4.2 Scheduling to Minimize Lateness

**Minimizing lateness problem.**
- Single resource processes one job at a time.
- Job $j$ requires $t_j$ units of processing time and its deadline is time $d_j$.
- If $j$ starts at time $s_j$, it finishes at time $f_j = s_j + t_j$.
- Lateness: $l_j = \max(0, f_j - d_j)$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max l_j$.

**Example:**

<table>
<thead>
<tr>
<th>$d_1$</th>
<th>$d_2$</th>
<th>$d_3$</th>
<th>$d_4$</th>
<th>$d_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>9</td>
<td>8</td>
<td>14</td>
</tr>
</tbody>
</table>

**Minimizing Lateness: Greedy Algorithms**

- **Greedy template.** Consider jobs in some order.
  - [Shortest processing time first] Consider jobs in ascending order of processing time $t_j$.
  - [Earliest deadline first] Consider jobs in ascending order of deadline $d_j$.
  - [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

**Greedy Algorithm.** Earliest deadline first.

1. Sort $n$ jobs by deadline so that $d_1 \leq d_2 \leq \ldots \leq d_n$.
2. For $j = 1$ to $n$:
   - Assign job $j$ to interval $[t, t + t_j]$.
   - $t = t + t_j$.
3. Output intervals $[s_j, f_j]$.

**Counterexample:**

<table>
<thead>
<tr>
<th>$d_1$</th>
<th>$d_2$</th>
<th>$d_3$</th>
<th>$d_4$</th>
<th>$d_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>11</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

**Greedy algorithm.** Consider jobs in descending order of processing time $t_j$.

1. Consider jobs in descending order of processing time $t_j$.
2. For $j = 1$ to $n$:
   - Assign job $j$ to interval $[t, t + t_j]$.
   - $t = t + t_j$.
3. Output intervals $[s_j, f_j]$.

**Counterexample:**

<table>
<thead>
<tr>
<th>$d_1$</th>
<th>$d_2$</th>
<th>$d_3$</th>
<th>$d_4$</th>
<th>$d_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

**Counterexample:**

<table>
<thead>
<tr>
<th>$d_1$</th>
<th>$d_2$</th>
<th>$d_3$</th>
<th>$d_4$</th>
<th>$d_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>15</td>
<td>14</td>
<td>15</td>
<td>14</td>
</tr>
</tbody>
</table>
Minimizing Lateness: No Idle Time

Observation. There exists an optimal schedule with no idle time.

\[ d = 4 \quad d = 6 \quad d = 12 \]

Observation. The greedy schedule has no idle time.

Minimizing Lateness: Inversions

Def. Given a schedule \( S \), an inversion is a pair of jobs \( i \) and \( j \) such that: \( d_i < d_j \) but \( j \) scheduled before \( i \).

Observation. Greedy schedule has no inversions.

Observation. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

Claim. Swapping two consecutive, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

Pf. Let \( \lambda \) be the lateness before the swap, and let \( \lambda' \) be it afterwards.

- For all jobs except \( i, j \), \( \lambda' (k) = \lambda (k) \)
- If job \( j \) is late:
  - \( \lambda' (j) = \lambda (j) - d_j \) (definition)
  - \( \lambda' (i) = \lambda (i) + d_i \) (job \( i \) finishes at time \( \lambda' (j) \))
  - \( \lambda' (i) < \lambda' (j) \) (definition)

Greedy Analysis Strategies

Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as that of any other algorithm.

Structural. Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Exchange argument. Gradually transform any optimal solution to the one found by the greedy algorithm without hurting its quality.

4.3 Optimal Caching
Optimal Offline Caching

Caching:
- Before computation, items are stored on cheap and slow disks.
- Cache is fast memory which can store k items.
- Computation makes a sequence of m item requests \( d_1, d_2, \ldots, d_m \).
- Cache hit: item already in cache when requested.
- Cache miss: item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.

Goal: Eviction schedule that minimizes number of cache misses.

Ex: \( k = 2 \), initial cache = ab, requests: a, b, c, b, c, a, a, b.
Optimal eviction schedule: 2 cache misses.

Farthest-In-Future (FF): Evict item in the cache that is not requested until farthest in the future.

**Theorem.** [Bellady, 1960s] FF is an optimal eviction schedule.

**Pf.** Algorithm and theorem are intuitive; proof is subtle.

Reduced Eviction Schedules

**Def.** A reduced schedule is a schedule that only inserts an item into the cache in a step in which that item is requested.

**Intuition.** Can transform an unreduced schedule into a reduced one with no more cache misses.

Theorem. (Bellady, 1960s) FF is an optimal eviction schedule.

**Pf.** Algorithm and theorem are intuitive; proof is subtle.

Farthest-In-Future: Analysis

**Theorem.** FF is an optimal eviction algorithm.

**Pf.** (by induction on number of requests \( j \))

- Case 3: \( d \) is not in the cache. \( S^j \) exists; \( S \) exists \( f \neq e \).
- Begin construction of \( S' \) from \( S \) by evicting \( e \) instead of \( f \).

Reduced Eviction Schedules

**Claim.** Any unreduced schedule \( S \) can be transformed into a reduced schedule \( S' \) with no more cache misses.

**Pf.** (by induction on number of unreduced items)

- Suppose \( S \) brings \( d \) into the cache at time \( t \), without a request.
- Let \( c \) be the item \( S \) evicts when it brings \( d \) into the cache.
- Case 1: \( d \) is already in the cache. \( S' = S \) satisfies invariant.
- Case 2: \( d \) is not in the cache and \( S \) and \( S' \) evict the same element.
- Case 3: \( d \) is not in the cache; \( S' \) evicts \( e \); \( S \) evicts \( f \neq e \).
Farthest-In-Future: Analysis

Let \( j' \) be the first time after \( j+1 \) that \( S \) and \( S' \) take a different action, and let \( g \) be item requested at time \( j' \).

- Case 3a: \( g = e \). Can't happen with Farthest-In-Future since there must be a request for \( f \) before \( e \).
- Case 3b: \( g = f \). Element \( f \) can't be in cache of \( S \), so let \( e' \) be the element that \( S \) evicts.
  - if \( e' = e \), \( S' \) accesses \( f \) from cache; now \( S \) and \( S' \) have same cache.
  - if \( e' \neq e \), \( S' \) evicts \( e' \) and brings \( e \) into cache; now \( S \) and \( S' \) have same cache.

Note: \( S' \) is no longer reduced, but can be transformed into a reduced schedule that agrees with \( S \) through step \( j \).

Caching Perspective

Online vs. offline algorithms.
- Offline: full sequence of requests is known a priori.
- Online (reality): requests are not known in advance.
- Caching is among most fundamental online problems in CS.
  - LIFO. Evict page brought in most recently.
  - LRU. Evict page whose most recent access was earliest.
  - FF with direction of time reversed.

Theorem. FF is optimal offline eviction algorithm.
- Provides basis for understanding and analyzing online algorithms.
- LRU is \( k \)-competitive. (Section 13.8)
- LIFO is arbitrarily bad.

4.8 Huffman Codes

Data Compression

Q. Given a text that uses 32 symbols (26 different letters, space, and some punctuation characters), how can we encode this text in bits?

Q. Some symbols (e, t, a, o, i, n) are used far more often than others. How can we use this to reduce our encoding?

Q. How do we know when the next symbol begins?
**Data Compression**

Q. Given a text that uses 32 symbols (26 different letters, space, and some punctuation characters), how can we encode this text in bits?

A. We can encode 32 different symbols using a fixed length of 5 bits per symbol. This is called fixed length encoding.

Q. Some symbols (e, t, a, o, i, n) are used far more often than others. How can we use this to reduce our encoding?

A. Encode these characters with fewer bits, and the others with more bits.

Q. How do we know when the next symbol begins?

A. Use a separation symbol (like the pause in Morse), or make sure that there is no ambiguity by ensuring that no code is a prefix of another one.

Ex. 

c(a) = 01

c(b) = 010

c(e) = 1

**Prefix Codes**

Definition: A prefix code for a set S is a function c that maps each symbol \( x \) to 1s and 0s in such a way that for \( x, y \in S \), \( x \neq y \), \( c(x) \) is not a prefix of \( c(y) \).

Ex. 

c(a) = 11

c(e) = 01

c(k) = 001

c(l) = 10

c(u) = 000

Q. What is the meaning of 1001000001?

A. “leuk”

Suppose frequencies are known in a text of 1G:

\[ f_a = 0.4, \ f_e = 0.2, \ f_k = 0.2, \ f_l = 0.1, \ f_u = 0.1 \]

Q. What is the size of the encoded text?

A. \( 2 \times f_a + 2 \times f_e + 3 \times f_k + 2 \times f_l + 3 \times f_u = 2.4 \) bits per letter

**Optimal Prefix Codes**

Definition. The average bits per letter of a prefix code \( c \) is the sum over all symbols of its frequency times the number of bits of its encoding:

\[
\text{ABPL}(c) = \sum_x f_x \times |c(x)|
\]

Optimality: We would like to find a prefix code that has the lowest possible average bits per letter.

Suppose we model a code in a binary tree...

Q. How does the tree of a prefix code look?

A. Only the leaves have a label.

Pf. An encoding of \( x \) is a prefix of an encoding of \( y \) if and only if the path of \( x \) is a prefix of the path of \( y \).
Q. What is the meaning of 111010001111101000?
A. “simpel”

Q. How can this prefix code be made more efficient?
A. Change encoding of p and s to a shorter one.
This tree is now full.

Definition. A tree is full if every node that is not a leaf has two children.

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

Pf. (by contradiction)
- Suppose T is binary tree of optimal prefix code and is not full.
- This means there is a node u with only one child v.
- Case 1: u is the root; delete u and use v as the root.
- Case 2: u is not the root
  - let w be the parent of u
  - delete u and make v be a child of w in place of u.
- In both cases the number of bits needed to encode any leaf in the subtree of v is decreased. The rest of the tree is not affected.
- Clearly this new tree T has a smaller ABL than T. Contradiction.

Q. Where in the tree of an optimal prefix code should letters be placed with a high frequency?
Optimal Prefix Codes: False Start

Q. Where in the tree of an optimal prefix code should letters be placed with a high frequency?
A. Near the top.

Greedy template. Create tree top-down, split S into two sets S₁ and S₂ with (almost) equal frequencies. Recursively build tree for S₁ and S₂.

[Shannon-Fano, 1949]

fa = 0.32, fe = 0.25, fk = 0.20, fl = 0.18, fu = 0.05

Optimal Prefix Codes: Huffman Encoding

Observation. Lowest frequency items should be at the lowest level in tree of optimal prefix code.

Observation. For n > 1, the lowest level always contains at least two leaves.

Observation. The order in which items appear in a level does not matter.

Claim. There is an optimal prefix code with tree T* where the two lowest-frequency letters are assigned to leaves that are siblings in T*.

Greedy template. [Huffman, 1952] Create tree bottom-up. Make two leaves for two lowest-frequency letters y and z. Recursively build tree for the rest using a meta-letter for yz.

Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.

PF. by induction, based on optimality of T' (y and z removed, ω added) (see next page)

Claim. ABL(T') = ABL(T) - fω

PF. ABL(T') = \sum \frac{f_i \cdot \text{depth}_T(x)}{2^i} = \frac{f_\omega \cdot \text{depth}_T(\omega)}{2} + \sum_{i \neq \omega} \frac{f_i \cdot \text{depth}_T(x)}{2^i} = \frac{f_\omega \cdot (\text{depth}_T(\omega) - \text{depth}_T(y) - \text{depth}_T(z))}{2} + \sum_{i \neq \omega} \frac{f_i \cdot \text{depth}_T(x)}{2^i} = \frac{f_\omega \cdot \text{depth}_T(\omega)}{2} + \sum_{i \neq \omega} \frac{f_i \cdot \text{depth}_T(x)}{2^i} = f_\omega + ABL(T')
Claim. Huffman code for $S$ achieves the minimum ABL of any prefix code.

Proof. (by induction over $|S|$)

Base: For $n=2$ there is no shorter code than root and two leaves.

Hypothesis: Suppose Huffman tree $T'$ for $S'$ of size $n-1$ with $\omega$ instead of $y$ and $z$ is optimal.

Step: (by contradiction)

- Idea of proof:
  - Suppose other tree $Z$ of size $n$ is better.
  - Delete lowest frequency items $y$ and $z$ from $Z$ and add $\omega$ creating $Z'$
  - $Z'$ cannot be better than $T'$ by IH.

- Similar $T'$ is derived from $S'$ in our algorithm.
- We know that $ABL(Z')=ABL(Z')-f_\omega$ as well as $ABL(T')=ABL(T')-f_\omega$
- But also $ABL(Z')=ABL(Z')-f_\omega$
- Contradiction with IH.