Why lots of memory is “bad”

- Using too much memory forces you to use slow memory
- Memory == $$
- May have too little memory for the algorithm to even run
- Lots of memory => not parallizable
- Contention for the memory
- See lecture slides on counting sort
- Memory <= time
Why lots of memory is “bad”

- Von Neumann bottleneck
- Don’t have enough memory
- Cache coherency
- Time $\geq$ space
- Fast memory is expensive
Warm up

Why is an algorithm’s space complexity (how much memory it uses) important?

Why might a memory-intensive algorithm be a “bad” one?
Why lots of memory is “bad”
Today’s Keywords

• Greedy Algorithms
• Choice Function
• Cache Replacement
• Hardware & Algorithms
• Chapter 16
Homeworks

• HW6 Due Tuesday, November 5 @ 11pm
  – Written (use latex)
  – DP and Greedy

• HW10A also due Tuesday, November 5 @ 11pm
  – No late submissions allowed

• HW4 and HW5 grades coming soon
Overview:
- Show that there is an optimal tree in which the least frequent characters are siblings
  - Exchange argument
- Show that making them siblings and solving the new smaller sub-problem results in an optimal solution
  - Proof by contradiction

Greedy Choice Property
Optimal Substructure works
Huffman Exchange Argument

• **Claim:** if $c_1, c_2$ are the least-frequent characters, then there is an optimal prefix-free code s.t. $c_1, c_2$ are siblings
  – i.e. codes for $c_1, c_2$ are the same length and differ only by their last bit

Case 1: Consider some optimal tree $T_{opt}$. If $c_1, c_2$ are siblings in this tree, then claim holds
• **Claim:** if $c_1, c_2$ are the least-frequent characters, then there is an optimal prefix-free code s.t. $c_1, c_2$ are siblings
  
  – i.e. codes for $c_1, c_2$ are the same length and differ only by their last bit

  Case 2: Consider some optimal tree $T_{opt}$, in which $c_1, c_2$ are not siblings

  Let $a, b$ be the two characters of lowest depth that are siblings
  (Why must they exist?)

  Idea: show that swapping $c_1$ with $a$ does not increase cost of the tree.

  Similar for $c_2$ and $b$

  Assume: $f_{c_1} \leq f_a$ and $f_{c_2} \leq f_b$
Optimal Substructure

- **Claim**: An optimal solution for $F$ involves finding an optimal solution for $F'$, then adding $c_1, c_2$ as children to $\sigma$.
• Why is using too much memory a bad thing?
Von Neumann Bottleneck

- Named for John von Neumann
- Inventor of modern computer architecture
- Other notable influences include:
  - Mathematics
  - Physics
  - Economics
  - Computer Science
Von Neumann Bottleneck

- Reading from memory is VERY slow
- Big memory = slow memory
- Solution: hierarchical memory
- Takeaway for Algorithms: Memory is time, more memory is a lot more time

![Diagram of memory hierarchy](image)

- CPU, registers
  - Access time: 1 cycle
- Cache
  - Access time: 10 cycles
- Disk
  - Access time: 1,000,000 cycles
- Hopefully your data in here
- If not look here
- Hope it’s not here
• Cache misses are very expensive
• When we load something new into cache, we must eliminate something already there
• We want the best cache “schedule” to minimize the number of misses
Caching Problem Definition

• Input:
  – \( k = \) size of the cache
  – \( M = [m_1, m_2, \ldots m_n] = \) memory access pattern

• Output:
  – “schedule” for the cache (list of items in the cache at each time) which minimizes cache fetches
Example

A   B   C   D   A   D   E   A   D   B   A   E   C   E   A

A   B   C

A   B   C

A   C   D   A   D   E   A   D   B   A   E   C   E   A

✓   ✓
Example

A   B   C   D   A   D   E   A   D   B   A   E   C   E   A

A   B   C

A  B  C  D  A  D  E  A  D  B  A  E  C  E  A
We must evict something to make room for D.
Example

If we evict A
Example

If we evict C
Our Problem vs Reality

- Assuming we know the entire access pattern
- Cache is Fully Associative
- Counting # of fetches (not necessarily misses)
- “Reduced” Schedule: Address only loaded on the cycle it’s required
  - Reduced == Unreduced (by number of fetches)

![Diagram showing reduced and unreduced schedules]

Leaving A in longer does not save fetches
Greedy Algorithms

• Require Optimal Substructure
  – Solution to larger problem contains the solution to a smaller one
  – Only one subproblem to consider!

• Idea:
  1. Identify a greedy choice property
     • How to make a choice guaranteed to be included in some optimal solution
  2. Repeatedly apply the choice property until no subproblems remain
Greedy choice property

- Belady evict rule:
  - Evict the item accessed farthest in the future
Greedy choice property

• Belady evict rule:
  – Evict the item accessed farthest in the future
• Belady evict rule:
  – Evict the item accessed farthest in the future
Greedy choice property

• Belady evict rule:
  – Evict the item accessed farthest in the future
Greedy choice property

• Belady evict rule:
  – Evict the item accessed farthest in the future

4 Cache Misses
• Require **Optimal Substructure**
  – Solution to larger problem contains the solution to a smaller one
  – Only one subproblem to consider!
• Idea:
  1. Identify a greedy **choice property**
     • How to make a choice guaranteed to be included in some optimal solution
  2. Repeatedly apply the choice property until no subproblems remain
Caching Greedy Algorithm

Initialize $\text{cache} = \text{first k accesses } O(k)$

For each $m_i \in M$: $n$ times

\begin{align*}
\text{if } m_i \in \text{cache: } & O(k) \\
\text{print cache } & O(k) \\
\text{else: } \\
\text{m = furthest-in-future from cache } & O(kn) \\
\text{evict m, load } m_i & O(1) \\
\text{print cache } & O(k)
\end{align*}

$O(kn^2)$
Exchange argument

• Shows correctness of a greedy algorithm

• Idea:
  – Show exchanging an item from an arbitrary optimal solution with your greedy choice makes the new solution no worse
  – How to show my sandwich is at least as good as yours:
    • Show: “I can remove any item from your sandwich, and it would be no worse by replacing it with the same item from my sandwich”
Belady Exchange Lemma

Let $S_{ff}$ be the schedule chosen by our greedy algorithm.

Let $S_i$ be a schedule which agrees with $S_{ff}$ for the first $i$ memory accesses.

We will show: there is a schedule $S_{i+1}$ which agrees with $S_{ff}$ for the first $i + 1$ memory accesses, and has no more misses than $S_i$ (i.e. $\text{misses}(S_{i+1}) \leq \text{misses}(S_i)$)
Belady Exchange Proof Idea

First $i$ accesses

$S_i$

$S_{i+1}$

Must agree with $S_{ff}$

Need to fill in the rest of $S_{i+1}$ to have no more misses than $S_i$
Proof of Lemma

Goal: find $S_{i+1}$ s.t. $\text{misses}(S_{i+1}) \leq \text{misses}(S_i)$

Since $S_i$ agrees with $S_{ff}$ for the first $i$ accesses, the state of the cache at access $i + 1$ will be the same

Consider access $m_{i+1} = d$

Case 1: if $d$ is in the cache, then neither $S_i$ nor $S_{ff}$ evict from the cache, use the same cache for $S_{i+1}$
Proof of Lemma

Goal: find $S_{i+1}$ s.t. $\text{misses}(S_{i+1}) \leq \text{misses}(S_i)$

Since $S_i$ agrees with $S_{ff}$ for the first $i$ accesses, the state of the cache at access $i + 1$ will be the same

\[
S_i \text{ Cache after } i \quad e \quad f \quad = \quad S_{ff} \text{ Cache after } i \quad e \quad f
\]

Consider access $m_{i+1} = d$

Case 2: if $d$ isn’t in the cache, and both $S_i$ and $S_{ff}$ evict $f$ from the cache, evict $f$ for $d$ in $S_{i+1}$

\[
S_{i+1} \text{ Cache after } i \quad e \quad d
\]
Proof of Lemma

Goal: find $S_{i+1}$ s.t. $\text{misses}(S_{i+1}) \leq \text{misses}(S_i)$

Since $S_i$ agrees with $S_{ff}$ for the first $i$ accesses, the state of the cache at access $i + 1$ will be the same

\[
\begin{array}{c|cc}
S_i \text{ Cache after } i & e & f \\
\end{array}
= \begin{array}{c|cc}
S_{ff} \text{ Cache after } i & e & f \\
\end{array}
\]

Consider access $m_{i+1} = d$

Case 3: if $d$ isn’t in the cache, $S_i$ evicts $e$ and $S_{ff}$ evicts $f$ from the cache

\[
\begin{array}{c|cc}
S_i \text{ Cache after } i + 1 & d & f \\
\end{array}
\neq \begin{array}{c|cc}
S_{ff} \text{ Cache after } i + 1 & e & d \\
\end{array}
\]
Case 3

First $i$ accesses

$S_i$

$S_{i+1}$

Must agree with $S_{ff}$

Need to fill in the rest of $S_{i+1}$ to have no more misses than $S_i$
Case 3

First $i$ accesses

$S_i$

Copy $S_i$

First place $S_i$ involves $e$ or $f$

$S_{i+1}$

$S_{ff}$

$m_t = \text{the first access after } i + 1 \text{ in which } S_i \text{ deals with } e \text{ or } f$

3 options: $m_t = e$ or $m_t = f$ or $m_t = x \neq e, f$
Case 3, \( m_t = e \)

First \( i \) accesses

\( S_i \)

Copy \( S_i \)

First place \( S_i \) uses \( e \) or \( f \)

\( S_{i+1} \)

\( m_t = \) the first access after \( i + 1 \) in which \( S_i \) deals with \( e \) or \( f \)

3 options: \( m_t = e \) or \( m_t = f \) or \( m_t = x \neq e, f \)
Case 3, $m_t = e$

**Goal:** find $S_{i+1}$ s.t. $\text{misses}(S_{i+1}) \leq \text{misses}(S_i)$

$S_i$ Cache after $t - 1$:

\[
\begin{array}{ccc}
S_i & x & d & f \\
& e & & \\
\end{array}
\]

$S_i$ must load $e$ into the cache, assume it evicts $x$

$S_{i+1}$ Cache after $t - 1$:

\[
\begin{array}{ccc}
S_{i+1} & x & e & d \\
& f & & \\
\end{array}
\]

$S_{i+1}$ will load $f$ into the cache, evicting $x$

The caches now match!

$S_{i+1}$ behaved exactly the same as $S_i$ between $i$ and $t$, and has the same cache after $t$, therefore $\text{misses}(S_{i+1}) = \text{misses}(S_i)$
Case 3, $m_t = f$

$m_t = \text{the first access after } i + 1 \text{ in which } S_i \text{ deals with } e \text{ or } f$

3 options: $m_t = e$ or $m_t = f$ or $m_t = x \neq e, f$
Case 3, $m_t = f$

**Cannot Happen!**

- $S_i$: 
  
- $S_{i+1}$: "Evict $f$"

- First place $S_i$ uses $e$ or $f$
  Means $f$ not farthest future access!

- $S_{ff}$: "Evict $f$"
Case 3, $m_t = x \neq e, f$

$m_t = \text{the first access after } i + 1 \text{ in which } S_i \text{ deals with } e \text{ or } f$

3 options: $m_t = e$ or $m_t = f$ or $m_t = x \neq e, f$
Case 3, $m_t = x \neq e, f$

**Goal**: find $S_{i+1}$ s.t. $\text{misses}(S_{i+1}) \leq \text{misses}(S_i)$

- $S_i$ loads $x$ into the cache, it must be evicting $f$

- $S_{i+1}$ will load $x$ into the cache, evicting $e$

The caches now match!

- $S_{i+1}$ behaved exactly the same as $S_i$ between $i$ and $t$, and has the same cache after $t$, therefore $\text{misses}(S_{i+1}) = \text{misses}(S_i)$
Use Lemma to show Optimality

\[ S^* \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_{ff} \]

*Agrees with* \( S_{ff} \) *on first 0 accesses*

*Agrees with* \( S_{ff} \) *on first access*

*Agrees with* \( S_{ff} \) *on first 2 accesses*

*Agrees with* \( S_{ff} \) *on all* \( n \) *accesses*