Greedy Algorithms

授课老师: 栗师
南京大学计算机科学与技术系
**Def.** In an **optimization problem**, our goal is to find a valid solution with the minimum cost (or maximum value).

**Trivial Algorithm for an Optimization Problem**

Enumerate all valid solutions, compare them and output the best one.

However, trivial algorithms often run in exponential time, as the number of potential solutions is often exponentially large.

$f(n)$ is a polynomial if $f(n) = O(n^k)$ for some constant $k > 0$.

**Convention:** polynomial time = efficient

**Goals of algorithm design**

1. Design efficient algorithms to solve problems
2. Design more efficient algorithms to solve problems
Def. In an optimization problem, our goal is to find a valid solution with the minimum cost (or maximum value).

Trivial Algorithm for an Optimization Problem
Enumerate all valid solutions, compare them and output the best one.
Def. In an *optimization problem*, our goal is to find a valid solution with the minimum cost (or maximum value).

**Trivial Algorithm for an Optimization Problem**
Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in **exponential** time, as the number of potential solutions is often exponentially large.
**Def.** In an optimization problem, our goal is to find a valid solution with the minimum cost (or maximum value).

**Trivial Algorithm for an Optimization Problem**
Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
Def. In an **optimization problem**, our goal is to find a valid solution with the minimum cost (or maximum value).

**Trivial Algorithm for an Optimization Problem**

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in **exponential** time, as the number of potential solutions is often exponentially large.
- \( f(n) \) is a **polynomial** if \( f(n) = O(n^k) \) for some constant \( k > 0 \).
Def. In an optimization problem, our goal is to find a valid solution with the minimum cost (or maximum value).

Trivial Algorithm for an Optimization Problem
Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
- $f(n)$ is a polynomial if $f(n) = O(n^k)$ for some constant $k > 0$.
- convention: polynomial time = efficient
**Def.** In an **optimization problem**, our goal is to find a valid solution with the minimum cost (or maximum value).

**Trivial Algorithm for an Optimization Problem**
Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in **exponential** time, as the number of potential solutions is often exponentially large.
- $f(n)$ is a **polynomial** if $f(n) = O(n^k)$ for some **constant** $k > 0$.
- convention: polynomial time = **efficient**

**Goals of algorithm design**
Def. In an **optimization problem**, our goal is to find a valid solution with the minimum cost (or maximum value).

**Trivial Algorithm for an Optimization Problem**

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in **exponential** time, as the number of potential solutions is often exponentially large.
- \( f(n) \) is a **polynomial** if \( f(n) = O(n^k) \) for some constant \( k > 0 \).
- convention: polynomial time = **efficient**

**Goals of algorithm design**

1. Design efficient algorithms to solve problems
Def. In an optimization problem, our goal of is to find a valid solution with the minimum cost (or maximum value).

**Trivial Algorithm for an Optimization Problem**
Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in exponential time, as the number of potential solutions is often exponentially large.
- $f(n)$ is a polynomial if $f(n) = O(n^k)$ for some constant $k > 0$.
- convention: polynomial time = efficient

**Goals of algorithm design**
1. Design efficient algorithms to solve problems
2. Design more efficient algorithms to solve problems
Common Paradigms for Algorithm Design

- Greedy Algorithms
- Divide and Conquer
- Dynamic Programming
Common Paradigms for Algorithm Design

- Greedy Algorithms
- Divide and Conquer
- Dynamic Programming

Greedy algorithms are often for optimization problems.
Common Paradigms for Algorithm Design

- Greedy Algorithms
- Divide and Conquer
- Dynamic Programming

Greedy algorithms are often for optimization problems.
They often run in polynomial time due to their simplicity.
Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a "reasonable" strategy
Greedy Algorithm

- Build up the solutions in steps
- At each step, make an *irrevocable* decision using a “reasonable” strategy

A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe”
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem
Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a “reasonable” strategy

A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)
Greedy Algorithm

- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

**Def.** A strategy is safe: there is always an optimum solution that agrees with the decision made according to the strategy.
Outline

1. Toy Example: Box Packing
2. Interval Scheduling
3. Scheduling to Minimize Lateness
4. Weighted Completion Time Scheduling
5. Offline Caching
   - Heap: Concrete Data Structure for Priority Queue
6. Data Compression and Huffman Code
7. Summary
Box Packing

**Input:** \( n \) boxes of capacities \( c_1, c_2, \ldots, c_n \)
m items of sizes \( s_1, s_2, \ldots, s_m \)

Can put at most 1 item in a box

Item \( j \) can be put into box \( i \) if \( s_j \leq c_i \)

**Output:** A way to put as many items as possible in the boxes.
Box Packing

**Input:** \( n \) boxes of capacities \( c_1, c_2, \cdots, c_n \) 
\( m \) items of sizes \( s_1, s_2, \cdots, s_m \)
Can put at most 1 item in a box
Item \( j \) can be put into box \( i \) if \( s_j \leq c_i \)

**Output:** A way to put as many items as possible in the boxes.

**Example:**
- Box capacities: 60, 40, 25, 15, 12
- Item sizes: 45, 42, 20, 19, 16
- Can put 3 items in boxes: 45 → 60, 20 → 40, 19 → 25
Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrerevocable decision using a “reasonable” strategy

Q: Take box 1. Which item should we put in box 1?
A: The item of the largest size that can be put into the box.
Greedy Algorithm

- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

Designing a Reasonable Strategy for Box Packing

- Q: Take box 1. Which item should we put in box 1?
Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a “reasonable” strategy

Designing a Reasonable Strategy for Box Packing

- Q: Take box 1. Which item should we put in box 1?
- A: The item of the largest size that can be put into the box.
A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe”
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem
A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe”
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem

Lemma The strategy that put into box 1 the largest item it can hold is “safe”: There is an optimum solution in which box 1 contains the largest item it can hold.
A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe”
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem

**Lemma**  The strategy that put into box 1 the largest item it can hold is “safe”: There is an optimum solution in which box 1 contains the largest item it can hold.

- Intuition: putting the item gives us the easiest residual problem.
A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe”
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem

Lemma  The strategy that put into box 1 the largest item it can hold is “safe”: There is an optimum solution in which box 1 contains the largest item it can hold.

- Intuition: putting the item gives us the easiest residual problem.
- formal proof via exchanging argument:
Lemma  There is an optimum solution in which box 1 contains the largest item it can hold.
Lemma  There is an optimum solution in which box 1 contains the largest item it can hold.

Proof.

Let $j =$ largest item that box 1 can hold.
Lemma  There is an optimum solution in which box 1 contains the largest item it can hold.

Proof.

- Let $j =$ largest item that box 1 can hold.
- Take any optimum solution $S$. If $j$ is put into Box 1 in $S$, done.
**Lemma** There is an optimum solution in which box 1 contains the largest item it can hold.

**Proof.**
- Let \( j = \) largest item that box 1 can hold.
- Take any optimum solution \( S \). If \( j \) is put into Box 1 in \( S \), done.
- Otherwise, assume this is what happens in \( S \):
**Lemma** There is an optimum solution in which box 1 contains the largest item it can hold.

**Proof.**

- Let $j =$ largest item that box 1 can hold.
- Take any optimum solution $S$. If $j$ is put into Box 1 in $S$, done.
- Otherwise, assume this is what happens in $S$:

  ![Diagram](image)

  - $s_{j'} \leq s_j$, and swapping gives another solution $S'$.
**Lemma** There is an optimum solution in which box 1 contains the largest item it can hold.

**Proof.**
- Let $j =$ largest item that box 1 can hold.
- Take any optimum solution $S$. If $j$ is put into Box 1 in $S$, done.
- Otherwise, assume this is what happens in $S$:

```
S': // S' is a possible arrangement of items
box 1

item $j'$  // Item $j'$ and $j$ are placed in box 1.

item $j$

......
```

- $s_{j'} \leq s_j$, and swapping gives another solution $S'$
- $S'$ is also an optimum solution. In $S'$, $j$ is put into Box 1. □
Notice that the exchanging operation is only for the sake of analysis; it is not a part of the algorithm.
Notice that the exchanging operation is only for the sake of analysis; it is not a part of the algorithm.

A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe”
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem
Notice that the exchanging operation is only for the sake of analysis; it is not a part of the algorithm.

A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe”
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem

Trivial: we decided to put Item $j$ into Box 1, and the remaining instance is obtained by removing Item $j$ and Box 1.
**Generic Greedy Algorithm**

1. **while** the instance is non-trivial **do**
2. make the choice using the greedy strategy
3. reduce the instance

**Greedy Algorithm for Box Packing**

1. \( T \leftarrow \{1, 2, 3, \ldots, m\} \)
2. **for** \( i \leftarrow 1 \) to \( n \) **do**
3. \[ \text{if some item in } T \text{ can be put into box } i \text{ then} \]
4. \( j \leftarrow \) the largest item in \( T \) that can be put into box \( i \)
5. print(“put item \( j \) in box \( i \)”)
6. \( T \leftarrow T \setminus \{j\} \)
Exchange argument: Proof of Safety of a Strategy

1. let $S$ be an arbitrary optimum solution.
2. if $S$ is consistent with the greedy choice, done.
3. otherwise, show that it can be modified to another optimum solution $S'$ that is consistent with the choice.
Exchange argument: Proof of Safety of a Strategy

- let $S$ be an arbitrary optimum solution.
- if $S$ is consistent with the greedy choice, done.
- otherwise, show that it can be modified to another optimum solution $S'$ that is consistent with the choice.

The procedure is not a part of the algorithm.
Outline

1 Toy Example: Box Packing
2 Interval Scheduling
3 Scheduling to Minimize Lateness
4 Weighted Completion Time Scheduling
5 Offline Caching
   • Heap: Concrete Data Structure for Priority Queue
6 Data Compression and Huffman Code
7 Summary
Interval Scheduling

**Input:** $n$ jobs, job $i$ with start time $s_i$ and finish time $f_i$

$i$ and $j$ are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint

**Output:** A maximum-size subset of mutually compatible jobs
Interval Scheduling

**Input:** \( n \) jobs, job \( i \) with start time \( s_i \) and finish time \( f_i \)

\( i \) and \( j \) are compatible if \([s_i, f_i)\) and \([s_j, f_j)\) are disjoint

**Output:** A maximum-size subset of mutually compatible jobs
Greedy Algorithm for Interval Scheduling

Which of the following strategies are safe?
Which of the following strategies are safe?

Schedule the job with the smallest size?
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
Greedy Algorithm for Interval Scheduling

Which of the following strategies are safe?

- Schedule the job with the smallest size? No!

![Diagram showing intervals on a timeline]
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
- Schedule the job conflicting with smallest number of other jobs?
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
- Schedule the job conflicting with smallest number of other jobs? No!
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
- Schedule the job conflicting with smallest number of other jobs? No!
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
- Schedule the job conflicting with smallest number of other jobs? No!
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
- Schedule the job conflicting with smallest number of other jobs? No!
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
- Schedule the job conflicting with smallest number of other jobs? No!
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
  - Schedule the job with the smallest size? No!
  - Schedule the job conflicting with smallest number of other jobs? No!
  - Schedule the job with the earliest finish time? Yes!
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
- Schedule the job conflicting with smallest number of other jobs? No!
- Schedule the job with the earliest finish time?
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
- Schedule the job conflicting with smallest number of other jobs? No!
- Schedule the job with the earliest finish time? Yes!
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
  - Schedule the job with the smallest size? No!
  - Schedule the job conflicting with smallest number of other jobs? No!
  - Schedule the job with the earliest finish time? Yes!
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
  - Schedule the job with the smallest size? No!
  - Schedule the job conflicting with smallest number of other jobs? No!
  - Schedule the job with the earliest finish time? Yes!

![Interval Scheduling Diagram]

---

17/87
Greedy Algorithm for Interval Scheduling

**Lemma** It is safe to schedule the job $j$ with the earliest finish time: There is an optimum solution where the job $j$ with the earliest finish time is scheduled.

**Proof.**
**Lemma**  It is safe to schedule the job $j$ with the earliest finish time: There is an optimum solution where the job $j$ with the earliest finish time is scheduled.

**Proof.**

- Take an arbitrary optimum solution $S$

$S$: 

![Interval Scheduling Diagram]
**Greedy Algorithm for Interval Scheduling**

**Lemma** It is safe to schedule the job $j$ with the earliest finish time: There is an optimum solution where the job $j$ with the earliest finish time is scheduled.

**Proof.**
- Take an arbitrary optimum solution $S$
- If it contains $j$, done
**Lemma**  It is safe to schedule the job $j$ with the earliest finish time: There is an optimum solution where the job $j$ with the earliest finish time is scheduled.

**Proof.**

- Take an arbitrary optimum solution $S$
- If it contains $j$, done

\[
S: \quad \text{jobs}\]
\[
j: \quad \text{job}
\]
**Lemma** It is safe to schedule the job $j$ with the earliest finish time: There is an optimum solution where the job $j$ with the earliest finish time is scheduled.

**Proof.**

- Take an arbitrary optimum solution $S$
- If it contains $j$, done
- Otherwise, replace the first job in $S$ with $j$ to obtain another optimum schedule $S'$.
Lemma It is safe to schedule the job \( j \) with the earliest finish time: There is an optimum solution where the job \( j \) with the earliest finish time is scheduled.

Proof.
- Take an arbitrary optimum solution \( S \)
- If it contains \( j \), done
- Otherwise, replace the first job in \( S \) with \( j \) to obtain another optimum schedule \( S' \).

\[
\begin{align*}
S: & & & & & & & & \\
j: & & & & & & & & \\
S': & & & & & & & & 
\end{align*}
\]
**Lemma**  It is safe to schedule the job $j$ with the earliest finish time: There is an optimum solution where the job $j$ with the earliest finish time is scheduled.

- What is the remaining task after we decided to schedule $j$?
- Is it another instance of interval scheduling problem?
Greedy Algorithm for Interval Scheduling

**Lemma** It is safe to schedule the job $j$ with the earliest finish time: There is an optimum solution where the job $j$ with the earliest finish time is scheduled.

- What is the remaining task after we decided to schedule $j$?
- Is it another instance of interval scheduling problem? Yes!
Lemma  It is safe to schedule the job $j$ with the earliest finish time: There is an optimum solution where the job $j$ with the earliest finish time is scheduled.

- What is the remaining task after we decided to schedule $j$?
- Is it another instance of interval scheduling problem? Yes!
**Greedy Algorithm for Interval Scheduling**

**Lemma** It is safe to schedule the job $j$ with the earliest finish time: There is an optimum solution where the job $j$ with the earliest finish time is scheduled.

- What is the remaining task after we decided to schedule $j$?
- Is it another instance of interval scheduling problem? Yes!
Greedy Algorithm for Interval Scheduling

Schedule($s$, $f$, $n$)

1: $A \leftarrow \{1, 2, \ldots, n\}, S \leftarrow \emptyset$
2: while $A \neq \emptyset$ do
3: \quad $j \leftarrow \arg\min_{j' \in A} f_{j'}$
4: \quad $S \leftarrow S \cup \{j\}; A \leftarrow \{j' \in A : s_{j'} \geq f_j\}$
5: return $S$
Greedy Algorithm for Interval Scheduling

\textbf{Schedule}(s, f, n)

1: \( A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \emptyset \)
2: \textbf{while} \( A \neq \emptyset \) \textbf{do}
3: \( j \leftarrow \text{arg min}_{j' \in A} f_{j'} \)
4: \( S \leftarrow S \cup \{j\}; \ A \leftarrow \{j' \in A : s_{j'} \geq f_j\} \)
5: \textbf{return} \( S \)
Greedy Algorithm for Interval Scheduling

`Schedule(s, f, n)`

1: \( A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \emptyset \)
2: \textbf{while } A \neq \emptyset \textbf{ do}
3: \( j \leftarrow \arg \min_{j' \in A} f_{j'} \)
4: \( S \leftarrow S \cup \{j\}; A \leftarrow \{j' \in A : s_{j'} \geq f_j\} \)
5: \textbf{return } S
Greedy Algorithm for Interval Scheduling

Schedule($s, f, n$)

1: $A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \emptyset$
2: \textbf{while} $A \neq \emptyset$ \textbf{do}
3: \quad $j \leftarrow \arg \min_{j' \in A} f_{j'}$
4: \quad $S \leftarrow S \cup \{j\}; A \leftarrow \{j' \in A : s_{j'} \geq f_j\}$
5: \textbf{return} $S$
Greedy Algorithm for Interval Scheduling

Schedule\((s, f, n)\)

1: \( A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \emptyset \)
2: \textbf{while} \( A \neq \emptyset \) \textbf{do}
3: \( j \leftarrow \arg \min_{j' \in A} f_{j'} \)
4: \( S \leftarrow S \cup \{j\}; \ A \leftarrow \{j' \in A : s_{j'} \geq f_j\} \)
5: \textbf{return} \( S \)
Greedy Algorithm for Interval Scheduling

Schedule($s, f, n$)

1: $A \leftarrow \{1, 2, \cdots, n\}$, $S \leftarrow \emptyset$
2: while $A \neq \emptyset$ do
3: \quad $j \leftarrow \arg\min_{j' \in A} f_{j'}$
4: \quad $S \leftarrow S \cup \{j\}$; $A \leftarrow \{j' \in A : s_{j'} \geq f_j\}$
5: \quad return $S$

Diagram showing intervals being scheduled.
Greedy Algorithm for Interval Scheduling

Schedule($s, f, n$)

1: $A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \emptyset$
2: while $A \neq \emptyset$ do
3: \hspace{1cm} $j \leftarrow \arg \min_{j' \in A} f_{j'}$
4: \hspace{1cm} $S \leftarrow S \cup \{j\}; A \leftarrow \{j' \in A : s_{j'} \geq f_j\}$
5: return $S$

Running time of algorithm?
Greedy Algorithm for Interval Scheduling

**Schedule**\((s, f, n)\)

1. \(A \leftarrow \{1, 2, \cdots, n\}, S \leftarrow \emptyset\)
2. while \(A \neq \emptyset\) do
3. \(j \leftarrow \text{arg min}_{j' \in A} f_{j'}\)
4. \(S \leftarrow S \cup \{j\}; A \leftarrow \{j' \in A : s_{j'} \geq f_j\}\)
5. return \(S\)

Running time of algorithm?

- Naive implementation: \(O(n^2)\) time
Greedy Algorithm for Interval Scheduling

\[\text{Schedule}(s, f, n)\]

1. \( A \leftarrow \{1, 2, \cdots , n\}, S \leftarrow \emptyset \)
2. \textbf{while} \( A \neq \emptyset \) \textbf{do}
3. \( j \leftarrow \arg \min_{j' \in A} f_{j'} \)
4. \( S \leftarrow S \cup \{j\}; \ A \leftarrow \{j' \in A : s_{j'} \geq f_j\} \)
5. \textbf{return} \( S \)

Running time of algorithm?

- Naive implementation: \( O(n^2) \) time
- Clever implementation: \( O(n \lg n) \) time
Clever Implementation of Greedy Algorithm

\textbf{Schedule}(s, f, n)

1: sort jobs according to $f$ values
2: $t \leftarrow 0$, $S \leftarrow \emptyset$
3: \textbf{for} every $j \in [n]$ according to non-decreasing order of $f_j$ \textbf{do}
4: \hspace{1em} \textbf{if} $s_j \geq t$ \textbf{then}
5: \hspace{2em} $S \leftarrow S \cup \{j\}$
6: \hspace{2em} $t \leftarrow f_j$
7: \textbf{return} $S$
Clever Implementation of Greedy Algorithm

Schedule\((s, f, n)\)

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0, \ S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
4: \(\text{if } s_j \geq t \text{ then}\)
5: \(S \leftarrow S \cup \{j\}\)
6: \(t \leftarrow f_j\)
7: return \(S\)
Clever Implementation of Greedy Algorithm

Schedule($s$, $f$, $n$)

1: sort jobs according to $f$ values
2: $t \leftarrow 0$, $S \leftarrow \emptyset$
3: for every $j \in [n]$ according to non-decreasing order of $f_j$ do
4:   if $s_j \geq t$ then
5:     $S \leftarrow S \cup \{j\}$
6:     $t \leftarrow f_j$
7: return $S$
Clever Implementation of Greedy Algorithm

**Schedule**($s, f, n$)

1: sort jobs according to $f$ values  
2: $t \leftarrow 0$, $S \leftarrow \emptyset$  
3: for every $j \in [n]$ according to non-decreasing order of $f_j$ do  
4: if $s_j \geq t$ then  
5: $S \leftarrow S \cup \{j\}$  
6: $t \leftarrow f_j$  
7: return $S$
Clever Implementation of Greedy Algorithm

**Schedule**(s, f, n)

1: sort jobs according to f values
2: \( t \leftarrow 0, S \leftarrow \emptyset \)
3: for every \( j \in [n] \) according to non-decreasing order of \( f_j \) do
4: \[ \text{if } s_j \geq t \text{ then} \]
5: \( S \leftarrow S \cup \{j\} \)
6: \( t \leftarrow f_j \)
7: return \( S \)
Clever Implementation of Greedy Algorithm

**Schedule**(*s, f, n*)

1: sort jobs according to *f* values
2: \( t \leftarrow 0, \ S \leftarrow \emptyset \)
3: **for** every \( j \in [n] \) according to non-decreasing order of \( f_j \) **do**
4: \[ \text{if } s_j \geq t \text{ then} \]
5: \( S \leftarrow S \cup \{j\} \)
6: \( t \leftarrow f_j \)
7: **return** \( S \)
Clever Implementation of Greedy Algorithm

Schedule($s, f, n$)

1: sort jobs according to $f$ values
2: $t \leftarrow 0$, $S \leftarrow \emptyset$
3: for every $j \in [n]$ according to non-decreasing order of $f_j$ do
4: \hspace{1em} if $s_j \geq t$ then
5: \hspace{2em} $S \leftarrow S \cup \{j\}$
6: \hspace{2em} $t \leftarrow f_j$
7: return $S$
Clever Implementation of Greedy Algorithm

Schedule\((s, f, n)\)

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0, S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
4: \hspace{1em} if \(s_j \geq t\) then
5: \hspace{2em} \(S \leftarrow S \cup \{j\}\)
6: \hspace{2em} \(t \leftarrow f_j\)
7: return \(S\)
Clever Implementation of Greedy Algorithm

\textbf{Schedule}(s, f, n)

1: sort jobs according to \( f \) values
2: \( t \leftarrow 0, S \leftarrow \emptyset \)
3: \textbf{for} every \( j \in [n] \) according to non-decreasing order of \( f_j \) \textbf{do}
   4: \quad \textbf{if} \ \( s_j \geq t \) \textbf{then}
   5: \quad \quad S \leftarrow S \cup \{j\}
   6: \quad \quad t \leftarrow f_j
7: \textbf{return} \ S
Clever Implementation of Greedy Algorithm

**Schedule**\((s, f, n)\)

1. sort jobs according to \(f\) values
2. \(t \leftarrow 0, S \leftarrow \emptyset\)
3. for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
   4. if \(s_j \geq t\) then
   5. \(S \leftarrow S \cup \{j\}\)
   6. \(t \leftarrow f_j\)
4. return \(S\)
Clever Implementation of Greedy Algorithm

Schedule\( (s, f, n) \)

1: sort jobs according to \( f \) values
2: \( t \leftarrow 0, S \leftarrow \emptyset \)
3: for every \( j \in [n] \) according to non-decreasing order of \( f_j \) do
4: \hspace{1em} if \( s_j \geq t \) then
5: \hspace{2em} \( S \leftarrow S \cup \{j\} \)
6: \hspace{2em} \( t \leftarrow f_j \)
7: return \( S \)
Clever Implementation of Greedy Algorithm

Schedule($s, f, n$)

1: sort jobs according to $f$ values
2: $t \leftarrow 0, S \leftarrow \emptyset$
3: for every $j \in [n]$ according to non-decreasing order of $f_j$
   do
4:     if $s_j \geq t$ then
5:         $S \leftarrow S \cup \{j\}$
6:         $t \leftarrow f_j$
7: return $S$
Clever Implementation of Greedy Algorithm

**Schedule**(*s, f, n*)

1: sort jobs according to *f* values
2: \( t \leftarrow 0, S \leftarrow \emptyset \)
3: for every \( j \in [n] \) according to non-decreasing order of \( f_j \) do
   4: if \( s_j \geq t \) then
   5: \( S \leftarrow S \cup \{j\} \)
   6: \( t \leftarrow f_j \)
7: return \( S \)
Clever Implementation of Greedy Algorithm

**Schedule**(*s*, *f*, *n*)

1: sort jobs according to *f* values
2: \( t \leftarrow 0, \ S \leftarrow \emptyset \)
3: for every \( j \in [n] \) according to non-decreasing order of \( f_j \) do
   4: \( \text{if } s_j \geq t \text{ then} \)
   5: \( S \leftarrow S \cup \{j\} \)
   6: \( t \leftarrow f_j \)
7: return \( S \)
Clever Implementation of Greedy Algorithm

Schedule\((s, f, n)\)

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0, S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
4: \hspace{1em} if \(s_j \geq t\) then
5: \hspace{2em} \(S \leftarrow S \cup \{j\}\)
6: \hspace{2em} \(t \leftarrow f_j\)
7: return \(S\)
Clever Implementation of Greedy Algorithm

Schedule\((s, f, n)\)

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0, S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
4: \quad if \(s_j \geq t\) then
5: \quad \quad \(S \leftarrow S \cup \{j\}\)
6: \quad \quad \(t \leftarrow f_j\)
7: return \(S\)
Clever Implementation of Greedy Algorithm

Schedule\((s, f, n)\)

1: sort jobs according to \(f\) values
2: \(t \leftarrow 0, S \leftarrow \emptyset\)
3: for every \(j \in [n]\) according to non-decreasing order of \(f_j\) do
   4: if \(s_j \geq t\) then
   5: \(S \leftarrow S \cup \{j\}\)
   6: \(t \leftarrow f_j\)
7: return \(S\)
Clever Implementation of Greedy Algorithm

\[ \text{Schedule}(s, f, n) \]

1: sort jobs according to \( f \) values
2: \( t \leftarrow 0, \ S \leftarrow \emptyset \)
3: for every \( j \in [n] \) according to non-decreasing order of \( f_j \) do
4: \hspace{1em} if \( s_j \geq t \) then
5: \hspace{2.8em} \( S \leftarrow S \cup \{j\} \)
6: \hspace{2.8em} \( t \leftarrow f_j \)
7: return \( S \)
Outline

1. Toy Example: Box Packing
2. Interval Scheduling
3. Scheduling to Minimize Lateness
4. Weighted Completion Time Scheduling
5. Offline Caching
   - Heap: Concrete Data Structure for Priority Queue
6. Data Compression and Huffman Code
7. Summary
**Scheduling to minimize lateness**

**Input:** \( n \) jobs, each job \( j \in [n] \) with a processing time \( p_j \) and deadline \( d_j \)

**Output:** schedule jobs on 1 machine, to minimize the max. lateness

\[ C_j: \text{completion time of } j \quad \text{lateness } l_j := \max\{C_j - d_j, 0\} \]

- Example input:

<table>
<thead>
<tr>
<th>( j )</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_j )</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>( d_j )</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( j )</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_j )</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>( d_j )</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

- solution 1: max lateness = \( \max\{0, 3 - 5, 6 - 7, 8 - 4, 9 - 8\} = 4 \)
- solution 2: max lateness = \( \max\{0, 2 - 4, 5 - 5, 8 - 7, 9 - 8\} = 1 \)
- solution 2 is better
Candidate algorithms

Schedule the jobs in some natural order. Which order should we choose?

A. Ascending order of processing times $p_j$
B. Ascending order of slackness $d_j - p_j$
C. Ascending order of deadline $d_j$. 

Lemma
The ascending order of deadlines $d_j$ (the Earliest Deadline First order or the EDF order) is the optimum schedule.
Candidate algorithms

Schedule the jobs in some natural order. Which order should we choose?

A. Ascending order of processing times $p_j$

B. Ascending order of slackness $d_j - p_j$

C. Ascending order of deadline $d_j$. 

Lemma

The ascending order of deadlines $d_j$ (the Earliest Deadline First order or the EDF order) is the optimum schedule.
Candidate algorithms

Schedule the jobs in some natural order. Which order should we choose?

A. Ascending order of processing times \( p_j \)
B. Ascending order of slackness \( d_j - p_j \)
C. Ascending order of deadline \( d_j \).

Lemma The ascending order of deadlines \( d_j \) (the Earliest Deadline First order or the EDF order) is the optimum schedule.
- maximum lateness $= \max \left\{ 0, \max_{j \in [n]} \{ C_j - d_j \} \right\}$. 
maximum lateness = \max \left\{ 0, \max_{j \in [n]} \{C_j - d_j\} \right\}.

\[
d_j > d_{j'}
\]

Before swapping, the maximum of the two terms strictly decreases.
maximum lateness = \( \max \left\{ 0, \max_{j \in [n]} \{ C_j - d_j \} \right\} \).
Maximum lateness = \( \max \left\{ 0, \max_{j \in [n]} \{ C_j - d_j \} \right\} \).

Before:
\[
\max\{t + p_j - d_j, t + p_j + p_{j'} - d_{j'}\} = t + p_j + p_{j'} - d_{j'}
\]
maximum lateness = \( \max \left\{ 0, \max_{j \in [n]} \{ C_j - d_j \} \right\} \).

\( d_j > d_{j'} \)

before: \( \max \{ t + p_j - d_j, t + p_j + p_{j'} - d_{j'} \} = t + p_j + p_{j'} - d_{j'} \)

after: \( \max \{ t + p_{j'} - d_{j'}, t + p_j + p_{j'} - d_j \} \)
- maximum lateness = \( \max \{ 0, \max_{j \in [n]} \{ C_j - d_j \} \} \).

\[
d_j > d_{j'}
\]

- before: \( \max \{ t + p_j - d_j, t + p_j + p_{j'} - d_{j'} \} = t + p_j + p_{j'} - d_{j'} \)
- after: \( \max \{ t + p_{j'} - d_{j'}, t + p_j + p_{j'} - d_j \} \)
- \( p_{j'} - d_{j'} < p_j + p_{j'} - d_{j'} \) and \( p_j + p_{j'} - d_j < p_j + p_{j'} - d_{j'} \)
- maximum lateness = \( \max \left\{ 0, \max_{j \in [n]} \{ C_j - d_j \} \right\} \).

\[ d_j > d_{j'} \]

\[
\begin{array}{cccccccc}
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\hline
& j & j' & \cdot & \cdot & \cdot & \cdot & \cdot \\
\hline
& j' & j & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array}
\]

- before: \( \max \{ t + p_j - d_j, t + p_j + p_{j'} - d_{j'} \} = t + p_j + p_{j'} - d_{j'} \)
- after: \( \max \{ t + p_{j'} - d_{j'}, t + p_j + p_{j'} - d_j \} \)

\( p_{j'} - d_{j'} < p_j + p_{j'} - d_{j'} \) \text{ and } \( p_j + p_{j'} - d_j < p_j + p_{j'} - d_{j'} \)

\( \max \{ t + p_{j'} - d_{j'}, t + p_j + p_{j'} - d_j \} < t + p_j + p_{j'} - d_{j'} \)
maximum lateness = \( \max \{ 0, \max_{j \in [n]} \{ C_j - d_j \} \} \).

\[
d_j > d_{j'}
\]

- before: \( \max \{ t + p_j - d_j, t + p_j + p_{j'} - d_{j'} \} = t + p_j + p_{j'} - d_{j'} \)
- after: \( \max \{ t + p_{j'} - d_{j'}, t + p_j + p_{j'} - d_j \} \)
- \( p_{j'} - d_{j'} < p_j + p_{j'} - d_{j'} \) and \( p_j + p_{j'} - d_j < p_j + p_{j'} - d_{j'} \)
- \( \max \{ t + p_{j'} - d_{j'}, t + p_j + p_{j'} - d_j \} < t + p_j + p_{j'} - d_{j'} \)
- after swapping, the maximum of the two terms strictly decreases
Repeated Swapping (for Analysis Only)

1: let $S$ be any schedule (i.e, a permutation of $[n]$)
2: while there are two adjacent jobs $j$ and $j'$ in $S$, with $j$ before $j'$ and $d_j > d_{j'}$ do
3: swap $j$ and $j'$ in $S$
Repeated Swapping (for Analysis Only)

1: let $S$ be any schedule (i.e., a permutation of $[n]$)
2: while there are two adjacent jobs $j$ and $j'$ in $S$, with $j$ before $j'$ and $d_j > d_{j'}$ do
3: swap $j$ and $j'$ in $S$

Q: Does the algorithm terminate?
Repeated Swapping (for Analysis Only)

1: let \( S \) be any schedule (i.e, a permutation of \([n]\))
2: while there are two adjacent jobs \( j \) and \( j' \) in \( S \), with \( j \) before \( j' \) and \( d_j > d_{j'} \) do
3: swap \( j \) and \( j' \) in \( S \)

Q: Does the algorithm terminate?

A: Yes. Number of inversions go down!

- \((j, j')\) is an inversion in \( S \) if \( j \) appears before \( j' \) and \( d_j > d_{j'} \).
Repeated Swapping (for Analysis Only)

1: let $S$ be any schedule (i.e, a permutation of $[n]$)
2: while there are two adjacent jobs $j$ and $j'$ in $S$, with $j$ before $j'$ and $d_j > d_{j'}$ do
3: swap $j$ and $j'$ in $S$

Q: Does the algorithm terminate?

A: Yes. Number of inversions go down!

- $(j, j')$ is an inversion in $S$ if $j$ appears before $j'$ and $d_j > d_{j'}$.
- So the algorithm converges to an EDF order.
Repeated Swapping (for Analysis Only)

1: let $S$ be any schedule (i.e, a permutation of $[n]$)
2: while there are two adjacent jobs $j$ and $j'$ in $S$, with $j$ before $j'$ and $d_j > d_{j'}$ do
3: swap $j$ and $j'$ in $S$

Q: Does the algorithm terminate?

A: Yes. Number of inversions go down!

- $(j, j')$ is an inversion in $S$ if $j$ appears before $j'$ and $d_j > d_{j'}$.
- So the algorithm converges to an EDF order.

Q: What if there are multiple EDF orders, i.e., some jobs have the same deadline?
Repeated Swapping (for Analysis Only)

1: let $S$ be any schedule (i.e., a permutation of $[n]$)
2: while there are two adjacent jobs $j$ and $j'$ in $S$, with $j$ before $j'$ and $d_j > d_{j'}$ do
3: swap $j$ and $j'$ in $S$

Q: Does the algorithm terminate?

A: Yes. Number of inversions go down!

- $(j, j')$ is an inversion in $S$ if $j$ appears before $j'$ and $d_j > d_{j'}$.
- So the algorithm converges to an EDF order.

Q: What if there are multiple EDF orders, i.e., some jobs have the same deadline?

A: All EDF orders have the same maximum lateness.
Outline

1. Toy Example: Box Packing
2. Interval Scheduling
3. Scheduling to Minimize Lateness
4. Weighted Completion Time Scheduling
5. Offline Caching
   - Heap: Concrete Data Structure for Priority Queue
6. Data Compression and Huffman Code
7. Summary
Scheduling to Minimize Weighted Completion Time

**Input:** A set of \( n \) jobs \([n] := \{1, 2, 3, \cdots, n\}\) each job \( j \) has a weight \( w_j \) and processing time \( p_j \)

**Output:** an ordering of jobs so as to minimize the total weighted completion time of jobs

\[
\begin{align*}
ap &= 1 & \quad b \quad pc &= 3 \\
w_a &= 2 & \quad w_b &= 5 & \quad w_c &= 7 \\
a & & b & & c \\
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\text{cost} &= 2 \times 1 + 5 \times 3 + 7 \times 6 = 59 \\
\end{align*}
\]

\[
\begin{align*}
bb &= 5 & \quad wc &= 7 & \quad wa &= 2 \\
a & & b & & c \\
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\text{cost} &= 5 \times 2 + 7 \times 5 + 2 \times 6 = 57
\end{align*}
\]
Candidate algorithms

Schedule the jobs in some natural order. Which order should we choose?

A. Ascending order of processing times $p_j$
B. Descending order of slackness $w_j$
C. Ascending order of $p_j - w_j$
D. Ascending order of $p_j / w_j$

Def. The Smith ratio of a job is $w_j / p_j$.

Lemma The descending order of Smith ratios (the Smith rule) is optimum.
Candidate algorithms

Schedule the jobs in some natural order. Which order should we choose?

A. Ascending order of processing times $p_j$

B. Descending order of slackness $w_j$

C. Ascending order of $p_j - w_j$

D. Ascending order of $p_j/w_j$

Def. The Smith ratio of a job is $w_j/p_j$.

Lemma

The descending order of Smith ratios (the Smith rule) is optimum.
Candidate algorithms

Schedule the jobs in some natural order. Which order should we choose?

A. Ascending order of processing times $p_j$
B. Descending order of slackness $w_j$
C. Ascending order of $p_j - w_j$
D. Ascending order of $p_j/w_j$

Def. The Smith ratio of a job is $w_j/p_j$.

Lemma The descending order of Smith ratios (the Smith rule) is optimum.
A schedule $S$, $j$ is right before $j'$. 

Therefore, swapping decrease the weighted completion time if $p_{j'}w_j < p_jw_{j'}$. 

Using the same argument as for the maximum lateness problem: ascending order of $p_j/w_j$ is optimum. Indeed, optimum weighted completion time is 

$$ X_j \in [n] \quad w_j p_j + \sum_{1 \leq j < j' \leq n} \min\{w_{j'} p_{j'}, w_{j'} p_j\}.$$
A schedule $S$, $j$ is right before $j'$. 

**Q:** How does the total weighted completion time change if we swap $j$ and $j'$? 

$(\cdots, j, j', \cdots) \implies (\cdots, j', j, \cdots)$
A schedule $S$, $j$ is right before $j'$.

Q: How does the total weighted completion time change if we swap $j$ and $j'$?

$(\cdots, j, j', \cdots) \implies (\cdots, j', j, \cdots)$

A: $w_{j'}p_j \implies w_j p_{j'}$

Therefore, swapping decrease the weighted completion time if $p_{j'} w_j < p_j w_{j'}$.

Using the same argument as for the maximum lateness problem: ascending order of $p_j/w_j$ is optimum.

Indeed, optimum weighted completion time is

$$X_{j \in [n]} w_j p_j + X_{1 \leq j < j' \leq n} \min\{w_j p_j', w_{j'} p_j\}. $$
A schedule $S$, $j$ is right before $j'$.

**Q:** How does the total weighted completion time change if we swap $j$ and $j'$?

$(\cdots, j, j', \cdots) \implies (\cdots, j', j, \cdots)$

**A:** $w_{j'}p_j \implies w_jp_{j'}$

Therefore, swapping decreases the weighted completion time if

$$\frac{p_{j'}}{w_{j'}} < \frac{p_j}{w_j}.$$
A schedule $S$, $j$ is right before $j'$. 

**Q:** How does the total weighted completion time change if we swap $j$ and $j'$?

$(\cdots, j, j', \cdots) \implies (\cdots, j', j, \cdots)$

**A:** $w_j p_j \implies w_j' p_j'$

Therefore, swapping decreases the weighted completion time if

$$\frac{p_j'}{w_j'} < \frac{p_j}{w_j}.$$ 

Using the same argument as for the maximum lateness problem: ascending order of $p_j/w_j$ is optimum.
A schedule $S$, $j$ is right before $j'$.

Q: How does the total weighted completion time change if we swap $j$ and $j'$?

$$(\cdots, j, j', \cdots) \implies (\cdots, j', j, \cdots)$$

A: $w_jp_j \implies w_j'p_j'$

Therefore, swapping decrease the weighted completion time if $\frac{p_j}{w_j} < \frac{p_j'}{w_j'}$.

Using the same argument as for the maximum lateness problem: ascending order of $p_j/w_j$ is optimum.

Indeed, optimum weighted completion time is

$$\sum_{j \in [n]} w_jp_j + \sum_{1 \leq j < j' \leq n} \min\{w_jp_j', w_j'p_j\}.$$
Outline

1. Toy Example: Box Packing
2. Interval Scheduling
3. Scheduling to Minimize Lateness
4. Weighted Completion Time Scheduling
5. Offline Caching
   - Heap: Concrete Data Structure for Priority Queue
6. Data Compression and Huffman Code
7. Summary
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests

Cache miss happens if requested page not in cache. We need to bring the page into cache, and evict some existing page if necessary. Cache hit happens if requested page already in cache. Goal: minimize the number of cache misses.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests

<table>
<thead>
<tr>
<th>page sequence</th>
<th>cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store \( k \) pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.

```
page sequence: 1 5 4 2 5 3 2 1

cache: [ ] [ ] [ ]
```
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store \( k \) pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache.
  We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache.
  We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.

<table>
<thead>
<tr>
<th>page sequence</th>
<th>cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✗ 1</td>
</tr>
<tr>
<td>5</td>
<td>✗ 1</td>
</tr>
<tr>
<td>4</td>
<td>✗ 1</td>
</tr>
<tr>
<td>2</td>
<td>✗ 1</td>
</tr>
<tr>
<td>5</td>
<td>✗ 1</td>
</tr>
<tr>
<td>3</td>
<td>✗ 1</td>
</tr>
<tr>
<td>2</td>
<td>✓ 1</td>
</tr>
<tr>
<td>1</td>
<td>✓ 1</td>
</tr>
</tbody>
</table>
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.

<table>
<thead>
<tr>
<th>page sequence</th>
<th>cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>×</td>
</tr>
<tr>
<td>5</td>
<td>×</td>
</tr>
<tr>
<td>4</td>
<td>×</td>
</tr>
<tr>
<td>2</td>
<td>×</td>
</tr>
<tr>
<td>5</td>
<td>×</td>
</tr>
<tr>
<td>3</td>
<td>×</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
</tr>
<tr>
<td>1</td>
<td>✓</td>
</tr>
</tbody>
</table>

misses = 6
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.
- Goal: minimize the number of cache misses.

<table>
<thead>
<tr>
<th>page sequence</th>
<th>cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x 1</td>
</tr>
<tr>
<td>5</td>
<td>x 1</td>
</tr>
<tr>
<td>4</td>
<td>x 1</td>
</tr>
<tr>
<td>2</td>
<td>x 1</td>
</tr>
<tr>
<td>5</td>
<td>x 1</td>
</tr>
<tr>
<td>3</td>
<td>x 1</td>
</tr>
<tr>
<td>2</td>
<td>✓ 1</td>
</tr>
<tr>
<td>1</td>
<td>✓ 1</td>
</tr>
</tbody>
</table>

misses = 6
A Better Solution for Example

<table>
<thead>
<tr>
<th>Page Sequence</th>
<th>Cache</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✗ 1</td>
<td>✗ 1</td>
</tr>
<tr>
<td>5</td>
<td>✗ 1 5</td>
<td>✗ 1 5</td>
</tr>
<tr>
<td>4</td>
<td>✗ 1 5 4</td>
<td>✗ 1 5 4</td>
</tr>
<tr>
<td>2</td>
<td>✗ 1 2 4</td>
<td>✗ 1 5 2</td>
</tr>
<tr>
<td>5</td>
<td>✗ 1 2 5</td>
<td>✓ 1 5 2</td>
</tr>
<tr>
<td>3</td>
<td>✗ 1 2 3</td>
<td>✗ 1 3 2</td>
</tr>
<tr>
<td>2</td>
<td>✓ 1 2 3</td>
<td>✓ 1 3 2</td>
</tr>
<tr>
<td>1</td>
<td>✓ 1 2 3</td>
<td>✓ 1 3 2</td>
</tr>
</tbody>
</table>

misses = 6

misses = 5
## Offline Caching Problem

**Input:**
- \( k \): the size of cache
- \( n \): number of pages
- \( \rho_1, \rho_2, \rho_3, \cdots, \rho_T \in [n] \): sequence of requests

**Output:**
- \( i_1, i_2, i_3, \cdots, i_T \in \{\text{hit, empty}\} \cup [n] \): indices of pages to evict ("hit" means evicting no page, "empty" means evicting empty page)

We use \([n]\) for \(\{1, 2, 3, \cdots, n\}\).
## Offline Caching Problem

**Input:**
- \( k \): the size of cache
- \( n \): number of pages
- \( \rho_1, \rho_2, \rho_3, \ldots, \rho_T \in [n] \): sequence of requests

**Output:**
- \( i_1, i_2, i_3, \ldots, i_T \in \{ \text{hit, empty} \} \cup [n] \): indices of pages to evict ("hit" means evicting no page, "empty" means evicting empty page)

- Offline Caching: we know the whole sequence ahead of time.
- Online Caching: we have to make decisions on the fly, before seeing future requests.

We use \([n]\) for \(\{1, 2, 3, \ldots, n\}\).
Offline Caching Problem

**Input:**
- $k$: the size of cache
- $n$: number of pages

We use $[n]$ for $\{1, 2, 3, \ldots, n\}$.

**Output:**
- $\rho_1, \rho_2, \rho_3, \ldots, \rho_T \in [n]$: sequence of requests
- $i_1, i_2, i_3, \ldots, i_T \in \{\text{hit, empty}\} \cup [n]$: indices of pages to evict ("hit" means evicting no page, "empty" means evicting empty page)

- Offline Caching: we know the whole sequence ahead of time.
- Online Caching: we have to make decisions on the fly, before seeing future requests.

Q: Which one is more realistic?
Offline Caching Problem

**Input:**
- $k$: the size of cache
- $n$: number of pages
- $\rho_1, \rho_2, \rho_3, \cdots, \rho_T \in [n]$: sequence of requests

**Output:**
- $i_1, i_2, i_3, \cdots, i_T \in \{\text{hit, empty}\} \cup [n]$: indices of pages to evict ("hit" means evicting no page, "empty" means evicting empty page)

- Offline Caching: we know the whole sequence ahead of time.
- Online Caching: we have to make decisions on the fly, before seeing future requests.

**Q:** Which one is more realistic?

**A:** Online caching
Offline Caching: we know the whole sequence ahead of time.
Online Caching: we have to make decisions on the fly, before seeing future requests.

Q: Which one is more realistic?
A: Online caching

Q: Why do we study the offline caching problem?
Offline Caching: we know the whole sequence ahead of time.

Online Caching: we have to make decisions on the fly, before seeing future requests.

Q: Which one is more realistic?

A: Online caching

Q: Why do we study the offline caching problem?

A: Use the offline solution as a benchmark to measure the “competitive ratio” of online algorithms
Offline Caching: Potential Greedy Algorithms

- FIFO (First-In-First-Out): always evict the first page in cache
Offline Caching: Potential Greedy Algorithms

- **FIFO (First-In-First-Out):** always evict the first page in cache
- **LRU (Least-Recently-Used):** Evict page whose most recent access was earliest

Indeed all the algorithms are “online”, i.e., the decisions can be made without knowing future requests. Online algorithms can not be optimum.
Offline Caching: Potential Greedy Algorithms

- **FIFO (First-In-First-Out):** always evict the first page in cache
- **LRU (Least-Recently-Used):** Evict page whose most recent access was earliest
- **LFU (Least-Frequently-Used):** Evict page that was least frequently requested
Offline Caching: Potential Greedy Algorithms

- **FIFO (First-In-First-Out):** always evict the first page in cache
- **LRU (Least-Recently-Used):** Evict page whose most recent access was earliest
- **LFU (Least-Frequently-Used):** Evict page that was least frequently requested

All the above algorithms are not optimum!

Indeed all the algorithms are “online”, i.e, the decisions can be made without knowing future requests. Online algorithms can not be optimum.
FIFO is not optimum
FIFO is not optimum

requests

| 1 |
| 2 |
| 3 |
| 4 |

FIFO

\[ \text{\texttimes} \]
FIFO is not optimum
FIFO is not optimum

requests

1
2
3
4

FIFO

1
2
3
4

1
2
3
4
FIFO is not optimum

requests

1
2
3
4
1
2
3
4
1
2

FIFO

[Diagram showing FIFO requests]

1
2
3
4
1
2
3
4
1
2
FIFO is not optimum

requests
1
2
3
4
1
2
3
4
FIFO

1
1
2
3
4
1

1
2
3
4
1

1
2
3
4
1
FIFO is not optimum
FIFO is not optimum

requests

1
2
3
4

FIFO

1
2
3

1
2
3
FIFO is not optimum

requests

1
2
3
4

FIFO

1
1
2
3
4

1
1
2
3
4
FIFO is not optimum

requests

FIFO

1

2

3

4

1

2

3

4

1

2

3

4

1

2

3
FIFO is not optimum

<table>
<thead>
<tr>
<th>requests</th>
<th>FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
FIFO is not optimum

<table>
<thead>
<tr>
<th>requests</th>
<th>FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>✗ 1</td>
</tr>
<tr>
<td>3</td>
<td>✗ 1 2</td>
</tr>
<tr>
<td>4</td>
<td>✗ 4 23</td>
</tr>
<tr>
<td>1</td>
<td>✗ 4 13</td>
</tr>
</tbody>
</table>

misses = 5
FIFO is not optimum

<table>
<thead>
<tr>
<th>requests</th>
<th>FIFO</th>
<th>Furthest-in-Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✓ 1</td>
<td>✓ 1</td>
</tr>
<tr>
<td>2</td>
<td>✓ 1 2</td>
<td>✓ 1 2</td>
</tr>
<tr>
<td>3</td>
<td>✓ 1 2 3</td>
<td>✓ 1 2 3</td>
</tr>
<tr>
<td>4</td>
<td>✓ 4 2 3</td>
<td>✓ 1 4 3</td>
</tr>
<tr>
<td>1</td>
<td>✓ 4 1 3</td>
<td>✓ 1 4 3</td>
</tr>
</tbody>
</table>

misses = 5

misses = 4
Optimum Offline Caching

**Furthest-in-Future (FF)**

- Algorithm: every time, evict the page that is not requested until furthest in the future, if we need to evict one.
- The algorithm is *not* an online algorithm, since the decision at a step depends on the request sequence in the future.
## Furthest-in-Future (FF)

<table>
<thead>
<tr>
<th>requests</th>
<th>FIFO</th>
<th>Furthest-in-Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>3</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>4</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>1</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **FIFO**: misses = 5
- **Furthest-in-Future**: misses = 4
Example

requests

1 5 4 2 5 3 2 4 3 1 5 3
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

x  x  x

1  1  1

5  5

4
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

X  X  X

1  1  1

5  5

4
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

\[\times \times \times \times \]

\[
\begin{array}{cccc}
\square & 1 & 1 & 1 & 2 \\
\square & \square & 5 & 5 & 5 \\
\square & \square & \square & 4 & 4 \\
\end{array}
\]
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

x  x  x  x

1  1  1  2

5  5  5

4  4
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

\[\times  \times  \times  \times  \checkmark\]

\[\square  1  1  1  2  2\]

\[\square  \square  5  5  5  5\]

\[\square  \square  \square  4  4  4\]
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

✓  ✓  ✓  ✓  ✓  ✓
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3
Example

requests

1 5 4 2 5 3 2 4 3 1 5 3
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

\[ \begin{bmatrix}
1 & 1 & 1 & 2 & 2 & 2 & 2 \\
5 & 5 & 5 & 5 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 & 4 & 4 & 4 \\
\end{bmatrix} \]
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

\[\begin{array}{cccccccccccc}
& & & & & & & \checkmark & & & & \\
& & & & & \times & & & & \times & & \\
& & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & \\
& & & 5 & 5 & 5 & 5 & 3 & 3 & 3 & \\
& & & & & 4 & 4 & 4 & 4 & 4 & 4 & \\
\end{array}\]
Example

requests

1  5  4  2  5  3  2  4  3  1  5  3

x  x  x  x  ✔  x  ✔  ✔  ✔

1  1  1  2  2  2  2  2  2

5  5  5  5  3  3  3  3

4  4  4  4  4  4  4  4
Example

requests

1 5 4 2 5 3 2 4 1 5 3

[green check marks]

1 1 1 2 2 2 2 2 2

5 5 5 5 3 3 3 3

4 4 4 4 4 4 4 4

[red x marks]
Example

requests

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  3  3  3  3

☐  ☐  ☐  4  4  4  4  4  4  4

41/87
Example

requests

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   3   3   3   3   3   3

☐  ☐  ☐  4   4   4   4   4   4   4   4   4

41/87
Example

requests

\[
\begin{array}{cccccccccccc}
1 & 5 & 4 & 2 & 5 & 3 & 2 & 4 & 3 & 1 & 5 & \textcircled{3} \\
\times & \times & \times & \times & \checkmark & \times & \checkmark & \checkmark & \checkmark & \times & \times & \checkmark \\
\square & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 5 & 5 \\
\square & \square & 5 & 5 & 5 & 5 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
\square & \square & \square & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
\end{array}
\]
Recall: Designing and Analyzing Greedy Algorithms

**Greedy Algorithm**
- Build up the solutions in steps
- At each step, make an irrevocable decision using a “reasonable” strategy

**A Common Way to Analyze Greedy Algorithms**
- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)
Recall: Designing and Analyzing Greedy Algorithms

**Greedy Algorithm**
- Build up the solutions in steps
- At each step, make an *irrevocable* decision using a “reasonable” strategy

**A Common Way to Analyze Greedy Algorithms**
- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)
Offline Caching Problem

**Input:** $k$: the size of cache  
$n$: number of pages  
$\rho_1, \rho_2, \rho_3, \cdots, \rho_T \in [n]$: sequence of requests

**Output:** $i_1, i_2, i_3, \cdots, i_t \in \{\text{hit, empty}\} \cup [n]$  
- empty stands for an empty page  
- “hit” means evicting no pages
Offline Caching Problem

**Input:**
- $k$: the size of cache
- $n$: number of pages
- $\rho_1, \rho_2, \rho_3, \cdots, \rho_T \in [n]$: sequence of requests
- $p_1, p_2, \cdots, p_k \in \{\text{empty}\} \cup [n]$: initial set of pages in cache

**Output:**
- $i_1, i_2, i_3, \cdots, i_t \in \{\text{hit, empty}\} \cup [n]
  - empty stands for an empty page
  - “hit” means evicting no pages
A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)
A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

**Lemma** Assume at time 1 a page fault happens and there are no empty pages in the cache. Let $p^*$ be the page in cache that is not requested until furthest in the future. It is safe to evict $p^*$ at time 1.
A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is "safe" (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

Lemma Assume at time 1 a page fault happens and there are no empty pages in the cache. Let $p^*$ be the page in cache that is not requested until furthest in the future. There is an optimum solution in which $p^*$ is evicted at time 1.
Proof.

1. $S$: any optimum solution
2. $p^*$: page in cache not requested until furthest in the future.
   - In the example, $p^* = 3$. 
**Proof.**

1. \( S \): any optimum solution
2. \( p^* \): page in cache not requested until furthest in the future.
   - In the example, \( p^* = 3 \).
3. Assume \( S \) evicts some \( p' \neq p^* \) at time 1; otherwise done.
   - In the example, \( p' = 2 \).
Proof.

1. $S$: any optimum solution
2. $p^*$: page in cache not requested until furthest in the future.
   - In the example, $p^* = 3$.
3. Assume $S$ evicts some $p' \neq p^*$ at time 1; otherwise done.
   - In the example, $p' = 2$. 
Proof.

Let $S$ be a page set and consider the page replacement algorithm with a prespecified order. Initially, $S$ contains pages $1, 2, 3, 4$. At time $1$, page $1$ is accessed. The page replacement algorithm replaces page $p^*$ (=3) with page $p^\prime$ (=2) instead of page $p^\prime$ (=2) at time 1.

After time 1, the cache status of $S$ and that of $S'$ differ by only 1 page. $S'$ contains page $p^\prime$ (=2) and $S$ contains page $p^*$ (=3).

From now on, $S'$ will "copy" $S$. 

\[ S : \]
\[
\begin{array}{ccc}
1 & 1 \\
2 & 4 \\
3 & 3 \\
\end{array}
\]
**Proof.**

Create $S'$. $S'$ evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.
Proof.

Create $S'$. $S'$ evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.
Proof.

4 Create $S'$. $S'$ evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.

5 After time 1, cache status of $S$ and that of $S'$ differ by only 1 page. $S'$ contains $p'(=2)$ and $S$ contains $p^*(=3)$. 
Proof.

4 Create $S'$. $S'$ evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.

5 After time 1, cache status of $S$ and that of $S'$ differ by only 1 page. $S'$ contains $p'(=2)$ and $S$ contains $p^*(=3)$.

6 From now on, $S'$ will “copy” $S$. 
Proof.

4 Create $S'$. $S'$ evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.

5 After time 1, cache status of $S$ and that of $S'$ differ by only 1 page. $S'$ contains $p'(=2)$ and $S$ contains $p^*(=3)$.

6 From now on, $S'$ will “copy” $S$. 
Proof.


5. After time 1, cache status of $S$ and that of $S'$ differ by only 1 page. $S'$ contains $p'(=2)$ and $S$ contains $p^*(=3)$.

6. From now on, $S'$ will “copy” $S$. 
### Proof.


5. After time 1, cache status of $S$ and that of $S'$ differ by only 1 page. $S'$ contains $p'(=2)$ and $S$ contains $p^*(=3)$.

6. From now on, $S'$ will "copy" $S$. 

---

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

$S$:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

$S'$:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
Proof.


5. After time 1, cache status of $S$ and that of $S'$ differ by only 1 page. $S'$ contains $p'(=2)$ and $S$ contains $p^*(=3)$.

6. From now on, $S'$ will “copy” $S$. 
Proof.


5. After time 1, cache status of $S$ and that of $S'$ differ by only 1 page. $S'$ contains $p'(=2)$ and $S$ contains $p^*(=3)$.

6. From now on, $S'$ will “copy” $S$. 
Proof.

4 Create $S'$. $S'$ evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.

5 After time 1, cache status of $S$ and that of $S'$ differ by only 1 page. $S'$ contains $p'(=2)$ and $S$ contains $p^*(=3)$.

6 From now on, $S'$ will “copy” $S$. 
Proof.

4 Create $S'$. $S'$ evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.

5 After time 1, cache status of $S$ and that of $S'$ differ by only 1 page. $S'$ contains $p'(=2)$ and $S$ contains $p^*(=3)$.

6 From now on, $S'$ will “copy” $S$. 
Proof.


5. After time 1, cache status of $S$ and that of $S'$ differ by only 1 page. $S'$ contains $p'(=2)$ and $S$ contains $p^*(=3)$.

6. From now on, $S'$ will “copy” $S$. 
Proof.

4 Create $S'$. $S'$ evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.

5 After time 1, cache status of $S$ and that of $S'$ differ by only 1 page. $S'$ contains $p'(=2)$ and $S$ contains $p^*(=3)$.

6 From now on, $S'$ will “copy” $S$. 
Proof.

If \( S \) evicted the page \( p^* \), then \( S' \) will evict the page \( p' \). Then, the cache status of \( S \) and that of \( S' \) will be the same. \( S \) and \( S' \) will be exactly the same from now on.

Assume \( S \) did not evict \( p^* \) (=3) before we see \( p' \) (=2).
Proof.

If $S$ evicted the page $p^*$, $S'$ will evict the page $p'$. Then, the cache status of $S$ and that of $S'$ will be the same. $S$ and $S'$ will be exactly the same from now on.
Proof.

7 If $S$ evicted the page $p^*$, $S'$ will evict the page $p'$. Then, the cache status of $S$ and that of $S'$ will be the same. $S$ and $S'$ will be exactly the same from now on.

8 Assume $S$ did not evict $p^*(=3)$ before we see $p'(=2)$.
Proof.

7 If $S$ evicted the page $p^*$, $S'$ will evict the page $p'$. Then, the cache status of $S$ and that of $S'$ will be the same. $S$ and $S'$ will be exactly the same from now on.

8 Assume $S$ did not evict $p^*(=3)$ before we see $p'(=2)$. 
Proof.

If $S$ evicts $p^* (=3)$ for $p' (=2)$, then $S$ won't be optimum. Assume otherwise. So far, $S'$ has 1 less page-miss than $S$ does. The status of $S'$ and that of $S$ only differ by 1 page.
Proof.

If $S$ evicts $p^\ast(=3)$ for $p'(=2)$, then $S$ won’t be optimum. Assume otherwise.

So far, $S'$ has 1 less page-miss than $S$ does.

The status of $S'$ and that of $S$ only differ by 1 page.
Proof.

If \( S \) evicts \( p^* (=3) \) for \( p' (=2) \), then \( S \) won't be optimum. Assume otherwise. So far, \( S' \) has 1 less page-miss than \( S \) does. The status of \( S' \) and that of \( S \) only differ by 1 page.
Proof.

If $S$ evicts $p^*(=3)$ for $p'(=2)$, then $S$ won’t be optimum. Assume otherwise.
**Proof.**

If $S$ evicts $p^*(=3)$ for $p'(=2)$, then $S$ won’t be optimum. Assume otherwise.
Proof.

If $S$ evicts $p^*(=3)$ for $p'(=2)$, then $S$ won’t be optimum. Assume otherwise.
Proof.

If \( S \) evicts \( p^*(=3) \) for \( p'(=2) \), then \( S \) won’t be optimum. Assume otherwise.
Proof.

9 If $S$ evicts $p^*(=3)$ for $p' (=2)$, then $S$ won’t be optimum. Assume otherwise.

10 So far, $S'$ has 1 less page-miss than $S$ does.
Proof.

9 If \( S \) evicts \( p^*(=3) \) for \( p'(=2) \), then \( S \) won’t be optimum. Assume otherwise.

10 So far, \( S' \) has 1 less page-miss than \( S \) does.

11 The status of \( S' \) and that of \( S \) only differ by 1 page.
\begin{proof}

We can then guarantee that $S'$ make at most the same number of page-misses as $S$ does.

Idea: if $S$ has a page-hit and $S'$ has a page-miss, we use the opportunity to make the status of $S'$ the same as that of $S$.

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
4 & 5 & 4 & \cdots & 2 & 3 \\
\hline
\hline
1 & 1 & 5 & 5 & \cdots & 6 & 2 \\
\hline
2 & 4 & 4 & 4 & 8 & 8 \\
\hline
3 & 3 & 3 & 3 & 3 & 3 \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
1 & 1 & 5 & 5 & \cdots & 6 & 6 \\
\hline
\hline
2 & 4 & 4 & 4 & 8 & 8 \\
\hline
3 & 2 & 2 & 2 & 2 & 2 \\
\hline
\end{tabular}
\end{center}

\end{proof}
Proof.

12 We can then guarantee that $S'$ make at most the same number of page-misses as $S$ does.
Proof.

We can then guarantee that $S'$ make at most the same number of page-misses as $S$ does.

- Idea: if $S$ has a page-hit and $S'$ has a page-miss, we use the opportunity to make the status of $S'$ the same as that of $S$. 

□
Thus, we have shown how to create another solution $S'$ with the same number of page-misses as that of the optimum solution $S$. Thus, we proved

**Lemma** Assume at time 1 a page fault happens and there are no empty pages in the cache. Let $p^*$ be the page in cache that is not requested until furthest in the future. There is an optimum solution in which $p^*$ is evicted at time 1.
Thus, we have shown how to create another solution $S'$ with the same number of page-misses as that of the optimum solution $S$. Thus, we proved

**Lemma**  Assume at time 1 a page fault happens and there are no empty pages in the cache. Let $p^*$ be the page in cache that is not requested until furthest in the future. It is safe to evict $p^*$ at time 1.
Thus, we have shown how to create another solution $S'$ with the same number of page-misses as that of the optimum solution $S$. Thus, we proved

**Lemma** Assume at time 1 a page fault happens and there are no empty pages in the cache. Let $p^*$ be the page in cache that is not requested until furthest in the future. It is safe to evict $p^*$ at time 1.

**Theorem** The furthest-in-future strategy is optimum.
1: \textbf{for } $t \leftarrow 1$ to $T$ \textbf{do} \\
2: \textbf{if } $\rho_t$ is in cache \textbf{then} do nothing \\
3: \textbf{else if } there is an empty page in cache \textbf{then} \\
4: \hspace{1em} evict the empty page and load $\rho_t$ in cache \\
5: \textbf{else} \\
6: \hspace{1em} $p^*$ $\leftarrow$ page in cache that is not used furthest in the future \\
7: \hspace{1em} evict $p^*$ and load $\rho_t$ in cache
Q: How can we make the algorithm as fast as possible?

A: The running time can be made to be $O(n + T \log k)$. For each page, use a linked list (or an array with dynamic size) to store the time steps in which the page is requested. We can find the next time a page is requested easily. Use a priority queue data structure to hold all the pages in cache, so that we can easily find the page that is requested furthest in the future.
Q: How can we make the algorithm as fast as possible?

A: The running time can be made to be $O(n + T \log k)$. For each page $p$, use a linked list (or an array with dynamic size) to store the time steps in which $p$ is requested. We can find the next time a page is requested easily. Use a priority queue data structure to hold all the pages in cache, so that we can easily find the page that is requested furthest in the future.
Q: How can we make the algorithm as fast as possible?

A:

- The running time can be made to be $O(n + T \log k)$.
- For each page $p$, use a linked list (or an array with dynamic size) to store the time steps in which $p$ is requested.
Q: How can we make the algorithm as fast as possible?

A:

- The running time can be made to be $O(n + T \log k)$.
- For each page $p$, use a linked list (or an array with dynamic size) to store the time steps in which $p$ is requested.
- We can find the next time a page is requested easily.
Q: How can we make the algorithm as fast as possible?

A:
- The running time can be made to be $O(n + T \log k)$.
- For each page $p$, use a linked list (or an array with dynamic size) to store the time steps in which $p$ is requested.
- We can find the next time a page is requested easily.
- Use a priority queue data structure to hold all the pages in cache, so that we can easily find the page that is requested furthest in the future.
The image shows a table with time and pages columns, and an adjacent priority queue with pages and priority values. The table is structured as follows:

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages</td>
<td>P1</td>
<td>P5</td>
<td>P4</td>
<td>P2</td>
<td>P5</td>
<td>P3</td>
<td>P2</td>
<td>P4</td>
<td>P3</td>
<td>P1</td>
<td>P5</td>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

The priority queue is organized as follows:

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From the table:
- At time 0, pages are: P1, P5, P4, P2, P5, P3, P2, P4, P3, P1, P5, P3.
- At time 1, lines 1 and 2 are: P1: 1 10, P2: 4 7.
- At time 2, lines 3 and 4 are: P3: 6 9 12, P4: 3 8.
- At time 3, lines 5 and 6 are: P5: 2 5 11.
<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages</td>
<td>P1</td>
<td>P5</td>
<td>P4</td>
<td>P2</td>
<td>P5</td>
<td>P3</td>
<td>P2</td>
<td>P4</td>
<td>P3</td>
<td>P1</td>
<td>P5</td>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

**P1:**

1 10

**P2:**

4 7

**P3:**

6 9 12

**P4:**

3 8

**P5:**

2 5 11

**Priority Queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

53/87
The diagram shows a sequence of pages accessed over time, with a priority queue indicating page priorities.

**Pages Accessed Over Time:**

<table>
<thead>
<tr>
<th>Time</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P1</td>
</tr>
<tr>
<td>1</td>
<td>P5</td>
</tr>
<tr>
<td>2</td>
<td>P4</td>
</tr>
<tr>
<td>3</td>
<td>P2</td>
</tr>
<tr>
<td>4</td>
<td>P5</td>
</tr>
<tr>
<td>5</td>
<td>P3</td>
</tr>
<tr>
<td>6</td>
<td>P2</td>
</tr>
<tr>
<td>7</td>
<td>P4</td>
</tr>
<tr>
<td>8</td>
<td>P3</td>
</tr>
<tr>
<td>9</td>
<td>P1</td>
</tr>
<tr>
<td>10</td>
<td>P5</td>
</tr>
<tr>
<td>11</td>
<td>P3</td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

**Priority Queue:**

- **Pages:** P1, P2, P3, P4, P5
- **Priority Values:**
  - P1: 10
  - P2: 7
  - P3: 12
  - P4: 8
  - P5: 11

**Diagram Details:**

- The time axis ranges from 0 to 12.
- Pages P1, P2, P3, P4, and P5 are accessed at specific times, indicated by the corresponding rows in the time table.
- The priority queue lists the pages with their respective priority values.
- The page access times and priorities are marked in a grid-like format.
<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages</td>
<td>P1</td>
<td>P5</td>
<td>P4</td>
<td>P2</td>
<td>P5</td>
<td>P3</td>
<td>P2</td>
<td>P4</td>
<td>P3</td>
<td>P1</td>
<td>P5</td>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

**Priority Queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
</tr>
<tr>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>0</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>pages</td>
<td>P1</td>
</tr>
</tbody>
</table>

**Priority Queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Pages**

<table>
<thead>
<tr>
<th>P1:</th>
<th>1</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2:</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>P3:</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>P4:</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>P5:</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>
### Priority Queue

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>pages</th>
<th>priority queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>P5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>P4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>P5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>P4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>P5</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **P1:** [1, 10]
- **P2:** [4, 7]
- **P3:** [6, 9, 12]
- **P4:** [3, 8]
- **P5:** [2, 5, 11]
<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages</td>
<td>P1</td>
<td>P5</td>
<td>P4</td>
<td>P2</td>
<td>P5</td>
<td>P3</td>
<td>P2</td>
<td>P4</td>
<td>P3</td>
<td>P1</td>
<td>P5</td>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

**Priority Queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
</tr>
<tr>
<td>P4</td>
<td>8</td>
</tr>
</tbody>
</table>

**Pages:**

- **P1:** 1, 10
- **P2:** 4, 7
- **P3:** 6, 9, 12
- **P4:** 3, 8
- **P5:** 2, 5, 11
The diagram illustrates a time-line with corresponding pages and priority queue values for five processes (P1, P2, P3, P4, P5). The time is represented along the x-axis, and the pages are listed beneath it. Each process has a priority queue element: P1 with values 1, 10; P2 with values 4, 7; P3 with values 6, 9, 12; P4 with values 3, 8; P5 with values 2, 5, 11. The priority queue values for these elements are shown in a table format: P1 has a priority value of 10, P5 has a priority value of 5, and P4 has a priority value of 8.
<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages</td>
<td>P1</td>
<td>P5</td>
<td>P4</td>
<td>P2</td>
<td>P5</td>
<td>P3</td>
<td>P2</td>
<td>P4</td>
<td>P3</td>
<td>P1</td>
<td>P5</td>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

**Priority Queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5</td>
<td>5</td>
</tr>
<tr>
<td>P4</td>
<td>8</td>
</tr>
<tr>
<td>time</td>
<td>0</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>pages</td>
<td>P1</td>
</tr>
</tbody>
</table>

**Priority Queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>7</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
</tr>
<tr>
<td>P4</td>
<td>8</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages</td>
<td>P1</td>
<td>P5</td>
<td>P4</td>
<td>P2</td>
<td>P5</td>
<td>P3</td>
<td>P2</td>
<td>P4</td>
<td>P3</td>
<td>P1</td>
<td>P5</td>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

**Priority Queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>7</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
</tr>
<tr>
<td>P4</td>
<td>8</td>
</tr>
<tr>
<td>time</td>
<td>0</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>pages</td>
<td>P1</td>
</tr>
</tbody>
</table>

P1: \[1\ 10\]

P2: \[4\ 7\]

P3: \[6\ 9\ 12\]

P4: \[3\ 8\]

P5: \[2\ 5\ 11\]

**priority queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>7</td>
</tr>
<tr>
<td>P5</td>
<td>11</td>
</tr>
<tr>
<td>P4</td>
<td>8</td>
</tr>
<tr>
<td>time</td>
<td>0</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>pages</td>
<td>P1</td>
</tr>
</tbody>
</table>

**Priority Queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>7</td>
</tr>
<tr>
<td>P5</td>
<td>11</td>
</tr>
<tr>
<td>P4</td>
<td>8</td>
</tr>
<tr>
<td>time</td>
<td>0</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>pages</td>
<td>P1</td>
</tr>
</tbody>
</table>

P1: \[\begin{bmatrix} 1 & 10 \end{bmatrix}\]

P2: \[\begin{bmatrix} 4 & 7 \end{bmatrix}\]

P3: \[\begin{bmatrix} 6 & 9 & 12 \end{bmatrix}\]

P4: \[\begin{bmatrix} 3 & 8 \end{bmatrix}\]

P5: \[\begin{bmatrix} 2 & 5 & 11 \end{bmatrix}\]

priority queue

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>7</td>
</tr>
<tr>
<td>P4</td>
<td>8</td>
</tr>
<tr>
<td>time</td>
<td>0</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>pages</td>
<td>P1</td>
</tr>
</tbody>
</table>

Priority queue

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>7</td>
</tr>
<tr>
<td>P3</td>
<td>9</td>
</tr>
<tr>
<td>P4</td>
<td>8</td>
</tr>
</tbody>
</table>
The diagram illustrates the operation of a priority queue over a sequence of pages. The table on the right shows the priority values for each page.

**Priority Queue:**

<table>
<thead>
<tr>
<th>Pages</th>
<th>Priority Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>7</td>
</tr>
<tr>
<td>P3</td>
<td>9</td>
</tr>
<tr>
<td>P4</td>
<td>8</td>
</tr>
</tbody>
</table>

**Pages Accessed:**

- Time 0: Pages 1, 10 (P1)
- Time 1: Pages 4, 7 (P2)
- Time 2: Pages 6, 9, 12 (P3)
- Time 3: Pages 3, 8 (P4)
- Time 4: Pages 2, 5, 11 (P5)
The diagram illustrates a priority queue with the following values:

- **P1**: pages 1, 10
- **P2**: pages 4, 7
- **P3**: pages 6, 9, 12
- **P4**: pages 3, 8
- **P5**: pages 2, 5, 11

The priority queue values are as follows:

<table>
<thead>
<tr>
<th>Pages</th>
<th>Priority Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>∞</td>
</tr>
<tr>
<td>P3</td>
<td>9</td>
</tr>
<tr>
<td>P4</td>
<td>8</td>
</tr>
</tbody>
</table>

The time and pages columns show the progression of page allocation over time.
<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages</td>
<td>P1</td>
<td>P5</td>
<td>P4</td>
<td>P2</td>
<td>P5</td>
<td>P3</td>
<td>P2</td>
<td>P4</td>
<td>P3</td>
<td>P1</td>
<td>P5</td>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

- **P1**: 1 [10]
- **P2**: 4 [7]
- **P3**: 6 [9] [12]
- **P4**: 3 [8]
- **P5**: 2 [5] [11]

---

**Priority Queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>∞</td>
</tr>
<tr>
<td>P3</td>
<td>9</td>
</tr>
<tr>
<td>P4</td>
<td>8</td>
</tr>
<tr>
<td>time</td>
<td>0</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>pages</td>
<td>P1</td>
</tr>
</tbody>
</table>

**Priority Queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>∞</td>
</tr>
<tr>
<td>P3</td>
<td>9</td>
</tr>
<tr>
<td>P4</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Pages**

- **P1:** 1 10
- **P2:** 4 7
- **P3:** 6 9 12
- **P4:** 3 8
- **P5:** 2 5 11

The green check marks indicate the pages that have been selected by the priority queue algorithm.
<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages</td>
<td>P1</td>
<td>P5</td>
<td>P4</td>
<td>P2</td>
<td>P5</td>
<td>P3</td>
<td>P2</td>
<td>P4</td>
<td>P3</td>
<td>P1</td>
<td>P5</td>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

**Priority Queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>∞</td>
</tr>
<tr>
<td>P3</td>
<td>9</td>
</tr>
<tr>
<td>P4</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Pages**

- P1: 1 10
- P2: 4 7
- P3: 6 9 12
- P4: 3 8
- P5: 2 5 11
<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages</td>
<td>P1</td>
<td>P5</td>
<td>P4</td>
<td>P2</td>
<td>P5</td>
<td>P3</td>
<td>P2</td>
<td>P4</td>
<td>P3</td>
<td>P1</td>
<td>P5</td>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

- **P1:** 1 10
- **P2:** 4 7
- **P3:** 6 9 12
- **P4:** 3 8
- **P5:** 2 5 11

**Priority Queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>∞</td>
</tr>
<tr>
<td>P3</td>
<td>12</td>
</tr>
<tr>
<td>P4</td>
<td>∞</td>
</tr>
<tr>
<td>Time</td>
<td>0</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>Pages</td>
<td>P1</td>
</tr>
</tbody>
</table>

**Priority Queue**

<table>
<thead>
<tr>
<th>Pages</th>
<th>Priority Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>12</td>
</tr>
<tr>
<td>P4</td>
<td>∞</td>
</tr>
</tbody>
</table>
### Priority Queue

<table>
<thead>
<tr>
<th>Pages</th>
<th>Priority Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>∞</td>
</tr>
<tr>
<td>P3</td>
<td>12</td>
</tr>
<tr>
<td>P4</td>
<td>∞</td>
</tr>
</tbody>
</table>

### Pages Access Over Time

<table>
<thead>
<tr>
<th>Time</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P1, P5</td>
</tr>
<tr>
<td>1</td>
<td>P4, P2</td>
</tr>
<tr>
<td>2</td>
<td>P5, P3</td>
</tr>
<tr>
<td>3</td>
<td>P2, P4</td>
</tr>
<tr>
<td>4</td>
<td>P4, P3</td>
</tr>
<tr>
<td>5</td>
<td>P1, P5</td>
</tr>
<tr>
<td>6</td>
<td>P3</td>
</tr>
<tr>
<td>7</td>
<td>P2</td>
</tr>
<tr>
<td>8</td>
<td>P4</td>
</tr>
<tr>
<td>9</td>
<td>P1, P5</td>
</tr>
<tr>
<td>10</td>
<td>P3</td>
</tr>
<tr>
<td>11</td>
<td>P1</td>
</tr>
<tr>
<td>12</td>
<td>P5</td>
</tr>
<tr>
<td>time</td>
<td>0</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>pages</td>
<td>P1</td>
</tr>
</tbody>
</table>

**Priority Queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>$\infty$</td>
</tr>
<tr>
<td>P3</td>
<td>12</td>
</tr>
<tr>
<td>P4</td>
<td>$\infty$</td>
</tr>
<tr>
<td>time</td>
<td>0</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>pages</td>
<td>P1</td>
</tr>
</tbody>
</table>

### Priority Queue

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>12</td>
</tr>
<tr>
<td>P4</td>
<td>∞</td>
</tr>
<tr>
<td>time</td>
<td>0</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>pages</td>
<td>P1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority queue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pages</td>
<td>priority values</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>pages</td>
<td>P1</td>
<td>P5</td>
<td>P4</td>
<td>P2</td>
<td>P5</td>
<td>P3</td>
<td>P2</td>
<td>P4</td>
<td>P3</td>
<td>P1</td>
<td>P5</td>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

- **P1:** 1 10
- **P2:** 4 7
- **P3:** 6 9 12
- **P4:** 3 8
- **P5:** 2 5 11

**Priority Queue**

<table>
<thead>
<tr>
<th>pages</th>
<th>priority values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5</td>
<td>∞</td>
</tr>
<tr>
<td>P3</td>
<td>12</td>
</tr>
<tr>
<td>P4</td>
<td>∞</td>
</tr>
</tbody>
</table>

- Red X marks pages that have been loaded into the page fault buffer.
- Green check marks pages that have been swapped out of the page fault buffer.
- Black check marks pages that have not been loaded.
- Time is shown in order of execution.
1. **for** every $p \leftarrow 1$ to $n$ **do**
2. $times[p] \leftarrow$ array of times in which $p$ is requested, in increasing order \(\triangleright\) put $\infty$ at the end of array
3. $pointer[p] \leftarrow 1$
4. $Q \leftarrow$ empty priority queue
5. **for** every $t \leftarrow 1$ to $T$ **do**
6. $pointer[\rho_t] \leftarrow pointer[\rho_t] + 1$
7. **if** $\rho_t \in Q$ **then**
8. $Q$.increase-key($\rho_t, times[\rho_t, pointer[\rho_t]]$), **print** “hit”, **continue**
9. **if** $Q$.size() $< k$ **then**
10. **print** “load $\rho_t$ to an empty page”
11. **else**
12. $p \leftarrow Q$.extract-max(), **print** “evict $p$ and load $\rho_t$”
13. $Q$.insert($\rho_t, times[\rho_t, pointer[\rho_t]]$) \(\triangleright\) add $\rho_t$ to $Q$ with key value $times[\rho_t, pointer[\rho_t]]$
Outline

1 Toy Example: Box Packing
2 Interval Scheduling
3 Scheduling to Minimize Lateness
4 Weighted Completion Time Scheduling
5 Offline Caching
   • Heap: Concrete Data Structure for Priority Queue
6 Data Compression and Huffman Code
7 Summary
Let $V$ be a ground set of size $n$.

**Def.** A priority queue is an abstract data structure that maintains a set $U \subseteq V$ of elements, each with an associated key value, and supports the following operations:

- $\text{insert}(v, key\_value)$: insert an element $v \in V \setminus U$, with associated key value $key\_value$.
- $\text{decrease\_key}(v, new\_key\_value)$: decrease the key value of an element $v \in U$ to $new\_key\_value$
- $\text{extract\_min}()$: return and remove the element in $U$ with the smallest key value
- ...
Simple Implementations for Priority Queue

- \( n = \text{size of ground set } V \)

<table>
<thead>
<tr>
<th>data structures</th>
<th>insert</th>
<th>extract_min</th>
<th>decrease_key</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sorted array</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Simple Implementations for Priority Queue

- $n = \text{size of ground set } V$

<table>
<thead>
<tr>
<th>data structures</th>
<th>insert</th>
<th>extract_min</th>
<th>decrease_key</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>sorted array</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Simple Implementations for Priority Queue

- $n = \text{size of ground set } V$

<table>
<thead>
<tr>
<th>data structures</th>
<th>insert</th>
<th>extract_min</th>
<th>decrease_key</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>sorted array</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Simple Implementations for Priority Queue

- \( n = \) size of ground set \( V \)

<table>
<thead>
<tr>
<th>data structures</th>
<th>insert</th>
<th>extract_min</th>
<th>decrease_key</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>sorted array</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>heap</td>
<td>( O(lg \ n) )</td>
<td>( O(lg \ n) )</td>
<td>( O(lg \ n) )</td>
</tr>
</tbody>
</table>
Heap

The elements in a heap is organized using a complete binary tree:

- Nodes are indexed as \( \{1, 2, 3, \ldots, s\} \)
- Parent of node \( i \): \( \lfloor i/2 \rfloor \)
- Left child of node \( i \): \( 2i \)
- Right child of node \( i \): \( 2i + 1 \)
A heap $H$ contains the following fields:

- $s$: size of $U$ (number of elements in the heap)
- $A[i], 1 \leq i \leq s$: the element at node $i$ of the tree
- $p[v], v \in U$: the index of node containing $v$
- $key[v], v \in U$: the key value of element $v$

```
1  
/ 
2  3
/ 
4  5
```

- $s = 5$
- $A = (‘f’, ‘g’, ‘c’, ‘e’, ‘b’)$
- $p[‘f’] = 1, p[‘g’] = 2, p[‘c’] = 3, p[‘e’] = 4, p[‘b’] = 5$
Heap

The following heap property is satisfied:

- for any two nodes \( i, j \) such that \( i \) is the parent of \( j \), we have \( key[A[i]] \leq key[A[j]] \).

A heap. Numbers in the circles denote key values of elements.
\text{insert}(v, \text{key\_value})
\textbf{insert}(v, key\_value)
insert($v, key\_value$)
\textbf{insert}((v, key\_value))
\textbf{insert}(v, \textit{key\_value})
\textbf{insert}(v, key\_value)

1: \( s \leftarrow s + 1 \)
2: \( A[s] \leftarrow v \)
3: \( p[v] \leftarrow s \)
4: \( key[v] \leftarrow key\_value \)
5: \textbf{heapify\_up}(s)

\textbf{heapify\_up}(i)

1: \textbf{while } i > 1 \textbf{ do}
2: \quad j \leftarrow \left\lfloor i/2 \right\rfloor
3: \quad \textbf{if } key[A[i]] < key[A[j]] \textbf{ then}
4: \quad \quad \textbf{swap } A[i] \textbf{ and } A[j]
5: \quad \quad p[A[i]] \leftarrow i, p[A[j]] \leftarrow j
6: \quad \quad i \leftarrow j
7: \quad \textbf{else } \textbf{break}
extract_min()
extract_min()
extract_min()
extract_min()
extract_min()
extract_min()
extract_min()  
1: ret ← A[1]  
3: p[A[1]] ← 1  
4: s ← s − 1  
5: if s ≥ 1 then  
6: heapify_down(1)  
7: return ret

decrease_key(v, key_value)  
1: key[v] ← key_value  
2: heapify-up(p[v])

heapify-down(i)  
1: while 2i ≤ s do  
2: if 2i = s or  
key[A[2i]] ≤ key[A[2i + 1]] then  
3: j ← 2i  
4: else  
5: j ← 2i + 1  
6: if key[A[j]] < key[A[i]] then  
7: swap A[i] and A[j]  
8: p[A[i]] ← i, p[A[j]] ← j  
9: i ← j  
10: else break
Running time of heapify\_up and heapify\_down: $O(lg\ n)$
- Running time of heapify\_up and heapify\_down: $O(\lg n)$
- Running time of insert, exact\_min and decrease\_key: $O(\lg n)$
- Running time of heapify\_up and heapify\_down: $O(\lg n)$
- Running time of insert, exact\_min and decrease\_key: $O(\lg n)$

<table>
<thead>
<tr>
<th>data structures</th>
<th>insert</th>
<th>extract_min</th>
<th>decrease_key</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>sorted array</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>heap</td>
<td>$O(\lg n)$</td>
<td>$O(\lg n)$</td>
<td>$O(\lg n)$</td>
</tr>
</tbody>
</table>
Two Definitions Needed to Prove that the Procedures Maintain Heap Property

**Def.** We say that $H$ is almost a heap except that $key[A[i]]$ is too small if we can increase $key[A[i]]$ to make $H$ a heap.

**Def.** We say that $H$ is almost a heap except that $key[A[i]]$ is too big if we can decrease $key[A[i]]$ to make $H$ a heap.
Outline

1. Toy Example: Box Packing
2. Interval Scheduling
3. Scheduling to Minimize Lateness
4. Weighted Completion Time Scheduling
5. Offline Caching
   - Heap: Concrete Data Structure for Priority Queue
6. Data Compression and Huffman Code
7. Summary
Encoding Letters Using Bits

- 8 letters $a, b, c, d, e, f, g, h$ in a language
- need to encode a message using bits
- idea: use 3 bits per letter

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
<th>$g$</th>
<th>$h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>

decafg $\rightarrow$ 011100000010101110

Q: Can we have a better encoding scheme?

- Seems unlikely: must use 3 bits per letter

Q: What if some letters appear more frequently than the others?
Q: If some letters appear more frequently than the others, can we have a better encoding scheme?

A: Using variable-length encoding scheme might be more efficient.

Idea

- using fewer bits for letters that are more frequently used, and more bits for letters that are less frequently used.
Q: What is the issue with the following encoding scheme?

- $a$: 0
- $b$: 1
- $c$: 00

Solution: Use prefix codes to guarantee a unique decoding.
Q: What is the issue with the following encoding scheme?

- \( a: 0 \)
- \( b: 1 \)
- \( c: 00 \)

A: Can not guarantee a unique decoding. For example, 00 can be decoded to \( aa \) or \( c \).
Q: What is the issue with the following encoding scheme?

- $a: 0$
- $b: 1$
- $c: 00$

A: Can not guarantee a unique decoding. For example, $00$ can be decoded to $aa$ or $c$.

Solution

Use prefix codes to guarantee a unique decoding.
**Def.** A prefix code for a set $S$ of letters is a function $\gamma : S \rightarrow \{0, 1\}^*$ such that for two distinct $x, y \in S$, $\gamma(x)$ is not a prefix of $\gamma(y)$.
**Def.** A prefix code for a set $S$ of letters is a function $\gamma : S \to \{0, 1\}^*$ such that for two distinct $x, y \in S$, $\gamma(x)$ is not a prefix of $\gamma(y)$.

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
<th>$g$</th>
<th>$h$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>100</td>
<td>11</td>
<td>1010</td>
<td>1011</td>
<td>01</td>
</tr>
</tbody>
</table>

![Prefix Codes Diagram](image-url)
Prefix Codes Guarantee Unique Decoding

- Reason: there is only one way to cut the first code.
Prefix Codes Guarantee Unique Decoding

- Reason: there is only one way to cut the first code.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>100</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
<tr>
<td>11</td>
<td>1010</td>
<td>1011</td>
<td>01</td>
</tr>
</tbody>
</table>
Prefix Codes Guarantee Unique Decoding

- Reason: there is only one way to cut the first code.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1010</td>
<td>1011</td>
<td>01</td>
<td></td>
</tr>
</tbody>
</table>

00010011100000001011110100001001
Prefix Codes Guarantee Unique Decoding

- Reason: there is only one way to cut the first code.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>100</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
<tr>
<td>11</td>
<td>1010</td>
<td>1011</td>
<td>01</td>
</tr>
</tbody>
</table>

- 0001/001100000001011110100001001
- c
Prefix Codes Guarantee Unique Decoding

- Reason: there is only one way to cut the first code.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>100</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
<tr>
<td>11</td>
<td>1010</td>
<td>1011</td>
<td>01</td>
</tr>
</tbody>
</table>

- 0001/001/100000001011110100001001
- ca
Prefix Codes Guarantee Unique Decoding

- Reason: there is only one way to cut the first code.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>100</td>
</tr>
<tr>
<td>e</td>
<td>11</td>
<td>1010</td>
<td>1011</td>
<td>01</td>
</tr>
</tbody>
</table>

- 0001/001/\textcolor{red}{100}/000001011110100001001
- cad
Prefix Codes Guarantee Unique Decoding

- Reason: there is only one way to cut the first code.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1010</td>
<td>1011</td>
<td>01</td>
<td></td>
</tr>
</tbody>
</table>

- 0001/001/100/0000/01011110100001001
- cadb
Prefix Codes Guarantee Unique Decoding

- Reason: there is only one way to cut the first code.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1010</td>
<td>1011</td>
<td>01</td>
</tr>
</tbody>
</table>

- 0001/001/100/0000/01/01110100001001
- cadbh
Prefix Codes Guarantee Unique Decoding

- Reason: there is only one way to cut the first code.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1010</td>
<td>1011</td>
<td>01</td>
</tr>
</tbody>
</table>

- 0001/001/100/0000/01/01/1110100001001
- cadbh
Prefix Codes Guarantee Unique Decoding

- Reason: there is only one way to cut the first code.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>100</td>
</tr>
<tr>
<td>e</td>
<td>11</td>
<td>1010</td>
<td>1011</td>
<td>01</td>
</tr>
</tbody>
</table>

- 0001/001/100/0000/01/01/11/10100001001
- cadbhhe
Prefix Codes Guarantee Unique Decoding

- Reason: there is only one way to cut the first code.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>100</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
<tr>
<td>11</td>
<td>1010</td>
<td>1011</td>
<td>01</td>
</tr>
</tbody>
</table>

- 0001/001/100/0000/01/01/11/1010/0001001
- cadbhhef
Prefix Codes Guarantee Unique Decoding

- Reason: there is only one way to cut the first code.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1010</td>
<td>1011</td>
<td>01</td>
</tr>
</tbody>
</table>

- 0001/001/100/0000/01/01/11/1010/0001/001
- cadbhhef

001/001/100/0000/01/01/11/1010/0001/001

Cadbhhef
Prefix Codes Guarantee Unique Decoding

- Reason: there is only one way to cut the first code.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1010</td>
<td>1011</td>
<td>01</td>
<td></td>
</tr>
</tbody>
</table>

- 0001/001/100/0000/01/01/11/1010/0001/001/
- cadbhhefca
Properties of Encoding Tree

- Rooted binary tree
- Left edges labelled 0 and right edges labelled 1
- A leaf corresponds to a code for some letter
- If coding scheme is not wasteful: a non-leaf has exactly two children

Best Prefix Codes

Input: frequencies of letters in a message
Output: prefix coding scheme with the shortest encoding for the message
Properties of Encoding Tree

- Rooted binary tree

Best Prefix Codes

Input: frequencies of letters in a message
Output: prefix coding scheme with the shortest encoding for the message
Properties of Encoding Tree
- Rooted binary tree
- Left edges labelled 0 and right edges labelled 1

Best Prefix Codes
Input: frequencies of letters in a message
Output: prefix coding scheme with the shortest encoding for the message
Properties of Encoding Tree

- Rooted binary tree
- Left edges labelled 0 and right edges labelled 1
- A leaf corresponds to a code for some letter

Best Prefix Codes
Input:
frequencies of letters in a message
Output:
prefix coding scheme with the shortest encoding for the message
Properties of Encoding Tree

- Rooted binary tree
- Left edges labelled 0 and right edges labelled 1
- A leaf corresponds to a code for some letter
- If coding scheme is not wasteful: a non-leaf has exactly two children
Properties of Encoding Tree

- Rooted binary tree
- Left edges labelled 0 and right edges labelled 1
- A leaf corresponds to a code for some letter
- If coding scheme is not wasteful: a non-leaf has exactly two children

Best Prefix Codes

**Input:** frequencies of letters in a message

**Output:** prefix coding scheme with the shortest encoding for the message
<table>
<thead>
<tr>
<th>letters</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequencies</td>
<td>18</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>10</td>
</tr>
</tbody>
</table>

scheme 1 length: 2
scheme 2 length: 3
scheme 3 length: 3

scheme 1: a d e
b c

scheme 2: a
b c d e

scheme 3: a
e
d
b c

scheme 1

scheme 2

scheme 3
# Example

<table>
<thead>
<tr>
<th>letters</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequencies</td>
<td>18</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>10</td>
</tr>
</tbody>
</table>

| scheme 1 length | 2 | 3 | 3 | 2 | 2 | total = 89 |
| scheme 2 length | 1 | 3 | 3 | 3 | 3 | total = 87 |
| scheme 3 length | 1 | 4 | 4 | 3 | 2 | total = 84 |

Scheme 1:
```
a     d     e
b     c
```

Scheme 2:
```
a
b     c
```

Scheme 3:
```
a
d     e
b     c
```
Example Input: \((a: 18, b: 3, c: 4, d: 6, e: 10)\)
Example Input: \((a: 18, b: 3, c: 4, d: 6, e: 10)\)

Q: What types of decisions should we make?
Example Input: \((a: 18, b: 3, c: 4, d: 6, e: 10)\)

Q: What types of decisions should we make?

Can we directly give a code for some letter?
Example Input: \((a: 18, b: 3, c: 4, d: 6, e: 10)\)

Q: What types of decisions should we make?

- Can we directly give a code for some letter?
- Hard to design a strategy; residual problem is complicated.
Example Input: \((a: 18, b: 3, c: 4, d: 6, e: 10)\)

**Q:** What types of decisions should we make?

- Can we directly give a code for some letter?
- Hard to design a strategy; residual problem is complicated.
- Can we partition the letters into left and right sub-trees?
Example Input: \((a: 18, b: 3, c: 4, d: 6, e: 10)\)

**Q:** What types of decisions should we make?

- Can we directly give a code for some letter?
- Hard to design a strategy; residual problem is complicated.
- Can we partition the letters into left and right sub-trees?
- Not clear how to design the greedy algorithm
Example Input: \((a: 18, b: 3, c: 4, d: 6, e: 10)\)

**Q:** What types of decisions should we make?

- Can we directly give a code for some letter?
- Hard to design a strategy; residual problem is complicated.
- Can we partition the letters into left and right sub-trees?
- Not clear how to design the greedy algorithm

**A:** We can choose two letters and make them brothers in the tree.
Which Two Letters Can Be Safely Put Together As Brothers?

- Focus on the “structure” of the optimum encoding tree
Which Two Letters Can Be Safely Put Together As Brothers?

- Focus on the “structure” of the optimum encoding tree
- There are two deepest leaves that are brothers

![Diagram showing two deepest leaves that are brothers]
Which Two Letters Can Be Safely Put Together As Brothers?

- Focus on the “structure” of the optimum encoding tree
- There are two deepest leaves that are brothers

![Diagram showing a triangle with two leaves at the bottom, indicating the two least frequent symbols.]

best to put the two least frequent symbols here!
Which Two Letters Can Be Safely Put Together As Brothers?

- Focus on the “structure” of the optimum encoding tree
- There are two deepest leaves that are brothers

Lemma It is safe to make the two least frequent letters brothers.
Lemma  There is an optimum encoding tree, where the two least frequent letters are brothers.
Lemma There is an optimum encoding tree, where the two least frequent letters are brothers.

- So we can irrevocably decide to make the two least frequent letters brothers.
**Lemma**  There is an optimum encoding tree, where the two least frequent letters are brothers.

- So we can irrevocably decide to make the two least frequent letters brothers.

**Q:** Is the residual problem another instance of the best prefix codes problem?
Lemma There is an optimum encoding tree, where the two least frequent letters are brothers.

- So we can irrevocably decide to make the two least frequent letters brothers.

Q: Is the residual problem another instance of the best prefix codes problem?

A: Yes, though it is not immediate to see why.
- $f_x$: the frequency of the letter $x$ in the support.
- $x_1$ and $x_2$: the two letters we decided to put together.
- $d_x$ the depth of letter $x$ in our output encoding tree.

\[
\sum_{x \in S} f_x d_x = \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2}
\]

\[
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1}
\]
- \( f_x \): the frequency of the letter \( x \) in the support.
- \( x_1 \) and \( x_2 \): the two letters we decided to put together.
- \( d_x \): the depth of letter \( x \) in our output encoding tree.

\[
\sum_{x \in S} f_x d_x
\]

\[
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2}
\]

\[
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1}
\]
- $f_x$: the frequency of the letter $x$ in the support.
- $x_1$ and $x_2$: the two letters we decided to put together.
- $d_x$ the depth of letter $x$ in our output encoding tree.

\[
\sum_{x \in S} f_x d_x
\]

\[
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2}
\]

\[
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1}
\]

Def: $f_{x'} = f_{x_1} + f_{x_2}$
- $f_x$: the frequency of the letter $x$ in the support.
- $x_1$ and $x_2$: the two letters we decided to put together.
- $d_x$ the depth of letter $x$ in our output encoding tree.

\[
\begin{align*}
\sum_{x \in S} f_x d_x &= f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2} \\
&= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1} \\
&= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x'} (d_{x'} + 1)
\end{align*}
\]

Def: $f_{x'} = f_{x_1} + f_{x_2}$
- $f_x$: the frequency of the letter $x$ in the support.
- $x_1$ and $x_2$: the two letters we decided to put together.
- $d_x$ the depth of letter $x$ in our output encoding tree.

**Def:** $f_{x'} = f_{x_1} + f_{x_2}$

\[
\begin{align*}
\sum_{x \in S} f_x d_x & = \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2} \\
& = \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1} \\
& = \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x'} (d_{x'} + 1) \\
& = \sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x + f_{x'}
\end{align*}
\]
- $f_x$: the frequency of the letter $x$ in the support.
- $x_1$ and $x_2$: the two letters we decided to put together.
- $d_x$ the depth of letter $x$ in our output encoding tree.

\[
\sum_{x \in S} f_x d_x = \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2}
\]

\[
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1}
\]

\[
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x'} (d_{x'} + 1)
\]

\[
= \sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x + f_{x'}
\]

Def: $f_{x'} = f_{x_1} + f_{x_2}$
In order to minimize
\[ \sum_{x \in S} f_x d_x, \]
we need to minimize
\[ \sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x, \]
subject to that \( d \) is the depth function for an encoding tree of \( S \setminus \{x_1, x_2\} \).

This is exactly the best prefix codes problem, with letters \( S \setminus \{x_1, x_2\} \cup \{x'\} \) and frequency vector \( f \)!
Example

A 27  B 15  C 11  D 9  E 8  F 5
Example
Example

A 27
B 15
C 11
D 9
E 8
F 5

A B C D E F 589111527
1320
Example

```
A  27
B  15
C  11
D  9
```

```
E  8
F  5
```
Example
Example

```
A 27
  /  
B 15  C 11
     /  
     D 9
        /  
        E 8
           /  
           F 5
```

- Node A with a weight of 27
- Node B with a weight of 15
- Node C with a weight of 11
- Node D with a weight of 9
- Node E with a weight of 8
- Node F with a weight of 5
Example
Example

\[ A : 00 \\
B : 10 \\
C : 010 \\
D : 011 \\
E : 110 \\
F : 111 \]
Def. The codes given the greedy algorithm is called the Huffman codes.
Def. The codes given the greedy algorithm is called the Huffman codes.

Huffman($S, f$)

1: while $|S| > 1$ do
2: let $x_1, x_2$ be the two letters with the smallest $f$ values
3: introduce a new letter $x'$ and let $f_{x'} = f_{x_1} + f_{x_2}$
4: let $x_1$ and $x_2$ be the two children of $x'$
5: $S \leftarrow S \setminus \{x_1, x_2\} \cup \{x'\}$
6: return the tree constructed
Algorithm using Priority Queue

**Huffman**\((S, f)\)

1: \(Q \leftarrow \text{build-priority-queue}(S)\)
2: \textbf{while} \(Q.\text{size} > 1\) \textbf{do}
3: \(x_1 \leftarrow Q.\text{extract-min}()\)
4: \(x_2 \leftarrow Q.\text{extract-min}()\)
5: introduce a new letter \(x'\) and let \(f_{x'} = f_{x_1} + f_{x_2}\)
6: let \(x_1\) and \(x_2\) be the two children of \(x'\)
7: \(Q.\text{insert}(x', f_{x'})\)
8: \textbf{return} the tree constructed
Outline

1. Toy Example: Box Packing
2. Interval Scheduling
3. Scheduling to Minimize Lateness
4. Weighted Completion Time Scheduling
5. Offline Caching
   - Heap: Concrete Data Structure for Priority Queue
6. Data Compression and Huffman Code
7. Summary
# Summary for Greedy Algorithms

## Greedy Algorithm
- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

### Interval scheduling problem
- Schedule the job $j^\ast$ with the earliest deadline

### Offline Caching
- Evict the page that is used furthest in the future

### Huffman codes
- Make the two least frequent letters brothers
Summary for Greedy Algorithms

Greedy Algorithm

- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

- Interval scheduling problem: schedule the job $j^*$ with the earliest deadline
Summary for Greedy Algorithms

**Greedy Algorithm**

- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

- Interval scheduling problem: schedule the job $j^*$ with the earliest deadline
- Offline Caching: evict the page that is used furthest in the future
Summary for Greedy Algorithms

**Greedy Algorithm**

- Build up the solutions in steps
- At each step, make an *irrevocable* decision using a “reasonable” strategy

- Interval scheduling problem: schedule the job $j^*$ with the earliest deadline
- Offline Caching: evict the page that is used furthest in the future
- Huffman codes: make the two least frequent letters brothers
Summary for Greedy Algorithms

A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)
Summary for Greedy Algorithms

A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

Def. A strategy is “safe” if there is always an optimum solution that “agrees with” the decision made according to the strategy.
Proving a Strategy is Safe

- Take an arbitrary optimum solution \( S \)
Proving a Strategy is Safe

- Take an arbitrary optimum solution $S$
- If $S$ agrees with the decision made according to the strategy, done
Proving a Strategy is Safe

- Take an arbitrary optimum solution $S$
- If $S$ agrees with the decision made according to the strategy, done
- So assume $S$ does not agree with decision
Proving a Strategy is Safe

- Take an arbitrary optimum solution \( S \)
- If \( S \) agrees with the decision made according to the strategy, done
- So assume \( S \) does not agree with decision
- Change \( S \) slightly to another optimum solution \( S' \) that agrees with the decision

Interval scheduling problem: exchange \( j^* \) with the first job in an optimal solution

Offline caching: a complicated “copying” algorithm

Huffman codes: move the two least frequent letters to the deepest leaves.
Proving a Strategy is Safe

- Take an arbitrary optimum solution $S$
- If $S$ agrees with the decision made according to the strategy, done
- So assume $S$ does not agree with decision
- Change $S$ slightly to another optimum solution $S'$ that agrees with the decision
- Interval scheduling problem: exchange $j^*$ with the first job in an optimal solution
Proving a Strategy is Safe

- Take an arbitrary optimum solution $S$
- If $S$ agrees with the decision made according to the strategy, done
- So assume $S$ does not agree with decision
- Change $S$ slightly to another optimum solution $S'$ that agrees with the decision
  - Interval scheduling problem: exchange $j^*$ with the first job in an optimal solution
  - Offline caching: a complicated “copying” algorithm
Take an arbitrary optimum solution $S$
If $S$ agrees with the decision made according to the strategy, done
So assume $S$ does not agree with decision
Change $S$ slightly to another optimum solution $S'$ that agrees with the decision
Interval scheduling problem: exchange $j^*$ with the first job in an optimal solution
Offline caching: a complicated “copying” algorithm
Huffman codes: move the two least frequent letters to the deepest leaves.
### Summary for Greedy Algorithms

#### A Common Way to Analyze Greedy Algorithms

1. **Prove that the reasonable strategy is “safe”** (*key*).
2. **Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem** (usually easy).

- Interval scheduling problem: remove $j^*$ and the jobs it conflicts with.
- Offline caching: trivial.
- Huffman codes: merge two letters into one.
- Two problems that do not fall into the category: lateness, weighted completion time.
Summary for Greedy Algorithms

A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

- Interval scheduling problem: remove $j^*$ and the jobs it conflicts with
Summary for Greedy Algorithms

A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

- Interval scheduling problem: remove $j^*$ and the jobs it conflicts with
- Offline caching: trivial
Summary for Greedy Algorithms

**A Common Way to Analyze Greedy Algorithms**

- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

- Interval scheduling problem: remove $j^*$ and the jobs it conflicts with
- Offline caching: trivial
- Huffman codes: merge two letters into one
Summary for Greedy Algorithms

A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

- Interval scheduling problem: remove $j^*$ and the jobs it conflicts with
- Offline caching: trivial
- Huffman codes: merge two letters into one

- Two problems that do not fall into the category: lateness, weighted completion time