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”

- limited by size of memory
- different speeds / sizes of memory
- memory is slow / CPU is fast
- cache misses
- fast memory = $$
- memory < time

more memory \Rightarrow slower memory
Today’s Keywords

• Greedy Algorithms
• Choice Function
• Cache Replacement
• Hardware & Algorithms
CLRS Readings

• Chapter 16
Homeworks

• HW6 Due Friday April 5 @11pm
  – Written (use latex)
  – DP and Greedy
Goal: Shortest Prefix-Free Encoding

• Input: A set of character frequencies $\{f_c\}$
• Output: A prefix-free code $T$ which minimizes

$$B(T, \{f_c\}) = \sum_{\text{character } c} \ell_c f_c$$

Huffman Coding!!
Huffman Algorithm

• Choose the least frequent pair, combine into a subtree
Huffman Algorithm

• Choose the least frequent pair, combine into a subtree

Subproblem of size $n - 1$!
Huffman Algorithm

- Choose the least frequent pair, combine into a subtree
REVIEW: Showing Huffman is Optimal

• 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
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 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 $\sigma$
Optimal Substructure

• **Claim:** An optimal solution for $F$ involves finding an optimal solution for $F'$, then adding $c_1, c_2$ as children to $\sigma$

If this is optimal

\[ f_\sigma = f_{c1} + f_{c2} \]

Then this is optimal

\[ B(T') = B(T) - f_{c1} - f_{c2} \]

\[ B(T) = C + f_{c1}(l_\sigma + 1) + f_{c2}(l_\sigma + 1) \]

\[ \ell_{c1} = \ell_\sigma + 1 \]
\[ \ell_{c2} = \ell_\sigma + 1 \]
Optimal Substructure

• **Claim**: An optimal solution for $F$ involves finding an optimal solution for $F'$, then adding $c_1, c_2$ as children to $\sigma$

Let $U$ be a lower-cost tree $B(U) < B(T)$

Toward contradiction

Suppose $T$ is not optimal

Let $U$ be a lower-cost tree $B(U) < B(T)$
Optimal Substructure

- **Claim**: An optimal solution for $F$ involves finding an optimal solution for $F'$, then adding $c_1, c_2$ as children to $\sigma$

\[
B(U) < B(T) \\
B(U') = B(U) - f_{c_1} - f_{c_2} < B(T) - f_{c_1} - f_{c_2} \\
= B(T') \\
B(U') < B(T')
\]

Contradicts optimality of $T'$, so $T$ is optimal!
Caching Problem

• 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

CPU, registers

Access time: 1 cycle

Cache

Access time: 10 cycles

RAM

Access time: 100+ cycles

Disk

Access time: 1,000,000+ cycles

If not look here

Hopefully your data in here

Access time: 1 cycle
Caching Problem

• 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
Example
Example
Example

We must evict something to make room for D.
Example

If we evict A

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

A   B   C   D   A   D   E   A   D   B   A   E   C   E   A
✓ ✓ ✓ ✗ ✗
Example

If we evict C

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

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

Unreduced

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

Reduced

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

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
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

![Diagram showing the Belady evict rule with a sequence of items accessed: A, B, C, D, A, D, E, A, D, B, A, E, C, E, A. The sequence is visualized with columns representing different time steps, where tick marks indicate items accessed and crosses indicate evictions. Evict B is highlighted.](image-url)
Greedy choice property

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

4 Cache Misses
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
Caching Greedy Algorithm

Initialize \textit{cache} = first k accesses \quad O(k)

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

\[ \mathcal{O}(kn^2) \]
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:

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

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 $S_{i+1}$

\[
\begin{array}{ccc}
S_{i+1} \text{ Cache after } i & e & d \\
\end{array}
\]
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

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

First $i$ accesses

$S_i$

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

Must agree with $S_{ff}$

$S_{i+1}$

$S_{ff}$
Case 3

First $i$ accesses

$S_i$

Copy $S_i$

$S_{i+1}$

First place $S_i$ involves $e$ or $f$ (at time $t$)

$S_{ff}$

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

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

First $i$ accesses

$S_i$

Copy $S_i$

$S_{i+1}$

First place $S_i$ uses $e$ or $f$

$S_{ff}$

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

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

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

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

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

$S_{i+1}$ 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, $m_t = f$

First $i$ accesses

$S_i$

Copy $S_i$

$S_{i+1}$

First place $S_i$ uses $e$ or $f$

$S_{ff}$

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

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

Cannot Happen!

\( S_i \)

"Evict \( f \)"

\( S_{i+1} \)

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

Means \( f \) not farthest future access!

\( S_{ff} \)

"Evict \( f \)"
Case 3, $m_t = x \neq e, f$

First $i$ accesses

$S_i$

Copy $S_i$

$S_{i+1}$

First place $S_i$ uses $e$ or $f$

$S_{\text{ff}}$

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

$m_t = e$ 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. $\text{misses}(S_{i+1}) \leq \text{misses}(S_i)$

$S_i$ Cache after $t - 1$

\[
\begin{array}{c}
S_i \text{ Cache after } t - 1 \\
d & f & x
\end{array}
\]

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

\[
\begin{array}{c}
S_{i+1} \text{ Cache after } t - 1 \\
\end{array}
\]

$S_i$ 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

$S^*$ Agrees with $S_{ff}$ on first 0 accesses

Lemma

$S_1$ Agrees with $S_{ff}$ on first access

Lemma

$S_2$ Agrees with $S_{ff}$ on first 2 accesses

Lemma

... Lemma

$S_{ff}$ Agrees with $S_{ff}$ on all $n$ accesses