Take-Aways

• Have optimal substructure

  - Similar to divide and conquer but there the choice is obvious or
  - Subproblems
  - Unlike Dynamic Programming which must consider all possible
  - Or it might be good enough (heuristic)
  - Might be able to prove later the choice is optimal
  - Might be able to prove that the choice is optimal
  - Subproblems - make a local choice
  - Choice how to divide into subproblems without considering all
  - Choice how to divide into subproblems without considering all
  - Divide into a single subproblem
  - Divide into a single subproblem
  - Like Dynamic Programming and divide and conquer
  - Like Dynamic Programming and divide and conquer
  - Have optimal substructure

Overview

• Huffman Codes
• Elements of the Greedy Strategy
• Activity Selection Problem
• Greedy Algorithms
• Enhanced Greedy Algorithms
Activity Selection Problem

• Set \( S = \{ a_1, a_2, ..., a_n \} \) proposed activities
  - Each activity has a start and end time \( 0 \leq s_i < f_i < \infty \)
  - Takes place during half-open time interval \([s_i, f_i)\)

What is the maximum number of events that can be scheduled in a lecture?

Select a maximum sized subset of mutually compatible activities

Arrange events by finish time

\[
\begin{array}{ccccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\hline
s_i & 1 & 3 & 0 & 5 & 3 & 5 & 6 & 8 & 10 & 11 & \\
f_i & 4 & 5 & 6 & 7 & 9 & 9 & 10 & 11 & 12 & 14 & 16 \\
\end{array}
\]

Set \( S \) = \{ \{ s_1, f_1 \}, \ldots, \{ s_n, f_n \} \} = A

Overview

• Greedy Algorithms
  ⇒ Activity Selection Problem
  • Elements of the Greedy Strategy
  • Huffman Codes

Greedy Algorithms
Framing Problem to Find Optimal Substructure

• Key insight
- Not all time points are interesting
- Only time points where an activity ends or starts is of interest

- Define \( S_{ij} = \{ a_k \in S | f_i \leq s_k \text{ and } f_k \leq s_j \} \)
- activities that start after \( a_i \) finishes and that end before \( a_j \) starts

- Optimal Substructure:
  - Suppose \( A_{ij} \) is a maximum set of mutually compatible activities in \( S_{ij} \)
  - Suppose that \( a_k \in A_{ij} \)
  - Can now divide the problem into finding max set of mutually compatible
- Easy to verify optimal substructure
- Can now divide the problem into finding max set of mutually compatible

Continued
Greedy Choice

- Pick the first one to finish. Can we just use local information to pick an activity?
- Hopefully guarantee that the activity is part of an optimal solution.
- Now just one problem left to solve, rather than two.
- All activities that end after $a_1^i$ are still eligible for inclusion.
- Pick one that leaves open as many other activities as possible.

Greedy Choice

- Define $c[i, j]$ as size of optimal solution for $S_{ij}$.
  
  $c[i, j] = \begin{cases} 
  0 & \text{if } S_{ij} = \emptyset \\
  \max_{a_k \in S_{i,j}} (c[i, k] + c[k, j] + 1) & \text{otherwise}
  \end{cases}$

Dynamic Programming:

- Bottom-up algorithm ordered by distance between $i$ and $j$.
- Top-down recursive algorithm with memoization.
- Define $c[i, j]$ as size of optimal solution for $S_{ij}$.
Recursive Algorithm
• Initially called with array s, f, start point 0, end point n

```
RECURSIVE-ACTIVITY-SELECTION (s, f, k, n)
1 m = k
2 while m ≤ n and s[m] < f[k]
3 return [f[k] > [s[m] and s[m] and s[m] + γ = w]
4 i = w
5 return
```

Running time?

Proof that Greedy Choice is Optimal
• Is this optimal?
  - Say that a1 is not part of any optimal solution
  - Must have finished after a1
  - If more than one, one must finish before a1 finished, contradiction
  - What activities are in optimal solution that are not compatible with a1?
  - Can replace it by a1 giving set of same size of compatible activities:

Conclusion
• Is this optimal?
Overview

Greedy Algorithms

• Greedy Algorithms

Elements of the Greedy Strategy

• Huffman Codes

Activity Selection Problem

Greedy Algorithms

Iterative Algorithm

Greedy-Activity-Selector (s, s', length)

1. for m = 2 to n
2. {m} = s
3. {m} = V
4. s'.length = u
5. if [s] f [m] s
6. \{m\} \cap A = A
7. w = \gamma
8. return A
More Directly

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.
4. Underneath every greedy algorithm is a more cumbersome DP solution.

Steps

1. Determine the optimal substructure of the problem.
2. Develop a recursive algorithm that implements the greedy strategy. (Steps 3 and 4 can occur in either order.)
3. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur on this recurrence.)
4. Develop a recursive solution. (For the activity-selection problem, we form the recurrence (16.2), but we bypass developing an recursive algorithm based on this recurrence.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.
Contrast: Knapsack Problem

Version 1: Must take it or leave it
Version 2: Can take fractional amounts

Examples:
- Item 1: Weighs 30 pounds worth $120
- Item 2: Weighs 20 pounds worth $100
- Item 3: Weighs 10 pounds worth $60

A thief robbing a store finds $n$ items. The $i$th item is worth $v_i$ dollars and weighs $w_i$ pounds, where $v_i$ and $w_i$ are integers. The thief can just carry $W$ pounds. What items should he take?

Example:

Version 1: Must take it or leave it
Version 2: Can take fractional amounts

Greedy Choice Property

- In DP, choice depends on solutions to subproblems
- Usually solve DP bottom-up
- Can often put choices into some order to allow very efficient choice
- Then solve the remaining problem
- Greedy choice that seems best (hopefully it is best)
- Can often pull choices into some order to allow very efficient choice
- Top-down approach
- Greedy Choice Property

CS533 Class 07: 16
P. Heeman, 2017
Overview

• Greedy Algorithms
• Activity Selection Problem
• Elements of the Greedy Strategy
  ⇒ Huffman Codes

Binary Knapsack Problem

• If you have an optimal solution for weight \( W \), and you take out an item from optimal solution, say \( i \) that weighs \( w_i \), the remaining items are optimal solution for weight \( W - w_i \) excluding \( i \).
• Can consider items in order.
  - So exhibits optimal substructure.
  - Remaining items are optimal solution for weight \( W - w_i \) excluding \( i \).
  - And you take out an item from optimal solution, say \( i \) that weighs \( w_i \).

Running Time?

- For polynomial, how do we order the subproblems?
- \( \frac{1}{n} \times \left( \lceil \frac{n}{m} \rceil \times \left( \frac{1}{2} + \left( \frac{1}{2n} \right) \right) \right) \) or \( \left( 1 + \frac{1}{m} \right) \times \frac{1}{2} \), \( m \) or \( \frac{1}{2} \) otherwise
- \( \frac{1}{n} \times \left( \lceil \frac{n}{m} \rceil \times \left( \frac{1}{2} + \left( \frac{1}{2n} \right) \right) \right) \) or \( \left( 1 + \frac{1}{m} \right) \times \frac{1}{2} \)

- Can consider items in order.
- So exhibits optimal substructure.
- Remaining items are optimal solution for weight \( W - w_i \) excluding \( i \).
- And you take out an item from optimal solution, say \( i \) that weighs \( w_i \).
Encoding Data

• Suppose we have a 100,000 character data file we wish to store compactly.

• There are only 6 different characters, and we can store characters in as many bits as we want.

• Each character needs to be prefixed-free.

• Each character can use a different number of bits.

• Fixed Length Code:

  - How many bits are needed for 6 characters?

  - Each character encoded in same number of bits.

  - There are only 6 different characters, and we can store compactly.

  - Suppose we have a 100,000 character data file we wish to store.
To prove Huffman algorithm is correct

- We show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal substructure properties.

**Optimal Substructure:**

Towards defining optimal prefix free for the code with the lowest frequency characters replaced by a single one

The optimal prefix free for a code is related to the optimal prefix free for the code with character replacement.

Huffman algorithm

```
HUFFMAN
1 n
2 Q
3 for i = 1 to n
4 allocate a new node x
5 y = EXTRACT-MIN(Q)
6 z = EXTRACT-MIN(Q)
7 b = freq(x) + freq(y)
8 c = freq(z)
9 allocate a new node z
10 j = i - u + 1
11 c[j] = z
12 |c| = |u| + 1
13 return EXTRACT-MIN(Q)
```

Cost of a binary tree over alphabet \( C \) is

\[
\sum_{c \in C} c.freq \times d_T(c)
\]

where \( d_T(c) \) is the depth of leaf with character \( c \) in tree \( T \).
Lemma 16.2

Claim: There is an optimal prefix tree in which the lowest two frequency characters are siblings at maximum depth.

Proof:
Let \( x \) and \( y \) have the lowest frequencies such that \( x.\text{freq} \leq y.\text{freq} \).
Let \( T \) be an optimal prefix tree, with \( a \) and \( b \) siblings of deepest node.

Case 1: \( x.\text{freq} = y.\text{freq} = a.\text{freq} = b.\text{freq} \)

Trivially true as \( T \) has lowest two frequency characters \( a \) and \( b \) at max depth.

Case 2: \( x.\text{freq} \leq y.\text{freq} \leq a.\text{freq} \)

So \( x.\text{freq} \leq a.\text{freq} \) and \( y.\text{freq} \leq a.\text{freq} \)

Let \( x \) and \( y \) have the lowest frequencies such that \( x.\text{freq} \leq y.\text{freq} \).

Preliminaries

There are at least two nodes at maximum depth that are siblings.

Every non-trivial code will have an optimal prefix tree in which:

- a leaf of both children.
- in an optimal prefix tree, every node will either have no children (nodes with no children) and not on internal nodes.
- a binary tree gives a prefix tree in which characters are on leaves.

Preliminaries
Preface to Lemma 16.3

Let \(C_{xy}\) be a given alphabet with frequency \(c.freq\) for each character \(c \in C\) where \(x\) and \(y\) be the least two frequent characters in \(C\).

Let \(C_z\) be same as \(C_{xy}\), but with \(x\) and \(y\) replaced by \(z\) and \(z\)'s frequency sum of \(x\) and \(y\). Let \(T_z\) be an optimal prefix tree for \(D\).

Create \(T_{xy}\) by replacing \(z\)’s leaf by a node with \(x\) and \(y\) as its children.

\[ B(T_z) = B(T_{xy}) - (x.freq + y.freq) \]

Claim: \(T_{xy}\) is an optimal prefix tree for \(C_{xy}\).

Let \(C\) be same as \(C_{xy}\), but \(x\) and \(y\) be the least two frequent characters in \(C\).

Let \(C\) be a given alphabet with frequency \(c.freq\) for each character \(c \in C\).

Case 16.3: 

So must have same costs so is also optimal

Since \(L\) is optimal \(\Rightarrow (nL)D \leq (mL)D\)

Similarly \(\Rightarrow (mL)D \geq (nL)D\) and \(n.freq \geq m.freq\)

So \(L\) is also optimal

Case 2: 

Since \(x, y, q \neq x\) and \(h\) will be siblings in \(J\),

Create \(L\) by swapping \(q\) and \(x\)

Create \(L\) by swapping \(q\) and \(x\)

\(q \neq x\)
Efficiency of Code

How much time does extract min and insert take?

- Could take \( O(n) \) using a simple array of freq.
- With binary search tree?
- Do we need all of the functionality of binary search tree?

```c
allocate a new node

\( b \) \( x \) + \( b \) \( y \) = \( b \) \( z \)

\( \text{EXTRACT-MIN}(Q) \)

\( \text{INSERT}(Q) \)

return the root of the tree
```

Lemma 16.3

Claim: \( T_{xy} \) is an optimal prefix tree for \( C_{xy} \).

Proof by contradiction:

Assume that \( T_{xy} \) is not optimal. Let \( T'_{xy} \) be an optimal prefix tree for \( C_{xy} \). For \( i = 1 \text{ to } n \)

\( \forall \) \( x \) \( \text{ and } y \) under the same node

Construct \( T'_{z} \) from \( T'_{xy} \) by replacing \( x \) and \( y \) as above:

Due to Lemma 16.2, WLG, assume that \( x \) and \( y \) under the same node

\( \text{EXTRACT-MIN}(Q) = (\frac{z}{\mu}L)Q \)

So \( \text{EXTRACT-MIN}(Q) \) is an optimal prefix for \( C_{xy} \).

Assume that \( T_{xy} \) is not optimal.

Claim: \( T_{xy} \) is an optimal prefix for \( C_{xy} \).

Proof by contradiction:

Due to Lemma 16.2, WLG, assume that \( x \) and \( y \) under the same node

\( \text{EXTRACT-MIN}(Q) = (\frac{z}{\mu}L)Q \)

So \( \text{EXTRACT-MIN}(Q) \) is an optimal prefix for \( C_{xy} \).

Assume that \( T_{xy} \) is not optimal.