CS 577 - More (Hard)/(Interesting) Greedy

Manolis Vlatakis

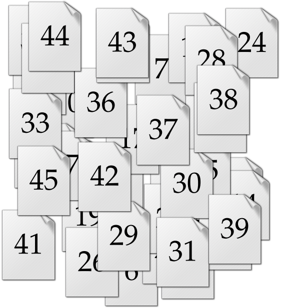
Department of Computer Sciences
University of Wisconsin – Madison

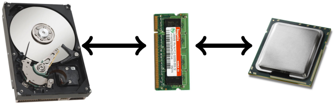
Fall 2024

PAGING

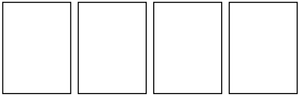
PAGING PROBLEM

$\mathcal{U}$:





Cache:



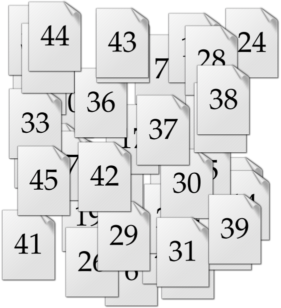
Requests:

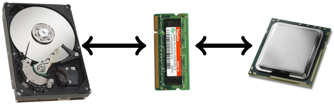
Definition

- $\mathcal{U}$: universe of pages ($|\mathcal{U}| > k$).
- Cache of size k.
- Requests are to the pages of $\mathcal{U}$.
- Goal: Minimize the number of page faults (requests to pages not in the cache).

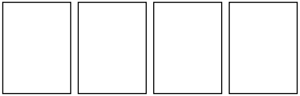
PAGING PROBLEM

$\mathcal{U}$:





Cache:

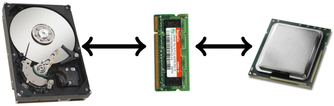


Requests:

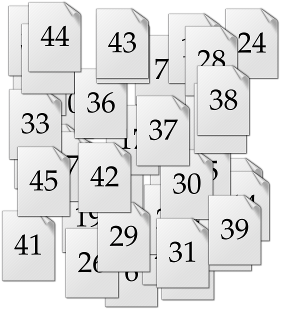
Definition

- $\mathcal{U}$: universe of pages ($|\mathcal{U}| > k$).
- Cache of size k.
- Requests are to the pages of $\mathcal{U}$.
- Goal: Minimize the number of page faults (requests to pages not in the cache).

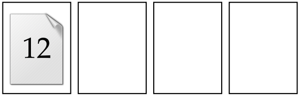
PAGING PROBLEM



$\mathcal{U}$:



Cache:



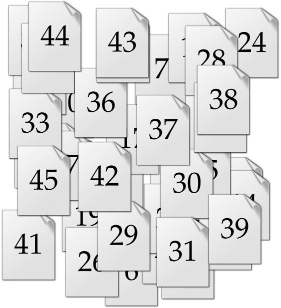
Requests:

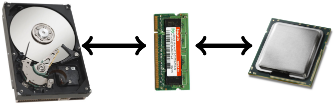
Definition

- $\mathcal{U}$: universe of pages ($|\mathcal{U}| > k$).
- Cache of size k.
- Requests are to the pages of $\mathcal{U}$.
- Goal: Minimize the number of page faults (requests to pages not in the cache).

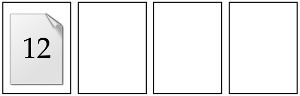
PAGING PROBLEM

$\mathcal{U}$:

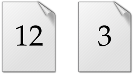




Cache:



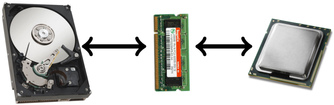
Requests:



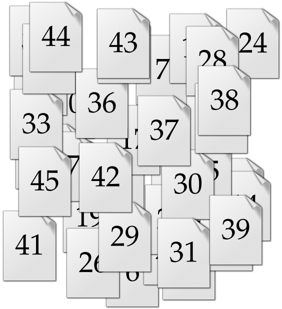
Definition

- $\mathcal{U}$: universe of pages ($|\mathcal{U}| > k$).
- Cache of size k.
- Requests are to the pages of $\mathcal{U}$.
- Goal: Minimize the number of page faults (requests to pages not in the cache).

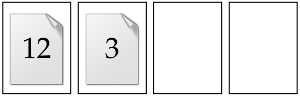
PAGING PROBLEM



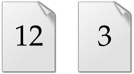
$\mathcal{U}$:



Cache:



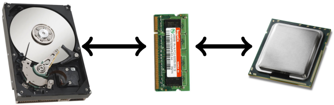
Requests:



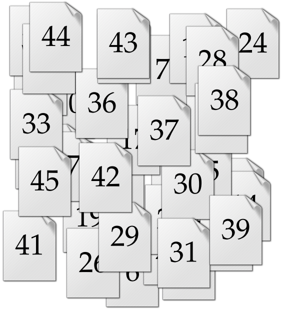
Definition

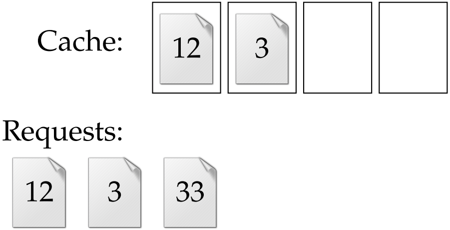
- $\mathcal{U}$: universe of pages ($|\mathcal{U}| > k$).
- Cache of size k.
- Requests are to the pages of $\mathcal{U}$.
- Goal: Minimize the number of page faults (requests to pages not in the cache).

PAGING PROBLEM



$\mathcal{U}$:





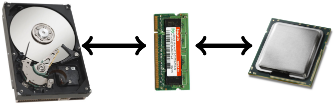
Cache:

Requests:

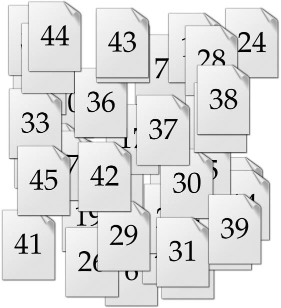
Definition

- $\mathcal{U}$: universe of pages ($|\mathcal{U}| > k$).
- Cache of size k.
- Requests are to the pages of $\mathcal{U}$.
- Goal: Minimize the number of page faults (requests to pages not in the cache).

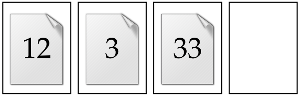
Paging Problem



$\mathcal{U}$:



Cache:



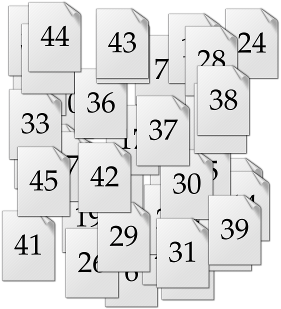
Requests:

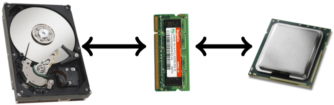
Definition

- $\mathcal{U}$: universe of pages ($|\mathcal{U}| > k$).
- Cache of size k.
- Requests are to the pages of $\mathcal{U}$.
- Goal: Minimize the number of page faults (requests to pages not in the cache).

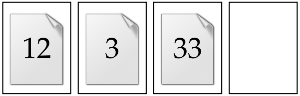
PAGING PROBLEM

$\mathcal{U}$:

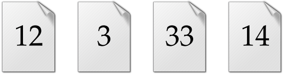




Cache:



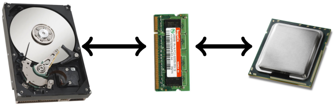
Requests:



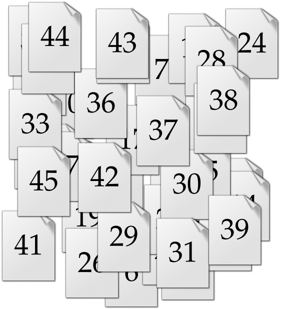
Definition

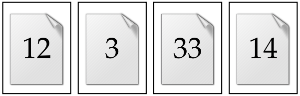
- $\mathcal{U}$: universe of pages ($|\mathcal{U}| > k$).
- Cache of size k.
- Requests are to the pages of $\mathcal{U}$.
- Goal: Minimize the number of page faults (requests to pages not in the cache).

PAGING PROBLEM

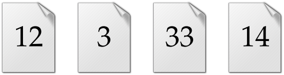


$\mathcal{U}$:





Cache: 12 3 33 14

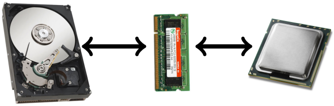


Requests: 12 3 33 14

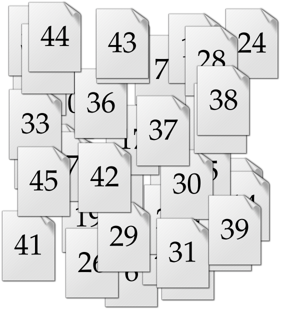
Definition

- $\mathcal{U}$: universe of pages ($|\mathcal{U}| > k$).
- Cache of size k.
- Requests are to the pages of $\mathcal{U}$.
- Goal: Minimize the number of page faults (requests to pages not in the cache).

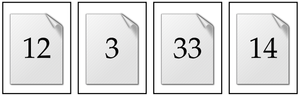
Paging Problem



$\mathcal{U}$:



Cache:



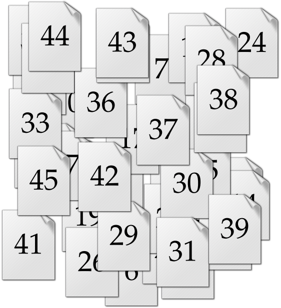
Requests:

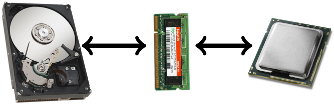
Definition

- $\mathcal{U}$: universe of pages ($|\mathcal{U}| > k$).
- Cache of size k.
- Requests are to the pages of $\mathcal{U}$.
- Goal: Minimize the number of page faults (requests to pages not in the cache).

PAGING PROBLEM

$\mathcal{U}$:





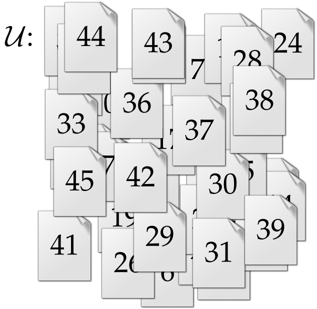
Cache: 12 3 33 14

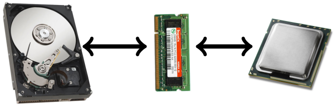
Requests: 12 3 33 14 12 20

Definition

- $\mathcal{U}$: universe of pages ($|\mathcal{U}| > k$).
- Cache of size k.
- Requests are to the pages of $\mathcal{U}$.
- Goal: Minimize the number of page faults (requests to pages not in the cache).

PAGING PROBLEM





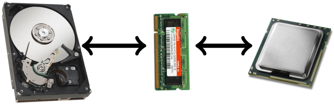
Cache: 12 3 33 14

Requests: 12 3 33 14 12 20

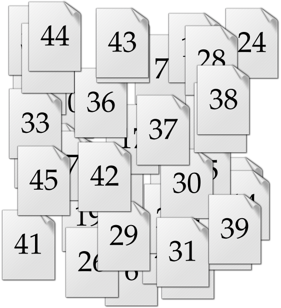
Eviction Strategies

- When designing an algorithm, we are picking an eviction strategy.

PAGING PROBLEM



$\mathcal{U}$:



Cache: 12 3 33 14

Requests: 12 3 33 14 12 20

Eviction Strategies

- When designing an algorithm, we are picking an eviction strategy.
- In the offline version, the algorithm knows the request sequence. What might be a good eviction strategy?

INTUITION

Which item we should evict?

current cache: | a | b | c | d | e | f |

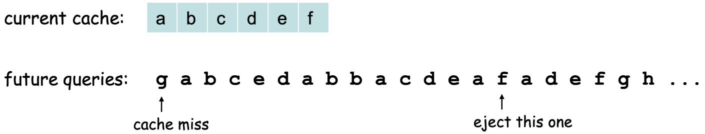
future queries: **g a b c e d a b b a c d e a f a d e f g h ...**

INTUITION

Which item we should evict?

Farthest-in-future

- Evict item in the cache that is not requested until farthest in the future.



current cache: a b c d e f

future queries: g a b c e d a b b a c d e a f a d e f g h . . .

cache miss

eject this one

OFFLINE GREEDY ALGORITHM

Farthest-in-Future (FF)

Evict the page whose next request is the furthest into the future.

OFFLINE GREEDY ALGORITHM

Farthest-in-Future (FF)

Evict the page whose next request is the furthest into the future.

Small Run:

- $\mathcal{U} = \{a, b, c\}$
- $k = 2$
- $\sigma = \langle a, b, c, b, c, a, b \rangle$

OFFLINE GREEDY ALGORITHM

Farthest-in-Future (FF)

Evict the page whose next request is the furthest into the future.

Small Run:

- $\mathcal{U} = \{a, b, c\}$
- $k = 2$
- $\sigma = \langle a, b, c, b, c, a, b \rangle$

🤔How many faults in small run?

OFFLINE GREEDY ALGORITHM

Farthest-in-Future (FF)

Evict the page whose next request is the furthest into the future.

Small Run:

- $\mathcal{U} = \{a, b, c\}$
- $k = 2$
- $\sigma = \langle a, b, c, b, c, a, b \rangle$

🤔Which strategy to prove optimality?

OFFLINE GREEDY ALGORITHM

Farthe

Evict th the future.

Small R

- $\mathcal{U}$ =
- k =
- σ =

🤔Whic

Proving FF Optimality

Exchange Argument

Theorem 1

Proof.

- If on request $j+1$, S behaves as S_{FF}, then define S' as S and the claim follows.

🤔Since S and S_{FF} agree up to now, how different are their caches at this point?

□

PROVING FF OPTIMALITY

EXCHANGE ARGUMENT

Theorem 1

Proof.

- If on request $j+1$, S behaves as S_{FF}, then define S' as S and the claim follows.

🤔Since S and S_{FF} agree up to now, how different are their caches at this point?

Same Initialization ⊕ Same Schedule ≡ Same Cache Content

□

PROVING FF OPTIMALITY

EXCHANGE ARGUMENT

Theorem 1

Let S be a schedule for the n requests that makes the same eviction decisions as S_{FF} for the first j items. Then, there is a schedule S' that makes the same eviction requests as S_{FF} for the first $j+1$ items with no more faults than S.

Proof.

- If on request $j+1$, S behaves as S_{FF}, then define S' as S and the claim follows.

🤔Are there trivial cases where after the $(j+1)$th step, S and S_{FF} will be the same again?

□

Proving FF Optimality

Exchange Argument

Theorem 1

Let S be a schedule for the n requests that makes the same eviction decisions as S_{FF} for the first j items. Then, there is a schedule S' that makes the same eviction requests as S_{FF} for the first $j+1$ items with no more faults than S.

Proof.

- If on request $j+1$, S behaves as S_{FF}, then define S' as S and the claim follows.

🤔Are there trivial cases where after the $(j+1)$th step, S and S_{FF} will be the same again?

Consider the $(j+1)$th request for item $d = d_{j+1}$. Since S and S_{FF} have agreed so far, their cache contents are the same.

- If d is already in the cache for both schedules, no eviction is needed.

PROVING FF OPTIMALITY

EXCHANGE ARGUMENT

Theorem 1

Let S be a schedule for the n requests that makes the same eviction decisions as S_{FF} for the first j items. Then, there is a schedule S' that makes the same eviction requests as S_{FF} for the first $j+1$ items with no more faults than S.

Proof.

- If on request $j+1$, S behaves as S_{FF}, then define S' as S and the claim follows.

🤔Any other trivial case where S and S_{FF} will be the same again?

□

Proving ff Optimality

Exchange Argument

Theorem 1

Let S be a schedule for the n requests that makes the same eviction decisions as S_{FF} for the first j items. Then, there is a schedule S' that makes the same eviction requests as S_{FF} for the first $j+1$ items with no more faults than S.

Proof.

- If on request $j+1$, S behaves as S_{FF}, then define S' as S and the claim follows.

🤔Any other trivial case where S and S_{FF} will be the same again?

If d needs to be brought into the cache, but S and S_{FF} both evict the same item.

□

PROVING FF OPTIMALITY

EXCHANGE ARGUMENT

Theorem 2

Let S be a schedule for the n request that make the same eviction decisions as S_{FF} for the first j items. Then, there is a schedule S' that makes the same eviction requests as S_{FF} for the first $j+1$ items with no more faults than S.

Proof.

- Otherwise, say S evicts f and S_{FF} evicts e. We will build S' by following S_{FF} for the first $j+1$ requests. Note that the number of faults are the same for S and S' up to $j+1$, and the caches match except for f and e.

Step	**Cache S**	**Cache $S' := S_{\text{FF}}$**
j	[same | f | e]	[same | f | e]
$j+1$	[same | d | e]	[same | f | d]

PROVING FF OPTIMALITY

EXCHANGE ARGUMENT

Theorem 2

Let S be a schedule for the n request that make the same eviction decisions as S_{FF} for the first j items. Then, there is a schedule S' that makes the same eviction requests as S_{FF} for the first $j + 1$ items with no more faults than S.

Proof.

- From $j + 2$ onward, S' follows S until they differ again because of some required element x and either:

PROVING FF OPTIMALITY

EXCHANGE ARGUMENT

Theorem 2

Let S be a schedule for the n request that make the same eviction decisions as S_{FF} for the first j items. Then, there is a schedule S' that makes the same eviction requests as S_{FF} for the first $j+1$ items with no more faults than S.

Proof.

- From $j+2$ onward, S' follows S until they differ again because of some required element x and either:
  1. S evicts e. In this case, S' evicts f.

Step	**Cache S**	**Cache $S' := S_{FF}$**
j'	[same | ? | e]	[same | f | ?]
$j'+1$	[same | ? | x]	[same | x | ?]

PROVING FF OPTIMALITY

EXCHANGE ARGUMENT

Theorem 2

Let S be a schedule for the n request that make the same eviction decisions as S_{FF} for the first j items. Then, there is a schedule S' that makes the same eviction requests as S_{FF} for the first $j+1$ items with no more faults than S.

Proof.

- From $j+2$ onward, S' follows S until they differ again because of some required element x and either:
  1. S evicts e. In this case, S' evicts f.
  2. S evicts $g \neq e$ to bring f into the cache. In this case, S' evicts g and brings in e.

Step	**Cache** S	**Cache** $S' := S_{FF}$
j'	[same | g | e]	[same | f | g]
$j'+1$	[same | f | e]	[same | f | e]

PROVING FF OPTIMALITY

EXCHANGE ARGUMENT

Theorem 2

Let S be a schedule for the n request that make the same eviction decisions as S_{FF} for the first j items. Then, there is a schedule S' that makes the same eviction requests as S_{FF} for the first $j+1$ items with no more faults than S.

Proof.

- From $j+2$ onward, S' follows S until they differ again because of some required element x and either:
  1. S evicts e. In this case, S' evicts f.
  2. S evicts $g \neq e$ to bring f into the cache. In this case, S' evicts g and brings in e.
  - Can S bring e into cache earlier than f???

PROVING FF OPTIMALITY

EXCHANGE ARGUMENT

Theorem 2

or

as

Aesop's Moral

- So, we started with a schedule:

$$S = \{\text{evict}_1, \ldots, \text{evict}_j\}[\text{evict}^a_{j+1}] \ldots [\text{evict}^a_{j'}] \ldots$$

$$S_{\text{FF}} = \{\text{evict}_1, \ldots, \text{evict}_j\}[\text{evict}^b_{j+1}] \ldots [\text{evict}^b_{j'}] \ldots$$

- Thus, if instead of some schedule S, we follow S_{FF} for $j+1$ steps, there will come a time in the future when S has a cache miss.
- At that point, either S also incurs a cache miss, or being more efficient, it may make a clever eviction to bring the caches in sync.

cts

No: Since S_{FF} evicts e at $j+1$, f must be requested before e.

- In either case, both S and S' have a page fault, and afterwards their cache match.

PROVING FF OPTIMALITY

EXCHANGE ARGUMENT

Theorem 2

Let S be a schedule for the n request that make the same eviction decisions as S_{FF} for the first j items. Then, there is a schedule S' that makes the same eviction requests as S_{FF} for the first $j+1$ items with no more faults than S.

How do we get optimality of S_{FF} from Theorem 1?

PROVING FF OPTIMALITY

EXCHANGE ARGUMENT

Theorem 2

Let S be a schedule for the n request that make the same eviction decisions as S_{FF} for the first j items. Then, there is a schedule S' that makes the same eviction requests as S_{FF} for the first $j+1$ items with no more faults than S.

How do we get optimality of S_{FF} from Theorem 1?

By induction: We begin with the optimal schedule S^* and inductively apply Theorem 1 for $j = 1, 2, 3, \ldots, n$, which after the n iterations, produces S_{FF}.

PREFIX CODES

BINARY ENCODING

Fixed-Width Encoding

- Set of symbols $S := \{a, b, c, d, e\}$.
- Encoding function $\gamma : S \to \{0,1\}^k$.
  $\gamma(S) := \{000, 001, 010, 011, 100\}$.
- Ex. ASCII

BINARY ENCODING

Fixed-Width Encoding

- Set of symbols $S := \{a, b, c, d, e\}$.
- Encoding function $\gamma : S \to \{0, 1\}^k$.
  $\gamma(S) := \{000, 001, 010, 011, 100\}$.
- Ex. ASCII
- Quiz #1: Decode 000010.

BINARY ENCODING

Fixed-Width Encoding

- Set of symbols $S := \{a, b, c, d, e\}$.
- Encoding function $\gamma : S \to \{0, 1\}^k$.
  $\gamma(S) := \{000, 001, 010, 011, 100\}$.
- Ex. ASCII
- Quiz #1: Decode 000010.

Variable-Width Encoding

- Set of symbols $S := \{a, b, c, d, e\}$.
- Encoding function $\gamma : S \to \{0, 1\}^*$.
  $\gamma(S) := \{0, 1, 10, 01, 11\}$.

BINARY ENCODING

Fixed-Width Encoding

- Set of symbols $S := \{a, b, c, d, e\}$.
- Encoding function $\gamma : S \to \{0, 1\}^k$.
  $\gamma(S) := \{000, 001, 010, 011, 100\}$.
- Ex. ASCII
- Quiz #1: Decode 000010.

Variable-Width Encoding

- Set of symbols $S := \{a, b, c, d, e\}$.
- Encoding function $\gamma : S \to \{0, 1\}^*$.
  $\gamma(S) := \{0, 1, 10, 01, 11\}$.
- Quiz #2: How many ways to decode 0010?

UNIQUE VARIABLE-WIDTH ENCODINGS

Prefix Codes

Encoding of S such that no encoding of a symbol in S is a prefix of another.

- Set of symbols $S := \{a, b, c, d, e\}$.
- Encoding function $\gamma : S \rightarrow \{0, 1\}^*$.
  $\gamma(S) := \{11, 01, 001, 000, 100\}$.

UNIQUE VARIABLE-WIDTH ENCODINGS

Prefix Codes

Encoding of S such that no encoding of a symbol in S is a prefix of another.

- Set of symbols $S := \{a, b, c, d, e\}$.
- Encoding function $\gamma : S \to \{0, 1\}^*$.
  $\gamma(S) := \{11, 01, 001, 000, 100\}$.
- 0010 invalid sequence

UNIQUE VARIABLE-WIDTH ENCODINGS

Prefix Codes

Encoding of S such that no encoding of a symbol in S is a prefix of another.

- Set of symbols $S := \{a, b, c, d, e\}$.
- Encoding function $\gamma : S \to \{0, 1\}^*$.
  $\gamma(S) := \{11, 01, 001, 000, 100\}$.
- 0010 invalid sequence
- Quiz #3: Decode 1101.

UNIQUE VARIABLE-WIDTH ENCODINGS

Prefix Codes

Encoding of S such that no encoding of a symbol in S is a prefix of another.

- Set of symbols $S := \{a, b, c, d, e\}$.
- Encoding function $\gamma : S \to \{0,1\}^*$.
  $\gamma(S) := \{11, 01, 001, 000, 100\}$.

Easy Decoding

Scan left to right, once an encoding is matched, output symbol.

UNIQUE VARIABLE-WIDTH ENCODINGS

Prefix Codes

Encoding of S such that no encoding of a symbol in S is a prefix of another.

- Set of symbols $S := \{a, b, c, d, e\}$.
- Encoding function $\gamma : S \to \{0,1\}^*$.
  $\gamma(S) := \{11, 01, 001, 000, 100\}$.

Easy Decoding

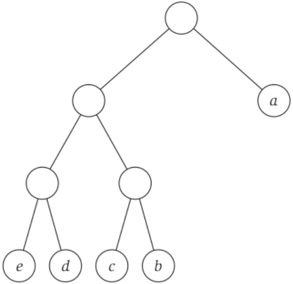
Scan left to right, once an encoding is matched, output symbol.

Optimal Prefix Codes

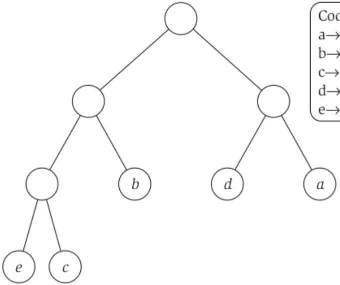
- For a set of symbols S, let f_x denote the frequency of x in the text to be encoded.
- Average bits $\textsc{abl}(\gamma) := \sum_{x \in S} f_x \cdot |\gamma(x)|$.
- Goal: Find γ that minimizes $\textsc{abl}$.

Algorithm Design

Prefix Binary Trees



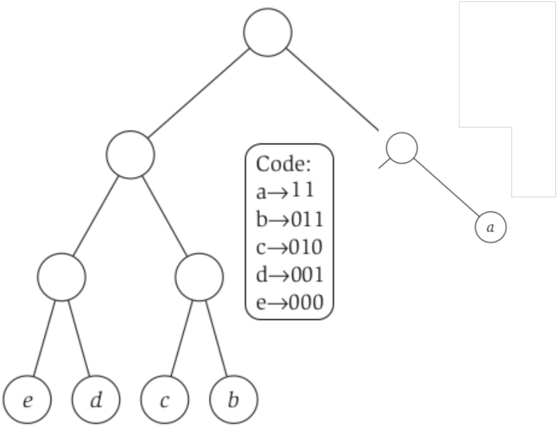
Code:
a→1
b→011
c→010
d→001
e→000



Code:
a→11
b→01
c→001
d→10
e→000

OPTIMAL PREFIX TREE SHOULD BE FULL

IMAGE



Code:
a→11
b→011
c→010
d→001
e→000

Could be that optimal???

OPTIMAL PREFIX TREE IS FULL

PROOF

Theorem 3

The binary tree corresponding to the optimal prefix code is full.

OPTIMAL PREFIX TREE IS FULL

PROOF

Theorem 3

The binary tree corresponding to the optimal prefix code is full.

Proof.

OPTIMAL PREFIX TREE IS FULL

PROOF

Theorem 3

The binary tree corresponding to the optimal prefix code is full.

Proof.

By exchange argument:

- Let T be an optimal prefix tree with a node u with one child v.

OPTIMAL PREFIX TREE IS FULL

PROOF

Theorem 3

The binary tree corresponding to the optimal prefix code is full.

Proof.

By exchange argument:

- Let T be an optimal prefix tree with a node u with one child v.
- Let T' be T with u replaced with v.

OPTIMAL PREFIX TREE IS FULL

PROOF

Theorem 3

The binary tree corresponding to the optimal prefix code is full.

Proof.

By exchange argument:

- Let T be an optimal prefix tree with a node u with one child v.
- Let T' be T with u replaced with v.
- Distance to v decreases by 1 in T', a contradiction.

□

TOP-DOWN APPROACH

Algorithm

- Split S into two sets such that the sets frequency are 1/2 the total frequency.
- Recurse on new sets until singletons.

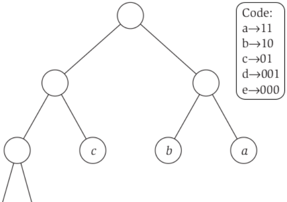
TOP-DOWN APPROACH

Algorithm

- Split S into two sets such that the sets frequency are 1/2 the total frequency.
- Recurse on new sets until singletons.

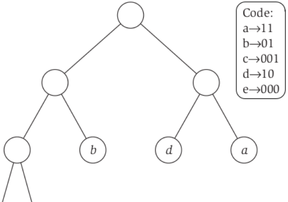
$$f_a = .32, f_b = .25, f_c = .2, f_d = .18, f_e = .05$$

$\textsc{abl}(\textsc{opt}) = 2.23$



Code:
a→11
b→10
c→01
d→001
e→000

$\textsc{abl}(\text{TopDown}) = 2.25$



Code:
a→11
b→01
c→001
d→10
e→000

HISTORICAL BREAK

HUFFMAN'S EPIPHANY

The Background

Huffman was a student in an information theory class taught by Robert Fano, who was a close colleague of Claude Shannon, the father of information theory. Fano and Shannon had previously developed a different greedy algorithm for producing prefix codes — split the frequency array into two subarrays as evenly as possible, and then recursively build a code for each subarray — but these Fano-Shannon codes were known not to be optimal.

HUFFMAN'S EPIPHANY

The Background

Huffman was a student in an information theory class taught by Robert Fano, who was a close colleague of Claude Shannon, the father of information theory. Fano and Shannon had previously developed a different greedy algorithm for producing prefix codes — split the frequency array into two subarrays as evenly as possible, and then recursively build a code for each subarray — but these Fano-Shannon codes were known not to be optimal.

The Challenge

Fano posed the problem of finding an optimal prefix code to his class. Huffman decided to solve the problem as a class project, instead of taking a final exam, not realizing that the problem was open, or that Fano and Shannon had already tried and failed to solve it.

THE EPIPHANY

Months of Effort

After several months of fruitless effort, Huffman eventually gave up and decided to take the final exam after all. As he was throwing his notes in the trash, the solution dawned on him.

THE EPIPHANY

Months of Effort

After several months of fruitless effort, Huffman eventually gave up and decided to take the final exam after all. As he was throwing his notes in the trash, the solution dawned on him.

The Realization

Huffman would later describe the epiphany as:

> *“The absolute lightning of sudden realization.”*

THE EPIPHANY

Months of Effort

After several months of fruitless effort, Huffman eventually gave up and decided to take the final exam after all. As he was throwing his notes in the trash, the solution dawned on him.

The Realization

Huffman would later describe the epiphany as:

"The absolute lightning of sudden realization."

The Result

Huffman's algorithm became the optimal method for generating prefix codes, known today as Huffman Coding.

WHAT IF WE KNEW THE OPTIMAL TREE?

Let T^* be the optimal (unlabeled) prefix tree.

Lemma 4

Let x and y be the two least frequent characters (breaking ties arbitrarily). There is an optimal code tree in which x and y are siblings and have the largest depth of any leaf.

Proof.

Let T be an optimal code tree, and suppose this tree has depth d. ☐

What time is it?

What if

Let T^*

Lemm

Let x an *ties*

arbitrar *y are*

siblings

WHAT IF WE KNEW THE OPTIMAL TREE?

Let T^* be the optimal (unlabeled) prefix tree.

Lemma 4

Let x and y be the two least frequent characters (breaking ties arbitrarily). There is an optimal code tree in which x and y are siblings and have the largest depth of any leaf.

- Since T^* is a full binary tree, it has at least two leaves at depth d that are siblings (not just one!).
- Suppose these two leaves are not x and y, but some other characters a and b.

WHAT IF WE KNEW THE OPTIMAL TREE?

Let T^* be the optimal (unlabeled) prefix tree.

Lemma 4

Let x and y be the two least frequent characters (breaking ties arbitrarily). There is an optimal code tree in which x and y are siblings and have the largest depth of any leaf.

- Let T' be the code tree obtained by swapping x and a, and let $\Delta = d - \text{depth}_{T'}(x)$.
- This swap increases the depth of x by Δ and decreases the depth of a by Δ, so:

$$\text{cost}(T') = \text{cost}(T) + \Delta \cdot (f[x] - f[a]).$$

WHAT IF WE KNEW THE OPTIMAL TREE?

Let T^* be the optimal (unlabeled) prefix tree.

Lemma 4

Let x and y be the two least frequent characters (breaking ties arbitrarily). There is an optimal code tree in which x and y are siblings and have the largest depth of any leaf.

- Our assumption that x is one of the two least frequent characters and a is not implies that $f[x] \leq f[a]$.
- Our assumption that a has maximum depth implies $\Delta \geq 0$.

$$\mathrm{cost}(T') = \mathrm{cost}(T) + \Delta \cdot (f[x] - f[a]).$$

It follows that $\mathrm{cost}(T') \leq \mathrm{cost}(T)$. Therefore, T' is optimal.

WHAT IF WE KNEW THE OPTIMAL TREE?

Let T^* be the optimal (unlabeled) prefix tree.

Lemma 4

Let x and y be the two least frequent characters (breaking ties arbitrarily). There is an optimal code tree in which x and y are siblings and have the largest depth of any leaf.

- Similarly, swapping y and b gives yet another optimal code tree.
- In this final optimal code tree, x and y are maximum-depth siblings, as required.

So the least frequent characters are at bottom

HUFFMAN'S EPIPHANY

HUFFMAN: Merge the two least frequent letters and recurse.

Let's see an example:

HUFFMAN'S EPIPHANY

HUFFMAN: Merge the two least frequent letters and recurse.

Let's see an example:

This sentence contains three a's, three c's, two d's, twenty-six e's, five f's, three g's, eight h's, thirteen i's, two l's, sixteen n's, nine o's, six r's, twenty-seven s's, twenty-two t's, two u's, five v's, eight w's, four x's, five y's, and only one z.

The pantagram of Sallow.

HUFFMAN'S EPIPHANY

HUFFMAN: Merge the two least frequent letters and recurse.

Let's see an example:

THISSENTENCECONTAINSTHREEASTHREECSTWODSTWENTYSIXESFIVEFST
HREEGSEIGHTHSTHIRTEENISTWOLSSIXTEENNSNINEOSSIXRSTWENTYSEV
ENSSTWENTYTWOTSTWOUSFIVEVSEIGHTWSFOURXSFIVEYSANDONLYONEZ[6]

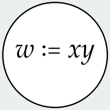
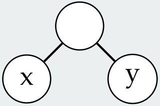
A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

The problem is the same if I normalize or not the frequencies!!!

BOTTOM-UP APPROACH

HUFFMAN CODE

Huffman's Algorithm

- (1) Bottom-up by lowest frequency:
    - Let x and y be the lowest frequency symbols.
    - Set $S := S \setminus \{x, y\} \cup \{w := xy\}$ and $f_w = f_x + f_y$.
    - Repeat until $|S| = 1$.
- (2) Generate the tree:
    - T := root with element from S.
    - Replace  with 
    - Repeat until leaves of T are original symbols.

HUFFMAN'S EPIPHANY

Back to our example:

THISSENTENCECONTAINSTHREEASTHREECSTWODSTWENTYSIXESFIVEFST
HREEGSEIGHTHSTHIRTEENISTWOLSSIXTEENNSNINEOSSIXRSTWENTYSEV
ENSSTWENTYTWOTSTWOUSFIVEVSEIGHTWSFOURXSFIVEYSANDONLYONEZ[6]

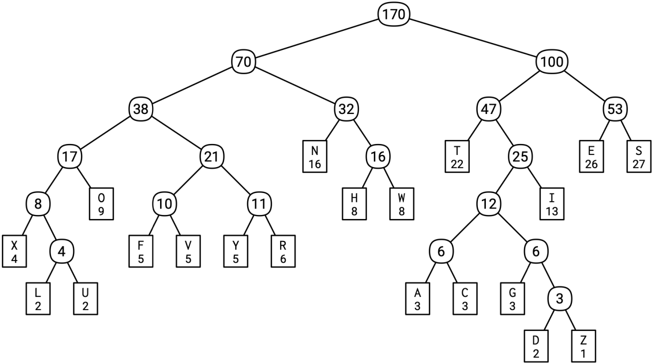
A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

$\Downarrow$ min freq: $\{D, Z\}$

A	C	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	DZ
3	3	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	3

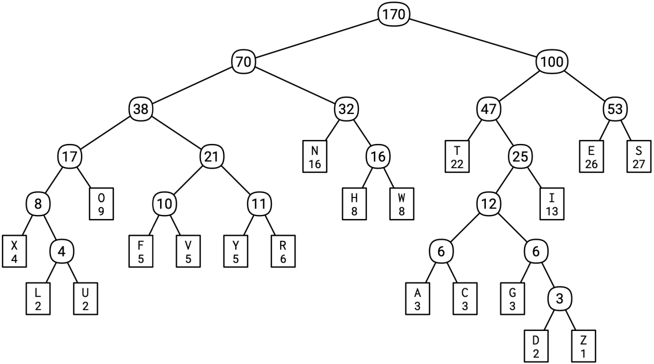
Huffman's Epiphany

Back to our example:



Huffman’s Epiphany

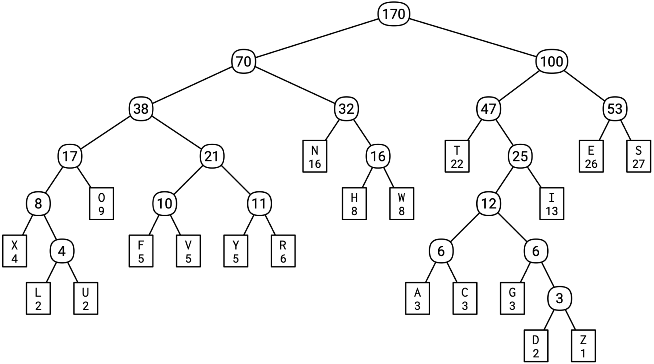
Back to our example:



char	A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
freq	3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1
depth	6	6	7	3	5	6	4	4	6	3	4	5	3	3	6	5	4	5	5	7
total	18	18	14	78	25	18	32	52	12	48	36	30	81	66	12	25	32	20	25	7

HUFFMAN'S EPIPHANY

Back to our example:



Encoding Sallows' sentence with this particular Huffman code would yield a bit string that starts like so:

100	0110	1011	111	111	110	010	100	110	010	101001	110	101001	0001	010	100	…
T	H	I	S	S	E	N	T	E	N	C	E	C	O	N	T	

HUFFMAN CODES ARE OPTIMAL

Lemma 5

Let T' be the tree at the $(k-1)$-st step, and let T be the tree at the k-th step. $\textsc{abl}(T') = \textsc{abl}(T) - f_w$, where w is the symbol replaced in the k-th step by y and z.

HUFFMAN CODES ARE OPTIMAL

Lemma 5

Let T' be the tree at the $(k-1)$-st step, and let T be the tree at the k-th step. $\textsc{abl}(T') = \textsc{abl}(T) - f_w$, where w is the symbol replaced in the k-th step by y and z.

Proof.

$$\begin{aligned}
\textsc{abl}(T) &= \sum_{x \in S} f_x \cdot \text{depth}(x) \\
&= f_y \cdot \text{depth}(y) + f_z \cdot \text{depth}(z) + \sum_{x \in S; x \notin \{y,z\}} f_x \cdot \text{depth}(x) \\
&= f_w + f_w \cdot \text{depth}(w) + \sum_{x \in S \setminus \{y,z\}} f_x \cdot \text{depth}(x) \\
&= f_w + \textsc{abl}(T')
\end{aligned}$$

☐

HUFFMAN CODES ARE OPTIMAL

Lemma 5

Let T' be the tree at the $(k-1)$-st step, and let T be the tree at the k-th step. $\textsc{abl}(T') = \textsc{abl}(T) - f_w$, where w is the symbol replaced in the k-th step by y and z.

Theorem 6

Huffman Algorithm is optimal.

HUFFMAN CODES ARE OPTIMAL

Lemma 5

Let T' be the tree at the $(k-1)$-st step, and let T be the tree at the k-th step. $\textsc{abl}(T') = \textsc{abl}(T) - f_w$, where w is the symbol replaced in the k-th step by y and z.

Theorem 6

Huffman Algorithm is optimal.

Proof.

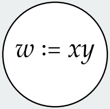
Reverse Induction: Reducing the problem size to smaller subproblems! ☺

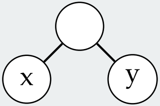
$\square$

BOTTOM-UP APPROACH

HUFFMAN CODE

Huffman's Algorithm

- (1) Bottom-up by lowest frequency:
    - Let x and y be the lowest frequency symbols.
    - Set $S := S \setminus \{x, y\} \cup \{w := xy\}$ and $f_w = f_x + f_y$.
    - Repeat until $|S| = 1$.
- (2) Generate the tree:
    - T := root with element from S.
    - Replace 

 with 

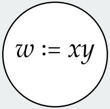
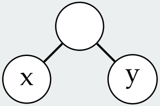
    - Repeat until leaves of T are original symbols.

Runtime:

Bottom-Up Approach

Huffman Code

Huffman's Algorithm

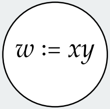
- (1) Bottom-up by lowest frequency:
  - Let x and y be the lowest frequency symbols.
  - Set $S := S \setminus \{x, y\} \cup \{w := xy\}$ and $f_w = f_x + f_y$.
  - Repeat until $|S| = 1$.
- (2) Generate the tree:
  - $T :=$ root with element from S.
  - Replace  with 
  - Repeat until leaves of T are original symbols.

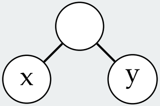
Runtime: $|S| - 1$ recursions with find min over $|S_i|$ elements

BOTTOM-UP APPROACH

HUFFMAN CODE

Huffman's Algorithm

- (1) Bottom-up by lowest frequency:
  - Let x and y be the lowest frequency symbols.
  - Set $S := S \setminus \{x, y\} \cup \{w := xy\}$ and $f_w = f_x + f_y$.
  - Repeat until $|S| = 1$.
- (2) Generate the tree:
  - T := root with element from S.
  - Replace 

with 

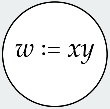
  - Repeat until leaves of T are original symbols.

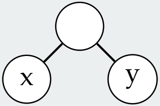
Runtime: $O(|S|^2)$

BOTTOM-UP APPROACH

HUFFMAN CODE

Huffman's Algorithm

- (1) Bottom-up by lowest frequency:
  - Let x and y be the lowest frequency symbols.
  - Set $S := S \setminus \{x, y\} \cup \{w := xy\}$ and $f_w = f_x + f_y$.
  - Repeat until $|S| = 1$.
- (2) Generate the tree:
  - T := root with element from S.
  - Replace 

with 

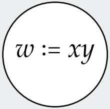
  - Repeat until leaves of T are original symbols.

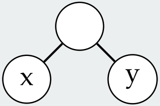
Runtime: $O(|S|^2)$
what about $O(|S| \log |S|)$?

BOTTOM-UP APPROACH

HUFFMAN CODE

Huffman's Algorithm

- (1) Bottom-up by lowest frequency:
  - Let x and y be the lowest frequency symbols.
  - Set $S := S \setminus \{x, y\} \cup \{w := xy\}$ and $f_w = f_x + f_y$.
  - Repeat until $|S| = 1$.
- (2) Generate the tree:
  - T := root with element from S.
  - Replace 

with 

  - Repeat until leaves of T are original symbols.

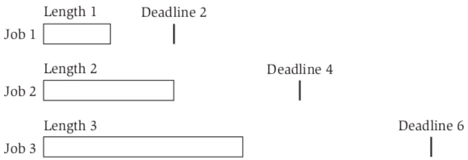
Runtime: $O(|S|^2)$
what about $O(|S| \log |S|)$? Priority Queue (min-heap)

HUFFMAN'S EPIPHANY

```
BUILDHUFFMAN($f[1..n]$):
    for $i \gets 1$ to $n$
        $L[i] \gets 0$;  $R[i] \gets 0$
        INSERT($i, f[i]$)
    for $i \gets n$ to $2n-1$
        $x \gets$ EXTRACTMIN( )        ⟨⟨find two rarest symbols⟩⟩
        $y \gets$ EXTRACTMIN( )
        $f[i] \gets f[x] + f[y]$       ⟨⟨merge into a new symbol⟩⟩
        INSERT($i, f[i]$)
        $L[i] \gets x$;  $P[x] \gets i$     ⟨⟨update tree pointers⟩⟩
        $R[i] \gets y$;  $P[y] \gets i$
    $P[2n-1] \gets 0$
```

EXCHANGE ARGUMENT: MINIMIZE MAX LATENESS

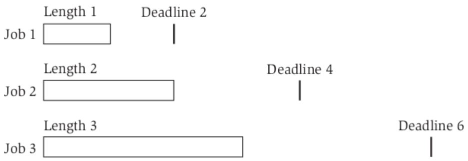
SCHEDULING PROBLEM: MINIMIZE MAX LATENESS



Problem Definition

- n jobs and a single machine that can process one job at a time

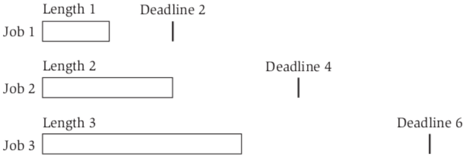
Scheduling Problem: Minimize Max Lateness



Problem Definition

- n jobs and a single machine that can process one job at a time
- For job i:

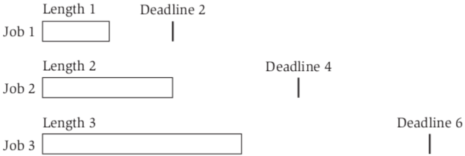
SCHEDULING PROBLEM: MINIMIZE MAX LATENESS



Problem Definition

- n jobs and a single machine that can process one job at a time
- For job i:
    - t_i is the processing time, d_i is the deadline.

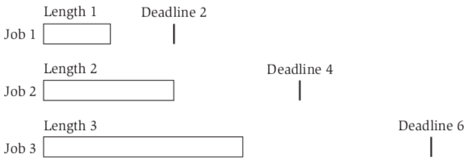
SCHEDULING PROBLEM: MINIMIZE MAX LATENESS



Problem Definition

- n jobs and a single machine that can process one job at a time
- For job i:
  - t_i is the processing time, d_i is the deadline.
  - Lateness $l_i = f_i - d_i$ if finish time $f_i > d_i$; 0 otherwise.

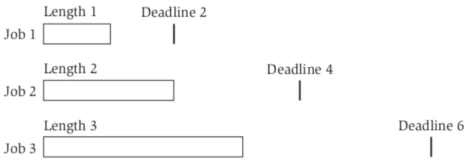
SCHEDULING PROBLEM: MINIMIZE MAX LATENESS



Problem Definition

- n jobs and a single machine that can process one job at a time
- For job i:
    - t_i is the processing time, d_i is the deadline.
    - Lateness $l_i = f_i - d_i$ if finish time $f_i > d_i$; 0 otherwise.
- Objective: Build a schedule for all the jobs that minimizes the max lateness.

SCHEDULING PROBLEM: MINIMIZE MAX LATENESS



Problem Definition

- n jobs and a single machine that can process one job at a time
- For job i:
  - t_i is the processing time, d_i is the deadline.
  - Lateness $l_i = f_i - d_i$ if finish time $f_i > d_i$; 0 otherwise.
- Objective: Build a schedule for all the jobs that minimizes the max lateness.

What greedy heuristic might work?

GREEDY ALGORITHMS FOR MINIMIZING MAX LATENESS

Heuristic 1: Increasing processing time.

Schedule jobs by increasing t_i.

GREEDY ALGORITHMS FOR MINIMIZING MAX LATENESS

Heuristic 1: Increasing processing time.

Schedule jobs by increasing t_i.

Optimal?

GREEDY ALGORITHMS FOR MINIMIZING MAX LATENESS

Heuristic 1: Increasing processing time.

Schedule jobs by increasing t_i.

Optimal?

Counter-example: Jobs (t_i, d_i): $\{(1, 100), (10, 10)\}$

Greedy Algorithms for Minimizing Max Lateness

Heuristic 2: Increasing slack.

Schedule by increasing $d_i - t_i$.

Greedy Algorithms for Minimizing Max Lateness

Heuristic 2: Increasing slack.

Schedule by increasing $d_i - t_i$.

Optimal?

GREEDY ALGORITHMS FOR MINIMIZING MAX LATENESS

Heuristic 2: Increasing slack.

Schedule by increasing $d_i - t_i$.

Optimal?

Counter-example:

$$\text{Jobs } (t_i, d_i)\text{: } \{(1,2),(10,10)\}$$

GREEDY ALGORITHMS FOR MINIMIZING MAX LATENESS

Heuristic 3: Earliest deadline first.

Schedule by increasing d_i.

GREEDY ALGORITHMS FOR MINIMIZING MAX LATENESS

Heuristic 3: Earliest deadline first.

Schedule by increasing d_i.

Optimal?

GREEDY ALGORITHMS FOR MINIMIZING MAX LATENESS

Heuristic 3: Earliest deadline first.

Schedule by increasing d_i.

Optimal?

Counter-example? Let's try and prove it.

EXERCISE: FORMALIZE THE ALGORITHM (PSEUDOCODE)

HEURISTIC 3: EARLIEST DEADLINE FIRST.

EXERCISE: FORMALIZE THE ALGORITHM (PSEUDOCODE)

HEURISTIC 3: EARLIEST DEADLINE FIRST.

Algorithm: EDF

```
Let $J$ be the set of jobs.
Let $S$ be an initially empty list.
while $J$ is not empty do
    Choose $j \in J$ with the smallest $d_i$ (break ties arbitrarily).
    Append $j$ to $S$.
end
return $S$
```

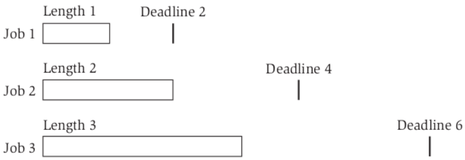
EXERCISE: FORMALIZE THE ALGORITHM (PSEUDOCODE)

HEURISTIC 3: EARLIEST DEADLINE FIRST.

Algorithm: EDF

```
Let $J$ be the set of jobs.
Let $S$ be an initially empty list.
while $J$ is not empty do
    Choose $j \in J$ with the smallest $d_i$ (break ties arbitrarily).
    Append $j$ to $S$.
end
return $S$
```

Sample Run (Quiz: What is max lateness?)



ANALYSIS OF EDF

Observation 1

There is an optimal schedule with no idle time.

ANALYSIS OF EDF

Observation 1

There is an optimal schedule with no idle time.

Showing Optimality

Let S^* be an optimal solution.

- Is it sufficient to show that $|S| = |S^*|$?

ANALYSIS OF EDF

Observation 1

There is an optimal schedule with no idle time.

Showing Optimality

Let S^* be an optimal solution.

- Is it sufficient to show that $|S| = |S^*|$? No.

ANALYSIS OF EDF

Observation 1

There is an optimal schedule with no idle time.

Showing Optimality

Let S^* be an optimal solution.

- Is it sufficient to show that $|S| = |S^*|$? No.
- Can there be multiple S^*?

ANALYSIS OF EDF

Observation 1

There is an optimal schedule with no idle time.

Showing Optimality

Let S^* be an optimal solution.

- Is it sufficient to show that $|S| = |S^*|$? No.
- Can there be multiple S^*? Yes.

ANALYSIS OF EDF

Observation 1

There is an optimal schedule with no idle time.

Showing Optimality

Let S^* be an optimal solution.

- Is it sufficient to show that $|S| = |S^*|$? No.
- Can there be multiple S^*? Yes.
- We need to show either $S = S^*$, or $S \equiv S^*$ for max lateness.

ANALYSIS OF EDF

Observation 1

There is an optimal schedule with no idle time.

Showing Optimality

Let S^* be an optimal solution.

- Is it sufficient to show that $|S| = |S^*|$? No.
- Can there be multiple S^*? Yes.
- We need to show either $S = S^*$, or $S \equiv S^*$ for max lateness.
- Technique: “Exchange Argument”

ANALYSIS OF EDF

Observation 1

There is an optimal schedule with no idle time.

Showing Optimality

Let S^* be an optimal solution.

- Is it sufficient to show that $|S| = |S^*|$? No.
- Can there be multiple S^*? Yes.
- We need to show either $S = S^*$, or $S \equiv S^*$ for max lateness.
- Technique: "Exchange Argument"
  - Start with an optimal solution S^* and transform it over a series of steps to something equivalent to S while maintaining optimality.

ANALYSIS OF EDF

Observation 1

There is an optimal schedule with no idle time.

Showing Optimality

Let S^* be an optimal solution.

- Is it sufficient to show that $|S| = |S^*|$? No.
- Can there be multiple S^*? Yes.
- We need to show either $S = S^*$, or $S \equiv S^*$ for max lateness.
- Technique: "Exchange Argument"
  - Start with an optimal solution S^* and transform it over a series of steps to something equivalent to S while maintaining optimality.
  - $S^* \equiv S_1 \equiv S_2 \equiv \cdots \equiv S$ for max lateness.

EXCHANGE ARGUMENT ANALYSIS

Definition 7

A schedule A has an inversion if the are jobs i and j with i scheduled before j and $d_j < d_i$.

EXCHANGE ARGUMENT ANALYSIS

Definition 7

A schedule A has an inversion if the are jobs i and j with i scheduled before j and $d_j < d_i$.

Lemma 8

All schedules with no inversions and no idle time have the same lateness.

EXCHANGE ARGUMENT ANALYSIS

Definition 7

A schedule A has an inversion if the are jobs i and j with i scheduled before j and $d_j < d_i$.

Lemma 8

All schedules with no inversions and no idle time have the same lateness.

Proof.

EXCHANGE ARGUMENT ANALYSIS

Definition 7

A schedule A has an inversion if the are jobs i and j with i scheduled before j and $d_j < d_i$.

Lemma 8

All schedules with no inversions and no idle time have the same lateness.

Proof.

- Only vary in jobs with the same deadline.
- Jobs with same deadline must sequential.
- Ordering of jobs with same deadline won't change lateness.

□

ANALYSIS OF EDF

Theorem 9

There is an optimal schedule that has no inversions and no idle time.

Analysis of EDF

Theorem 9

There is an optimal schedule that has no inversions and no idle time.

Proof.

ANALYSIS OF EDF

Theorem 9

There is an optimal schedule that has no inversions and no idle time.

Proof.

- If S^* has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has $d_j < d_i$.

ANALYSIS OF EDF

Theorem 9

There is an optimal schedule that has no inversions and no idle time.

Proof.

- If S^* has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has $d_j < d_i$.
- We will swap i and j to create a new schedule S'. Note that S' has one less inversion than S^*.

ANALYSIS OF EDF

Theorem 9

There is an optimal schedule that has no inversions and no idle time.

Proof.

- If S^* has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has $d_j < d_i$.
- We will swap i and j to create a new schedule S'. Note that S' has one less inversion than S^*.
- We need to show that S' has the same max lateness as S^*:

ANALYSIS OF EDF

Theorem 9

There is an optimal schedule that has no inversions and no idle time.

Proof.

- If S^* has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has $d_j < d_i$.
- We will swap i and j to create a new schedule S'. Note that S' has one less inversion than S^*.
- We need to show that S' has the same max lateness as S^*:
  - Swapping i and j means that l'_j (lateness in S') is less than that in S^*.

ANALYSIS OF EDF

Theorem 9

There is an optimal schedule that has no inversions and no idle time.

Proof.

- If S^* has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has $d_j < d_i$.
- We will swap i and j to create a new schedule S'. Note that S' has one less inversion than S^*.
- We need to show that S' has the same max lateness as S^*:
    - Swapping i and j means that l'_j (lateness in S') is less than that in S^*.
    - Lateness of i may increase, but: $l'_i = f'_i - d_i = f^*_j - d_i \leq f^*_j - d_j = l^*_j$.

ANALYSIS OF EDF

Theorem 9

There is an optimal schedule that has no inversions and no idle time.

Proof.

- If S^* has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has $d_j < d_i$.
- We will swap i and j to create a new schedule S'. Note that S' has one less inversion than S^*.
- We need to show that S' has the same max lateness as S^*:
  - Swapping i and j means that l'_j (lateness in S') is less than that in S^*.
  - Lateness of i may increase, but: $l'_i = f'_i - d_i = f^*_j - d_i \leq f^*_j - d_j = l^*_j$.
- Let $S^* := S'$ and repeat until no more inversions.

☐

EDF IS OPTIMAL

Corollary 10

EDF produces an optimal schedule.

Proof.

EDF IS OPTIMAL

Corollary 10

EDF *produces an optimal schedule.*

Proof.

- EDF produces a schedule with no inversions and no idle time.
- From Theorem 9, there is an optimal schedule with no inversions and no idle time.
- Lemma 8 shows that these two schedules have the same max lateness.

☐

EDF IS OPTIMAL

Corollary 10

EDF produces an optimal schedule.

Proof.

- EDF produces a schedule with no inversions and no idle time.
- From Theorem 9, there is an optimal schedule with no inversions and no idle time.
- Lemma 8 shows that these two schedules have the same max lateness.

☐

Run time: .

EDF IS OPTIMAL

Corollary 10

EDF produces an optimal schedule.

Proof.

- EDF produces a schedule with no inversions and no idle time.
- From Theorem 9, there is an optimal schedule with no inversions and no idle time.
- Lemma 8 shows that these two schedules have the same max lateness.

☐

Run time: Sort the jobs by deadline: $O(n \log n)$.

Appendix

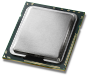
REFERENCES

IMAGE SOURCES I

https://www.cse.unsw.edu.au/~cs1521/17s2/lecs/notices/slide068.html

http://mediablogrueil.blogspot.fr/2012/11/one-page-design-effet-de-mode-ou-reel.html

http://www.culturizame.es/articulo/nuestro-pequeno-diccionario-de-tecnologia



http://computer-help-tips.blogspot.fr/2011/04/different-types-of-computer-processors.html

Image Sources II

https://brand.wisc.edu/web/logos/