CSE 548: Algorithms

Greedy Algorithms

R. Sekar

Overview

- One of the strategies used to solve *optimization problems*
  - Multiple solutions exist; pick one of low (or least) cost
- *Greedy strategy:* make a locally optimal choice, or simply, what appears best at the moment

Overview

- One of the strategies used to solve *optimization problems*
  - Multiple solutions exist; pick one of low (or least) cost
- *Greedy strategy:* make a locally optimal choice, or simply, what appears best at the moment
- Often, *locally optimality* $\not\Rightarrow$ *global optimality*
- So, use with a great deal of care
  - *Always need to prove optimality*

Overview

- One of the strategies used to solve *optimization problems*
  - Multiple solutions exist; pick one of low (or least) cost
- *Greedy strategy:* make a locally optimal choice, or simply, what appears best at the moment
- Often, *locally optimality* $\not\Rightarrow$ *global optimality*
- So, use with a great deal of care
  - *Always need to prove optimality*
- If it is unpredictable, why use it?
  - *It simplifies the task!*

Making change

Given coins of denominations 25¢, 10¢, 5¢ and 1¢, make change for x cents ($0 < x < 100$) using *minimum number of coins.*

Greedy solution

makeChange(x)

if ($x = 0$) **return**

Let y be the largest denomination that satisfies $y \leq x$

Issue $\lfloor x/y \rfloor$ coins of denomination y

makeChange($x \mod y$)

Making change

Given coins of denominations 25¢, 10¢, 5¢ and 1¢, make change for x cents ($0 < x < 100$) using *minimum number of coins.*

Greedy solution

makeChange(x)
if ($x = 0$) **return**
Let y be the largest denomination that satisfies $y \leq x$
Issue $\lfloor x/y \rfloor$ coins of denomination y
makeChange($x \mod y$)

- Show that it is optimal

Making change

Given coins of denominations 25¢, 10¢, 5¢ and 1¢, make change for x cents $(0 < x < 100)$ using *minimum number of coins.*

Greedy solution

makeChange(x)

if $(x = 0)$ **return**

Let y be the largest denomination that satisfies $y \leq x$

Issue $\lfloor x/y \rfloor$ coins of denomination y

makeChange$(x \mod y)$

- Show that it is optimal
- Is it optimal for arbitrary denominations?

When does a Greedy algorithm work?

Greedy choice property

The greedy (i.e., locally optimal) choice is always consistent with some (globally) optimal solution

When does a Greedy algorithm work?

Greedy choice property

The greedy (i.e., locally optimal) choice is always consistent with some (globally) optimal solution

What does this mean for the coin change problem?

When does a Greedy algorithm work?

Greedy choice property

The greedy (i.e., locally optimal) choice is always consistent with some (globally) optimal solution

What does this mean for the coin change problem?

Optimal substructure

The optimal solution contains optimal solutions to subproblems.

When does a Greedy algorithm work?

Greedy choice property

The greedy (i.e., locally optimal) choice is always consistent with some (globally) optimal solution

What does this mean for the coin change problem?

Optimal substructure

The optimal solution contains optimal solutions to subproblems.

Implies that a greedy algorithm can invoke itself recursively after making a greedy choice.

Knapsack Problem

- A sack that can hold a maximum of x lbs
- You have a choice of items you can pack in the sack
- Maximize the combined "value" of items in the sack

item	calories/lb	weight
bread	1100	5
butter	3300	1
tomato	80	1
cucumber	55	2

Knapsack Problem

- A sack that can hold a maximum of x lbs
- You have a choice of items you can pack in the sack
- Maximize the combined "value" of items in the sack

item	calories/lb	weight
bread	1100	5
butter	3300	1
tomato	80	1
cucumber	55	2

0-1 knapsack: Take all of one item or none at all

Fractional knapsack: Fractional quantities acceptable

Knapsack Problem

- A sack that can hold a maximum of *x* lbs
- You have a choice of items you can pack in the sack
- Maximize the combined “value” of items in the sack

item	calories/lb	weight
bread	1100	5
butter	3300	1
tomato	80	1
cucumber	55	2

0-1 knapsack: Take all of one item or none at all

Fractional knapsack: Fractional quantities acceptable

Greedy choice: pick item that maximizes calories/lb

Knapsack Problem

- A sack that can hold a maximum of x lbs
- You have a choice of items you can pack in the sack
- Maximize the combined "value" of items in the sack

item	calories/lb	weight
bread	1100	5
butter	3300	1
tomato	80	1
cucumber	55	2

0-1 knapsack: Take all of one item or none at all

Fractional knapsack: Fractional quantities acceptable

Greedy choice: pick item that maximizes calories/lb

Will a greedy algorithm work, with $x = 5$?

Fractional Knapsack

Greedy choice property

Proof by contradiction: Start with the assumption that there is an optimal solution that does not include the greedy choice, and show a contradiction.

Fractional Knapsack

Greedy choice property

Proof by contradiction: Start with the assumption that there is an optimal solution that does not include the greedy choice, and show a contradiction.

Optimal substructure

After taking as much of the item with jth maximal value/weight, suppose that the knapsack can hold y more lbs.

Then the optimal solution for the problem includes the optimal choice of how to fill a knapsack of size y with the remaining items.

Fractional Knapsack

Greedy choice property

Proof by contradiction: Start with the assumption that there is an optimal solution that does not include the greedy choice, and show a contradiction.

Optimal substructure

After taking as much of the item with jth maximal value/weight, suppose that the knapsack can hold y more lbs.

Then the optimal solution for the problem includes the optimal choice of how to fill a knapsack of size y with the remaining items.

Does not work for 0-1 knapsack because greedy choice property does not hold.

0-1 knapsack is NP-hard, but a pseudo-polynomial algorithm is available.

Spanning Tree

A subgraph of a graph $G = (V, E)$ that includes:

- All the vertices V in the graph
- A subset of E such that these edges form a tree

Spanning Tree

A subgraph of a graph $G = (V, E)$ that includes:

- All the vertices V in the graph
- A subset of E such that these edges form a tree

We consider *connected undirected graphs*, where the second condition for MST can be replaced by

Spanning Tree

A subgraph of a graph $G = (V, E)$ that includes:

- All the vertices V in the graph
- A subset of E such that these edges form a tree

We consider *connected undirected graphs*, where the second condition for MST can be replaced by

- A maximal subset of E such that the subgraph has no cycles

Spanning Tree

A subgraph of a graph $G = (V, E)$ that includes:

- All the vertices V in the graph
- A subset of E such that these edges form a tree

We consider *connected undirected graphs*, where the second condition for MST can be replaced by

- A maximal subset of E such that the subgraph has no cycles
- A subset of E with $|V| - 1$ edges such that the subgraph is connected

Spanning Tree

A subgraph of a graph $G = (V, E)$ that includes:

- All the vertices V in the graph
- A subset of E such that these edges form a tree

We consider *connected undirected graphs*, where the second condition for MST can be replaced by

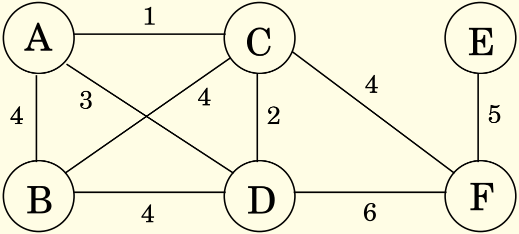
- A maximal subset of E such that the subgraph has no cycles
- A subset of E with $|V| - 1$ edges such that the subgraph is connected
- A subset of E such that there is a unique path between any two vertices in the subgraph

Minimal Spanning Tree (MST)

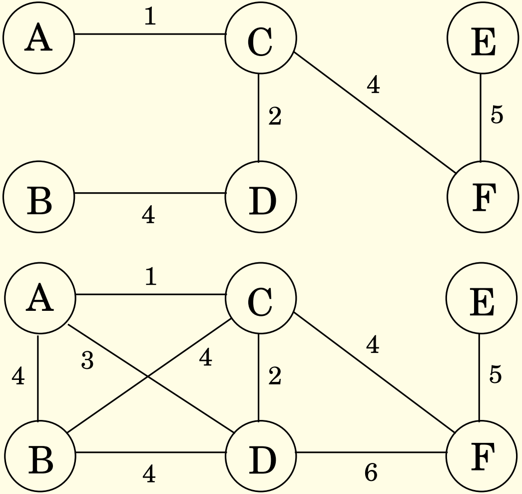
A spanning tree with *minimal cost.* Formally:

Input: An undirected graph $G = (V, E)$, a cost function $w : E \to \mathbb{R}$.

Output: A tree $T = (V, E')$ such that $E' \subseteq E$ that minimizes $\sum_{e \in E'} w(e)$



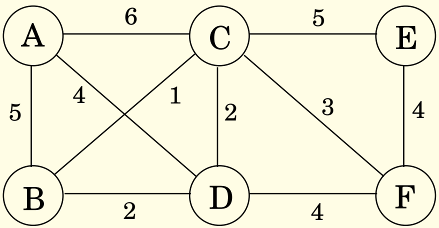
Minimal Spanning Tree (MST)

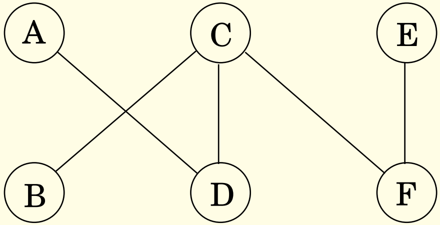


Kruskal's algorithm

- Start with the empty set of edges
- Repeat: add lightest edge that doesn't create a cycle

Adds edges B—C, C—D, C—F, A—D, E—F





Kruskal's algorithm

MST(V, E, w)

$X = \phi$

$Q = priorityQueue(E)$ // from min to max weight

while Q is nonempty

$e = deleteMin(Q)$

if e connects two disconnected components in (V, X) $X = X \cup \{e\}$

Kruskal's: Correctness (by induction)

Induction Hypothesis: The first i edges selected by Kruskal's are included in some MST T

Kruskal's: Correctness (by induction)

Induction Hypothesis: The first i edges selected by Kruskal's are included in some MST T

Base case: trivial — the empty set of edges is always in any MST.

Kruskal's: Correctness (by induction)

Induction Hypothesis: The first i edges selected by Kruskal's are included in some MST T

Base case: trivial — the empty set of edges is always in any MST.

Induction step: Show that $i+1$th edge chosen by Kruskal's is in the MST T

Proof: Let $e = (v, w)$ be the edge chosen at $i+1$th step of Kruskal's.

- *Case 1:* $e \in T$: The induction step is done.

Kruskal's: Correctness (by induction)

Induction Hypothesis: The first i edges selected by Kruskal's are included in some MST T

Base case: trivial — the empty set of edges is always in any MST.

Induction step: Show that $i+1$th edge chosen by Kruskal's is in the MST T

Proof: Let $e = (v, w)$ be the edge chosen at $i + 1$th step of Kruskal's.

- *Case 1:* $e \in T$: The induction step is done.
- *Case 2:* $e \notin T$: T is a spanning tree: must include a unique path from v to w

Kruskal's: Correctness (by induction)

Induction Hypothesis: The first i edges selected by Kruskal's are included in some MST T

Base case: trivial — the empty set of edges is always in any MST.

Induction step: Show that $i+1$th edge chosen by Kruskal's is in the MST T

Proof: Let $e = (v, w)$ be the edge chosen at $i + 1$th step of Kruskal's.

- *Case 1:* $e \in T$: The induction step is done.
- *Case 2:* $e \notin T$: T is a spanning tree: must include a unique path from v to w
  - At least one edge e' on this path is not in X, the set of edges chosen in the first i steps by Kruskal's. (Otherwise, v and w will be connected in X, so Kruskal's won't chose e.)

Kruskal's: Correctness (by induction)

Induction Hypothesis: The first i edges selected by Kruskal's are included in some MST T

Base case: trivial — the empty set of edges is always in any MST.

Induction step: Show that $i+1$th edge chosen by Kruskal's is in the MST T

Proof: Let $e = (v, w)$ be the edge chosen at $i + 1$th step of Kruskal's.

- *Case 1:* $e \in T$: The induction step is done.
- *Case 2:* $e \notin T$: T is a spanning tree: must include a unique path from v to w
  - At least one edge e' on this path is not in X, the set of edges chosen in the first i steps by Kruskal's. (Otherwise, v and w will be connected in X, so Kruskal's won't chose e.)
  - Since neither e nor e' are in X, and Kruskal's chose e, $w(e') \geq w(e)$.

Kruskal's: Correctness (by induction)

Induction Hypothesis: The first i edges selected by Kruskal's are included in some MST T

Base case: trivial — the empty set of edges is always in any MST.

Induction step: Show that $i+1$th edge chosen by Kruskal's is in the MST T

Proof: Let $e = (v, w)$ be the edge chosen at $i + 1$th step of Kruskal's.

- *Case 1:* $e \in T$: The induction step is done.
- *Case 2:* $e \notin T$: T is a spanning tree: must include a unique path from v to w
  - At least one edge e' on this path is not in X, the set of edges chosen in the first i steps by Kruskal's. (Otherwise, v and w will be connected in X, so Kruskal's won't chose e.)
  - Since neither e nor e' are in X, and Kruskal's chose e, $w(e') \geq w(e)$.
  - Replace e' by e in T to get another spanning tree T'. Either $w(T') < w(T)$, a contradiction to the assumption T is minimal; or $w(T') = w(T)$, and we have another MST T' consistent with $X \cup \{e\}$. In both cases, we have completed the induction step. ■

Kruskal's: Runtime complexity

$MST(V, E, w)$

$X = \phi$

$Q = priorityQueue(E, w)$ // from min to max weight

while Q is nonempty

  $e = deleteMin(Q)$

  if e connects two disconnected components in (V, X) $X = X \cup \{e\}$

- Priority queue: $O(\log |E|) = O(\log V)$ per operation
- Connectivity test: $O(\log V)$ per check using a disjoint set data structure

Thus, for $|E|$ iterations, we have a runtime of $O(|E| \log |V|)$

MST: Applications

Network design: Communication networks, transportation networks, electrical grid, oil/water pipelines, ...

Clustering: Application of minimum spanning forest (stop when $|X| = |V| - k$ to get k clusters)

Broadcasting: Spanning tree protocol in Ethernets

Information Theory and Coding

Information content

For an event e that occurs with probability p, its information content is given by

$I(e) = -\log p$

Information Theory and Coding

Information content

For an event e that occurs with probability p, its information content is given by
$I(e) = -\log p$

- "surprise factor" — low probability event conveys more information; an event that is almost always likely ($p \approx 1$) conveys no information.
- Information content adds up: for two events e_1 and e_2, their combined information content is $-(\log p_1 + \log p_2)$

Information theory: Entropy

Information entropy

For a discrete random variable X that can take a value x_i with probability p_i, its entropy is defined as the *expectation* ("weighted average") over the information content of x_i:

$$H(X) = E[I(X)] = -\sum_{i=1}^{n} p_i \log p_i$$

Information theory: Entropy

Information entropy

For a discrete random variable X that can take a value x_i with probability p_i, its entropy is defined as the *expectation* ("weighted average") over the information content of x_i:

$$H(X) = E[I(X)] = -\sum_{i=1}^{n} p_i \log p_i$$

- Entropy is a measure of uncertainty
- Plays a fundamental role in many areas, including coding theory and machine learning.

Optimal code length

Shannon's source coding theorem

A random variable X denoting chars in an alphabet $\Sigma = \{x_1, \ldots, x_n\}$

- cannot be encoded in fewer than $H(X)$ bits.
- can be encoded using at most $H(X) + 1$ bits

Optimal code length

Shannon's source coding theorem

A random variable X denoting chars in an alphabet $\Sigma = \{x_1, \ldots, x_n\}$

- cannot be encoded in fewer than $H(X)$ bits.
- can be encoded using at most $H(X) + 1$ bits

- The first part of this theorem sets a lower bound, regardless of how clever the encoding is.

Optimal code length

Shannon's source coding theorem

A random variable X denoting chars in an alphabet $\Sigma = \{x_1, \ldots, x_n\}$

- cannot be encoded in fewer than $H(X)$ bits.
- can be encoded using at most $H(X) + 1$ bits

- The first part of this theorem sets a lower bound, regardless of how clever the encoding is.
- Surprisingly simple proof for such a fundamental theorem! (See Wikipedia.)

Optimal code length

Shannon's source coding theorem

A random variable X denoting chars in an alphabet $\Sigma = \{x_1, \ldots, x_n\}$

- cannot be encoded in fewer than $H(X)$ bits.
- can be encoded using at most $H(X) + 1$ bits

- The first part of this theorem sets a lower bound, regardless of how clever the encoding is.
- Surprisingly simple proof for such a fundamental theorem! (See Wikipedia.)
- Huffman coding: an algorithm that achieves this bound

Variable-length encoding

Let $\Sigma = \{A, B, C, D\}$ with probabilities $0.55, 0.02, 0.15, 0.28$.

- If we use a fixed-length code, each character will use 2-bits.
- Alternatively, use a variable length code

Variable-length encoding

Let $\Sigma = \{A, B, C, D\}$ with probabilities $0.55, 0.02, 0.15, 0.28$.

- If we use a fixed-length code, each character will use 2-bits.
- Alternatively, use a variable length code
  - Let us use as many bits as the *information content* of a character
  - A uses 1 bit, B uses 6 bits, C uses 3 bits, and D uses 2 bits.
  - You get an average saving of 15%

    $0.55 * 1 + 0.02 * 6 + 0.15 * 3 + 0.28 * 2 = 1.68$ bits

Variable-length encoding

Let $\Sigma = \{A, B, C, D\}$ with probabilities $0.55, 0.02, 0.15, 0.28$.

- If we use a fixed-length code, each character will use 2-bits.
- Alternatively, use a variable length code
  - Let us use as many bits as the *information content* of a character
  - A uses 1 bit, B uses 6 bits, C uses 3 bits, and D uses 2 bits.
  - You get an average saving of 15%

    $$0.55 * 1 + 0.02 * 6 + 0.15 * 3 + 0.28 * 2 = 1.68 \text{ bits}$$

  - Lower bound (entropy)

    $$-(.5 \log_2 .5 + .02 \log_2 .02 + .14 \log_2 .14 + .27 \log_2 .27) = 1.51 \text{ bits}$$

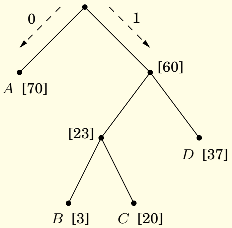
Variable-length encoding

Let $\Sigma = \{A, B, C, D\}$ with probabilities 0.55, 0.02, 0.15, 0.28.

- Let us try fixing the codes, not just their lengths:

  $A = 0, D = 11, C = 101, B = 100.$

- Note: enough to assign 3 bits to B, not 6. So, average coding size reduces to 1.62.



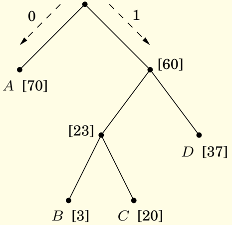
Variable-length encoding

Let $\Sigma = \{A, B, C, D\}$ with probabilities $0.55, 0.02, 0.15, 0.28$.

- Let us try fixing the codes, not just their lengths:

  $A = 0, D = 11, C = 101, B = 100.$

- Note: enough to assign 3 bits to B, not 6. So, average coding size reduces to 1.62.

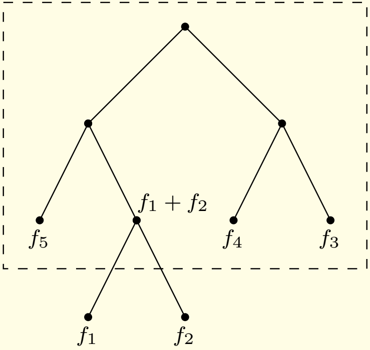


Prefix encoding

- No code is a prefix of another.
- Necessary property to enable decoding.
- Every such encoding can be represented using a full binary tree (either 0 or 2 children for every node)

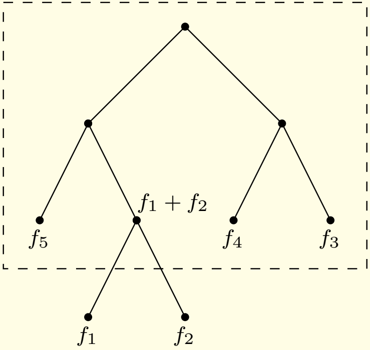
Huffman encoding

- Build the prefix tree bottom-up
- Start with a node whose children are codewords c_1 and c_2 that occur least often
- Remove c_1 and c_2 from alphabet, replace with c' that occurs with frequency $f_1 + f_2$
- Recurse



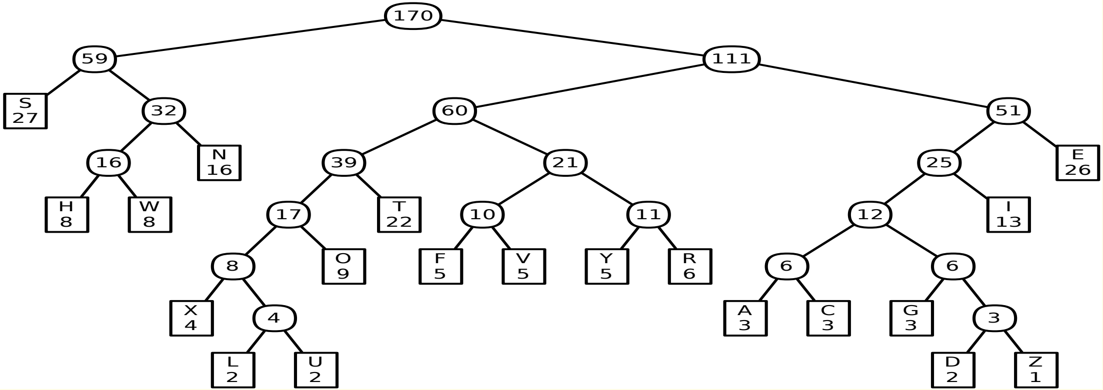
Huffman encoding

- Build the prefix tree bottom-up
- Start with a node whose children are codewords c_1 and c_2 that occur least often
- Remove c_1 and c_2 from alphabet, replace with c' that occurs with frequency $f_1 + f_2$
- Recurse



- How to make this algorithm fast?
- What is its complexity?

Huffman encoding: 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. Images from Jeff Erickson's "Algorithms"

Uses about 650 bits, vs 850 for fixed-length (5-bit) code.

Huffman encoding: Optimality

- Crux of the proof: *Greedy choice property*
- Familiar exchange argument

Huffman encoding: Optimality

- Crux of the proof: *Greedy choice property*
- Familiar exchange argument
  - Suppose the optimal prefix tree does not use longest path for two least frequent codewords c_1 and c_2
  - Show that by exchanging c_1 with the codeword using the longest path in the optimal tree, you can reduce the cost of the "optimal code" — a contradiction

Huffman encoding: Optimality

- Crux of the proof: *Greedy choice property*
- Familiar exchange argument
  - Suppose the optimal prefix tree does not use longest path for two least frequent codewords c_1 and c_2
  - Show that by exchanging c_1 with the codeword using the longest path in the optimal tree, you can reduce the cost of the "optimal code" — a contradiction
  - Same argument holds for c_2

Huffman Coding: Applications

- Document compression
- Signal encoding
- As part of other compression algorithms (MP3, gzip, PKZIP, JPEG, ...)

Lossless Compression

- How much compression can we get using Huffman?

Lossless Compression

- How much compression can we get using Huffman?
  - It depends on what we mean by a codeword!

Lossless Compression

- How much compression can we get using Huffman?
  - It depends on what we mean by a codeword!
    - If they are English characters, effect is relatively small
    - if they are English words, or better, sentences, then much higher compression is possible

Lossless Compression

- How much compression can we get using Huffman?
  - It depends on what we mean by a codeword!
    - If they are English characters, effect is relatively small
    - if they are English words, or better, sentences, then much higher compression is possible
- To use words/sentences as codewords, we probably need to construct document-specific codebook
  - Larger alphabet size implies larger codebooks!
  - Need to consider the combined size of codebook plus the encoded document

Lossless Compression

- How much compression can we get using Huffman?
  - It depends on what we mean by a codeword!
    - If they are English characters, effect is relatively small
    - if they are English words, or better, sentences, then much higher compression is possible
- To use words/sentences as codewords, we probably need to construct document-specific codebook
  - Larger alphabet size implies larger codebooks!
  - Need to consider the combined size of codebook plus the encoded document
- Can the codebook be constructed on-the-fly?
  - Lempel-Ziv compression algorithms (gzip)

gzip Algorithm [Lempel-Ziv 1977]

Key Idea: Use preceding W-bytes as the codebook ("sliding window", up to 32KB in gzip)

gzip Algorithm [Lempel-Ziv 1977]

Key Idea: Use preceding W-bytes as the codebook ("sliding window", up to 32KB in gzip)

Encoding:

- Strings previously seen in the window are replaced by the pair (*offset*, *length*)
  - Need to find the longest match for the current string
  - Matches should have a minimum length, or else they will be emitted as literals
  - Encode offset and length using Huffman encoding

gzip Algorithm [Lempel-Ziv 1977]

Key Idea: Use preceding W-bytes as the codebook ("sliding window", up to 32KB in gzip)

Encoding:

- Strings previously seen in the window are replaced by the pair (*offset*, *length*)
    - Need to find the longest match for the current string
    - Matches should have a minimum length, or else they will be emitted as literals
    - Encode offset and length using Huffman encoding

Decoding: Interpret (*offset*, *length*) using the same window of W-bytes of preceding text. (Much faster than encoding.)

Greedy Algorithms: Summary

- One of the strategies used to solve *optimization problems*
- Frequently, locally optimal choices are *NOT* globally optimal, so use with a great deal of care.
  - *Always need to prove optimality.* Proof typically relies on *greedy choice property*, usually established by an "exchange" argument, and *optimal substructure.*

Greedy Algorithms: Summary

- One of the strategies used to solve *optimization problems*
- Frequently, locally optimal choices are *NOT* globally optimal, so use with a great deal of care.
  - *Always need to prove optimality.* Proof typically relies on *greedy choice property*, usually established by an "exchange" argument, and *optimal substructure*.

Greedy Algorithms: Summary

- One of the strategies used to solve *optimization problems*
- Frequently, locally optimal choices are *NOT* globally optimal, so use with a great deal of care.
  - *Always need to prove optimality.* Proof typically relies on *greedy choice property*, usually established by an "exchange" argument, and *optimal substructure.*
- Examples
  - MST and clustering
  - Shortest path
  - Huffman encoding