$ is not optimum to $P_{I_{k-1}}$ and that we have another solution $I''$ to $P_{I_{k-1}}$ that has a higher total value $V'' > V'$. Then $I'' \cup i_k$ is a solution to $P_{i_k}$ with a higher value $V'' + x_k * w_k > V' + x_k * w_k = V$. This means $I$ is not an optimum solution which contradicts our assumption.

# Activity Selection problem

- Imagine there are 100 courses available next semester. Each course has a start time and a finish time, say, 10 a.m. to 11:20 a.m. but it only has one session per week.
- There are many conflicts. You cannot take two conflicted courses at the same time.
- You want to take as many courses as possible. How do you pick the courses?

# Activity Selection problem

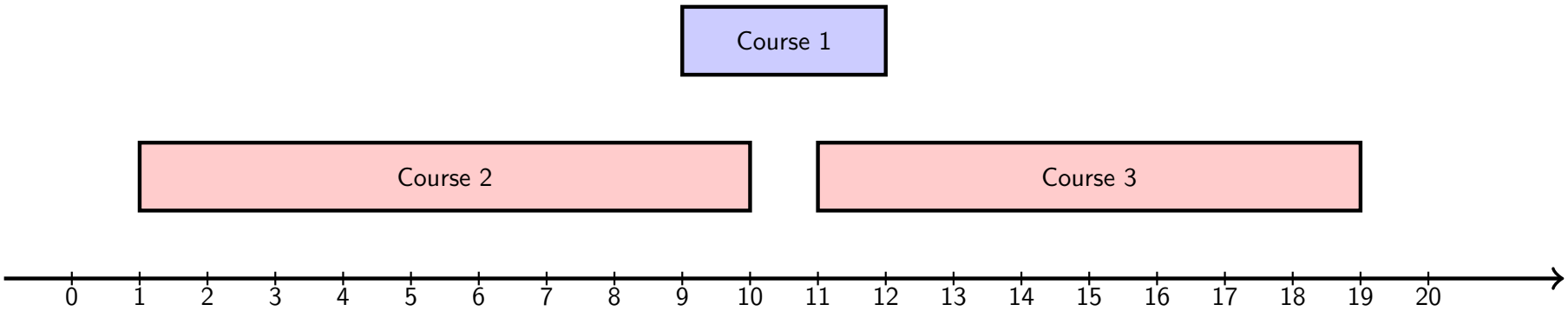



In this example, we have courses (0,4),(5,9),(3,7),(8,13),(6,10),(11,15),(14,19),(16,20). We can pick at most 4 courses.

# Which greedy algorithm is correct?

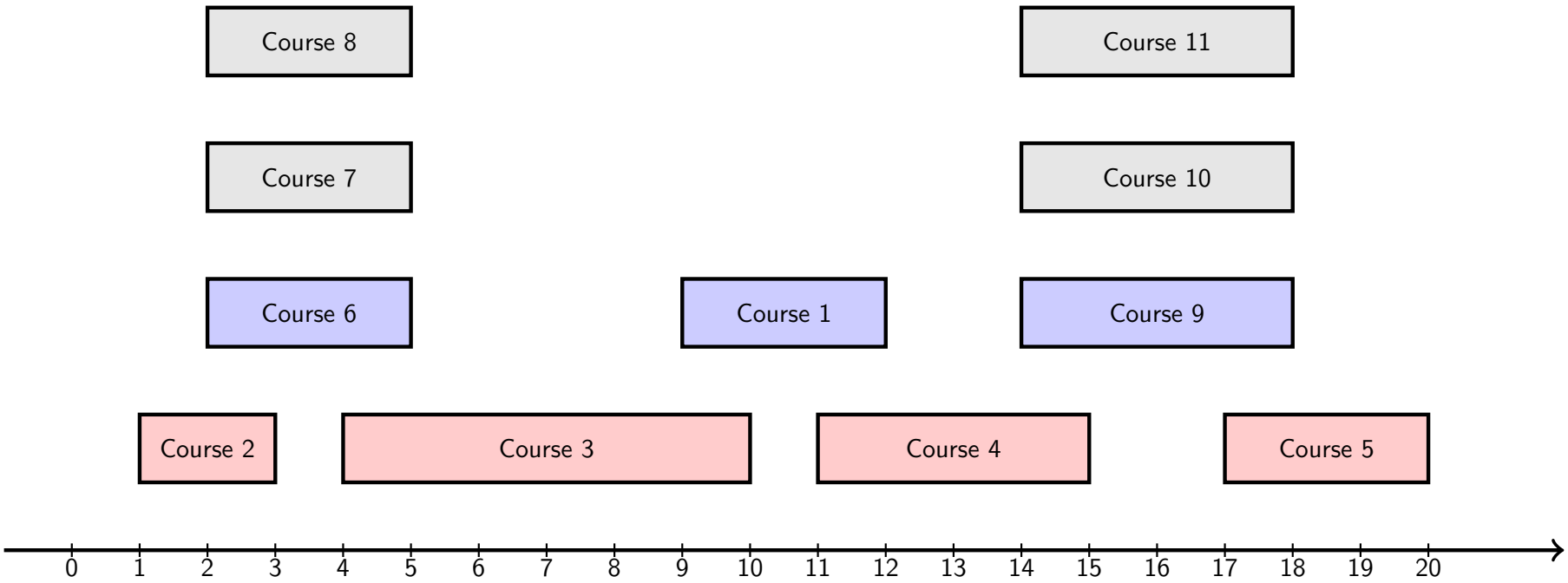
- We can have several greedy algorithms. For example:
  1. Always add the one that is not conflict and has the shortest duration.
  2. Always add the one that has the fewest conflicts with the rest of the activities.
  3. Always add the one from the remaining non-conflicting activities that has the earliest start time.
  4. Always add the one from the remaining non-conflicting activities that has the earliest finish time.
- Which one is correct?

# Counter example of picking the shortest



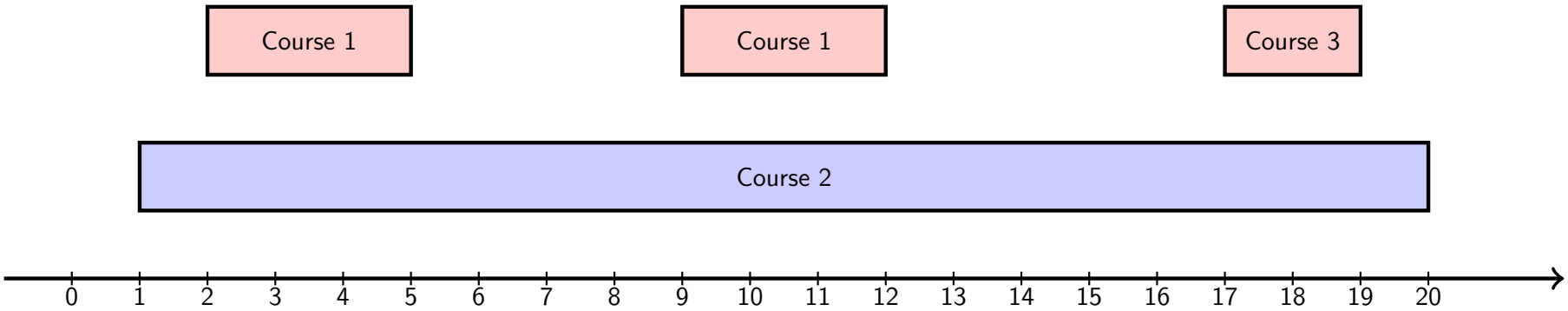

- Suboptimum: 1
- Optimum: 2

# Counter example of picking the fewest conflicts




- Suboptimum: 3
- Optimum: 4

# Counter example of picking the earliest




- Suboptimum: 1
- Optimum: 3

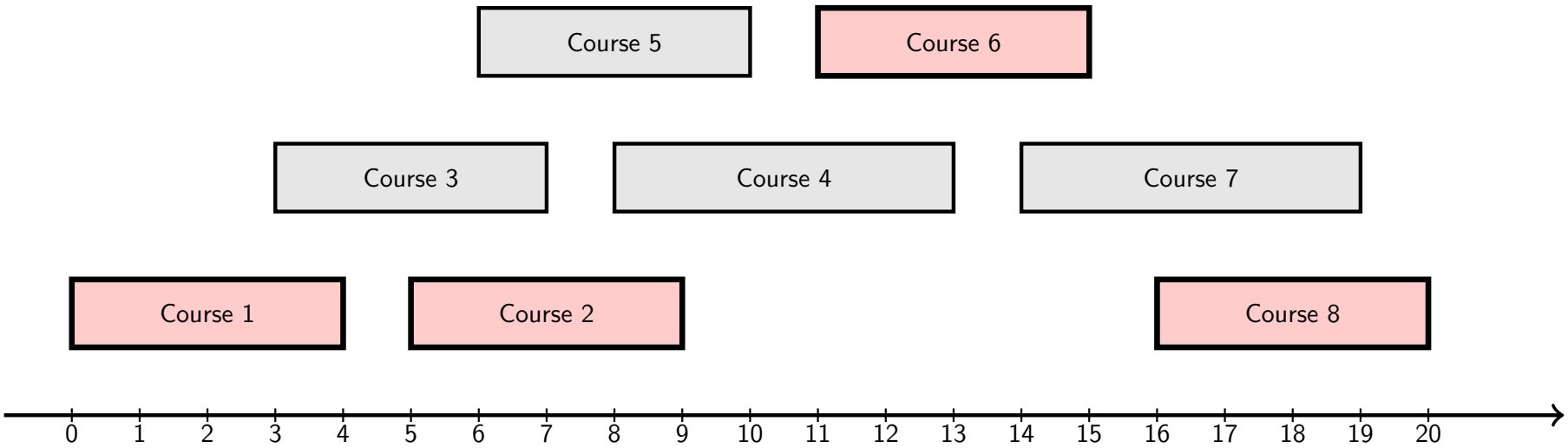
# The correct algorithm

- If the activities are not sorted by the finish time, sort them. This takes $O(n \log n)$.
- For each step, pick the non-conflicting activities that has the earliest finish time.
- Repeat until done.

# Proof of GCP

- The induction of the problem is the sequence of problem that focuses on the subset of activities from the finish time $f_k$ of the last picked activity to the end, denoted as $P_{f_k}$.
- The mapping is from the solution of $P_{f_{k-1}}$ to the solution of $P_{f_k}$, where you increase the $f_{k-1}$ to the earliest finish time $f_k$ of the next non-conflicting activity .

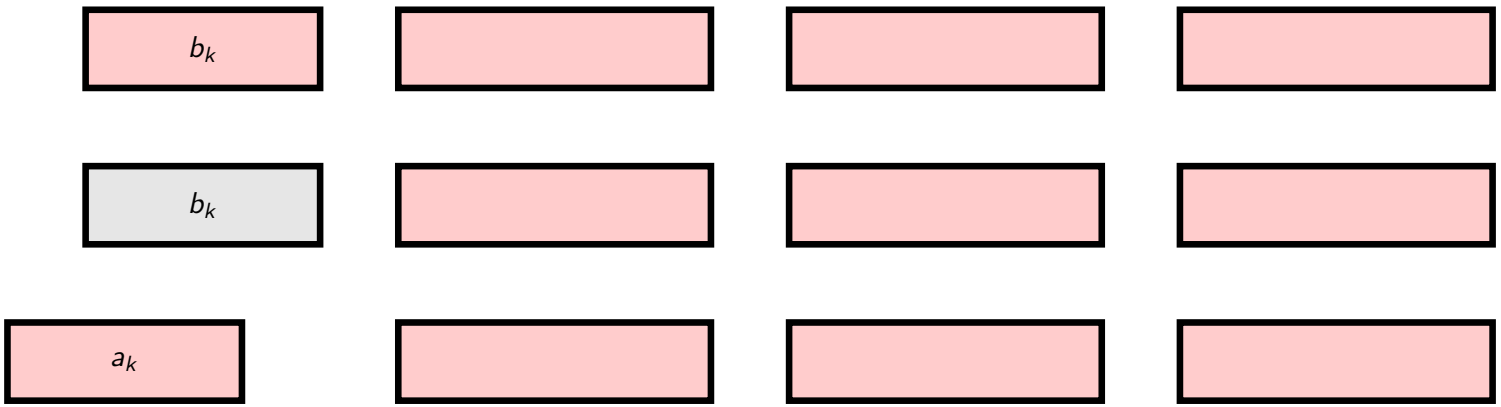
# Proof of GCP




- $P_{f_1} = \{\text{Course } 1,2,3,4,5,6,7,8\}$ Solution: $\{\text{Course } 1,2,6,8\}$ $f_1 = 4$
- $P_{f_2} = \{\text{Course } 2,4,5,6,7,8\}$ Solution: $\{\text{Course } 2,6,8\}$ $f_2 = 9$
- $P_{f_3} = \{\text{Course } 6,7,8\}$ Solution: $\{\text{Course } 6,8\}$ $f_3 = 15$
- $P_{f_4} = \{\text{Course } 8\}$ Solution: $\{\text{Course } 8\}$ $f_4 = 20$

# Proof of GCP

- This time, we show GCP by direct proof. The target is to show **If optimal, then there is one that can be achieved by our mapping**.
- For any $P_{f_k}$, suppose one of the optimum solution is $S$.
- If $S$ contains the earliest finish activity $a_k$, done.
- If $S$ does not contain the earliest finish activity $a_k$. Let's say the earliest finish activity in $S$ is $b_k$. We can replace it with $a_k$. The new solution $S'$ is optimum and contains $a_k$, done.



# Proof of OSP

- We still show the OSP by contradiction. The target is to show **if optimum, then the induced is optimum**.
- Let $S$ be one optimum solution for $P_{f_k}$. Let $a_k$ be the first activity in $S$.
- We need to show $S - \{a_k\}$ is the optimum solution for $P_{f_{k+1}}$
- Suppose the optimum solution for $P_{f_{k+1}}$ is not $S - \{a_k\}$ but another $S'$ that $|S'| > |S - \{a_k\}|$ then $S' + \{a_k\}$ should be the optimum solution for $P_{f_k}$ and $|S' + \{a_k\}| > |S|$. Then $|S|$ is not an optimum solution for $P_{f_k}$ which contradicts our assumption.

# Caching Problem

- A system with cache access usually performs much faster than system with no cache.
- Memory is cheap but slow. Cache is fast but expensive.
- Cache acts as a mirror of the memory and stores copy of the data in pages. When the reading or writing requests happen, the CPU will first ask if the cache has that page.
  1. If the requested page is in the cache, then the cache hits. CPU will access the data in the cache.
  2. If the requested page is not in the cache, then the cache misses. CPU may or may not make changes to the cache strategically. The changes can be evicting the existing page and swapping in a new page.
- The goal is to find the best strategy that minimizes cache misses.

# Offline Caching problem

## Offline Caching

**Input:** $k$: Cache size
$P = \{p_1, p_2, ..., p_T\} \in [n]^T$: Sequence of page requests

**Output:** $I = \{i_1, i_2, ..., i_T\} \in \{hit, empty\} \cup [n]$: indices of pages to evict
The hit means there is no evicting happening. The empty means evicting the empty page.

# Online and Offline Caching

- Offline caching is when we already see the whole future sequence. We can make decisions based on future.
- Online caching is that we can only make decisions based on history cache usage. This is a more realistic situation.
- We study offline version to determine if an online version is good or not. Later, we will show that the offline version has an optimum greedy solution.

# Some potential greedy algorithms

- Online:
  1. FIFO: first in first out, treat the cache like a cyclic buffer, always evict the first one.
  2. LRU: least recently used, evict the one whose usage was the earliest.
  3. LFU: least frequently used, evict the one that has the least frequent usage.
- Offline:
  1. FF: Furthest-in-Future, evict one that is not requested until the furthest in future.
- FF gives optimum solution.

# Example of FF

| | 1 | 5 | 4 | 2 | 5 | 3 | 2 | 4 | 3 | 1 | 5 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | | | | | | | | | |
| | | 5 | 5 | | | | | | | | | |
| | | | 4 | | | | | | | | | |
| | x | x | x | | | | | | | | | |

# Example of FF

| | 1 | 5 | 4 | 2 | 5 | 3 | 2 | 4 | 3 | 1 | 5 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 2 | 2 | | | | | | | |
| | | 5 | 5 | 5 | 5 | | | | | | | |
| | | | 4 | 4 | 4 | | | | | | | |
| | x | x | x | x | | | | | | | | |

# Example of FF

| | 1 | 5 | 4 | 2 | 5 | 3 | 2 | 4 | 3 | 1 | 5 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | | | |
| | | 5 | 5 | 5 | 5 | 3 | 3 | 3 | 3 | | | |
| | | | 4 | 4 | 4 | 4 | 4 | 4 | 4 | | | |
| | x | x | x | x | | x | | | | | | |

# Example of FF

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 5 | 4 | 2 | 5 | 3 | 2 | 4 | 3 | 1 | 5 | 3 |
| | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 5 | 5 |
| | | 5 | 5 | 5 | 5 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| | | | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| | x | x | x | x | | x | | | | x | x | |

# Proof of GCP

- The induction of the subproblem is

## Offline Caching subproblem from step $t$ to $T$

**Input:** $k$: Cache size
$P = \{p_1, p_2, ..., p_T\} \in [n]^T$: Sequence of page requests
$C = \{c_1, c_2, ..., c_k\} \in \{empty\} \cup [n]$: current pages in the cache.
**Output:** $I = \{i_1, i_2, ..., i_t\} \in \{hit, empty\} \cup [n]$: indices of pages to evict
The hit means there is no evicting happening. The empty means evicting the empty page.

- Our solution mapping is taking the FF strategy to update the cache page of step $t$ to $t + 1$.

# Proof of GCP

- We want to use the contradiction proof. The outline is again **Assume optimum, then we can get one that is not worse than our assumption, thus our assumption is not optimum.**
- Let $S$ be any optimum solution at step $t$. Assume that this step we took a page $p'$ that is not the furthest $p^*$ in the future. This means $p' < p^*$. Then create solution $S'$ at step $t$, $S$ and $S'$ differ in just one page. Then from step $t$ to the step before that $p'$ is requested there can be two cases:
  1. That different page has been evicted at the same time in $S$ and $S'$ . Then $S$ and $S'$ are both optimum.
  2. That different page has not been evicted until $p'$ is requested. Then $S'$ hits while $S$ misses. $S'$ is better than $S$. Contradiction.

# Proof of OSP

- Proof of OSP is trivial. It is almost identical to activity selection.