COMP20007 Design of Algorithms

Data Compression

Daniel Beck

Lecture 16

Semester 1, 2020

Introduction

- So far, we talked about speed and space performance from an algorithm point of view.

Introduction

- So far, we talked about speed and space performance from an algorithm point of view.
- We assumed that records could fit in memory. (although we did mention secondary memory in Mergesort and B-trees)

Hard drives

Introduction

- So far, we talked about speed and space performance from an algorithm point of view.
- We assumed that records could fit in memory. (although we did mention secondary memory in Mergesort and B-trees)
- What to do when records are *too large?* (videos, for instance)

Fixed-length encoding

For text files, suppose each character has an fixed-size binary code.

Fixed-length encoding

For text files, suppose each character has an fixed-size binary code.

B A G G E D

Fixed-length encoding

For text files, suppose each character has an fixed-size binary code.

B A G G E D

01000010 01000001 01000111 01000111 01000101 01000100

8 bits

Fixed-length encoding

For text files, suppose each character has an fixed-size binary code.

B A G G E D

01000010 01000001 01000111 01000111 01000101 01000100

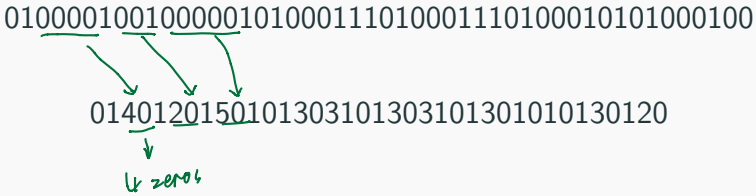
This is exactly what ASCII does.

Key insight: this coding has *redundant* information.

Run-length encoding

01000010010000010100011101000111010001010100010 0

Run-length encoding



Run-length encoding

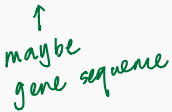
01000010010000010100011101000111010001010100100

01401201501013031013031013010101301 20

Character-level:

B A G G E D → B A 2 G E D

AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD



4A3BAA5B8CDABCB3A4B3CD

Run-length encoding

- While not very useful for text data, it can work for some kinds of binary data.

Run-length encoding

- While not very useful for text data, it can work for some kinds of binary data.
- For text, the best algorithms move away from using fixed-length codes (ASCII).

we don't need to waste 8 bit for every character

Variable-length Encoding

Variable-length Encoding

- **Key idea:** some symbols appear more *frequently* than others.

Variable-length Encoding

- **Key idea:** some symbols appear more *frequently* than others.
- Instead of a fixed number of bits per symbol, use a *variable* number:
  - More frequent symbols use less bits.
  - Less frequent symbols use more bits.

Variable-length Encoding

- **Key idea:** some symbols appear more *frequently* than others.
- Instead of a fixed number of bits per symbol, use a *variable* number:
  - More frequent symbols use less bits.
  - Less frequent symbols use more bits.
- For this scheme to work, no symbol code can be a prefix of another symbol's code.

eg. if we let E → 1
A → 10 or 00 ✗

Variable-Length Encoding

Suppose we count symbols and find these numbers of occurrences:

frequency

Symbol	Weight
B	4
D	5
G	10
F	12
C	14
E	27
A	28

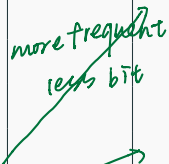
Variable-Length Encoding

Suppose we count symbols and find these numbers of occurrences:

Symbol	Weight
B	4
D	5
G	10
F	12
C	14
E	27
A	28

Here are some sensible codes that we may use for symbols:

Symbol	Code
A	11
B	0000
C	011
D	0001
E	10
F	010
G	001



Encoding a string

- Codes can be stored in a dictionary

Encoding a string

- Codes can be stored in a dictionary
- Once we have the codes, encoding is straightforward.

Encoding a string

- Codes can be stored in a dictionary
- Once we have the codes, encoding is straightforward.
- For example, to encode 'BAGGED', simply concatenate the codes for B, A, G, G, E and D:

A	11
B	0000
C	011
D	0001
E	10
F	010
G	001

0000|11|001|001|10|0001|

B A G G E D

→ shorter than ASCII

we can't let any symbol be prefix of others

Decoding a string

- To decode we can use another dictionary where keys are codes and values are symbols.

Decoding a string

- To decode we can use another dictionary where keys are codes and values are symbols.
- Starting from the first digit, look in the dictionary. If not present, concatenate the next digit and repeat until code is valid.

Decoding a string

- To decode we can use another dictionary where keys are codes and values are symbols.
- Starting from the first digit, look in the dictionary. If not present, concatenate the next digit and repeat until code is valid.

11	A
0000	B
011	C
0001	D
10	E
010	F
001	G

000011001001100001

look at digit
↓
not present
↓
include next digit and find

Decoding a string

- To decode we can use another dictionary where keys are codes and values are symbols.
- Starting from the first digit, look in the dictionary. If not present, concatenate the next digit and repeat until code is valid.

11	A
0000	B
011	C
0001	D
10	E
010	F
001	G

000011001001100001

Seems like it requires lots of misses, is there a better way?

if search 0 in the dictionary → miss
00 → miss
⋮
0000 → found !

Tries

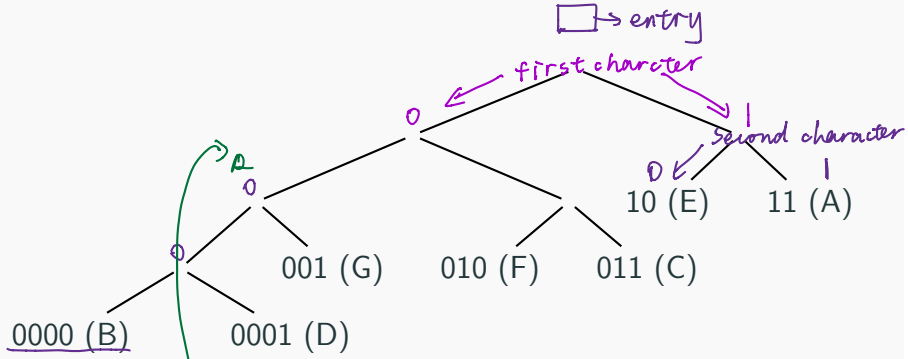
- Another implementation of a dictionary.

Tries

- Another implementation of a dictionary.
- Works when keys can be decomposed

Tries

- Another implementation of a dictionary.
- Works when keys can be decomposed



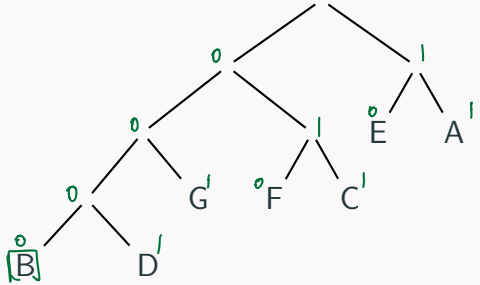
This specific trie stores values only in the leaves → keeps prefix property.

if store in one node, against the prefix property mismatch.

Tries

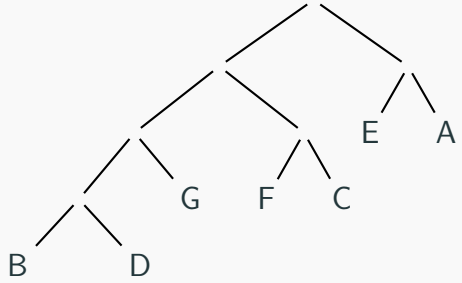
To decode 00001100100110100001, use the trie, repeatedly starting from the root, and printing each symbol found as a leaf.

BAGGED



Tries

To decode 000011001001100001, use the trie, repeatedly starting from the root, and printing each symbol found as a leaf.



How to choose the codes?

Huffman Encoding

Huffman Encoding

- Goal: obtain the *optimal* encoding given symbol frequencies.

shortest compression in bit

Huffman Encoding

- Goal: obtain the *optimal* encoding given symbol frequencies.
- Treat each symbol as a *leaf* and build a binary tree bottom-up.

Huffman Encoding

- Goal: obtain the *optimal* encoding given symbol frequencies.
- Treat each symbol as a *leaf* and build a binary tree bottom-up.
- Two nodes are *fused* if they have the *smallest* frequency.

join or blend to form a single entity

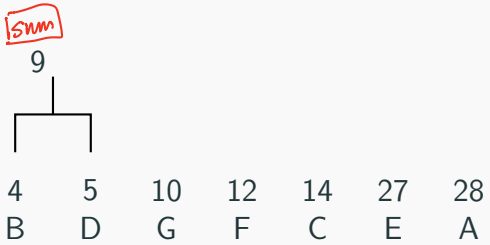
Huffman Encoding

- Goal: obtain the *optimal* encoding given symbol frequencies.
- Treat each symbol as a *leaf* and build a binary tree bottom-up.
- Two nodes are *fused* if they have the *smallest* frequency.
- The resulting tree is a Huffman tree.

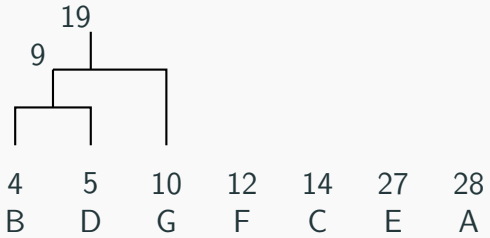
Huffman Trees

4	5	10	12	14	27	28
B	D	G	F	C	E	A

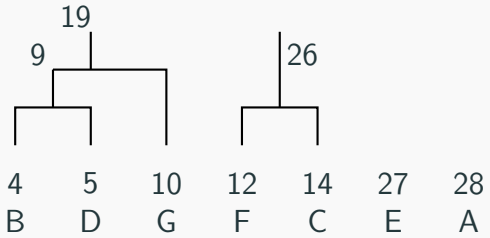
Huffman Trees



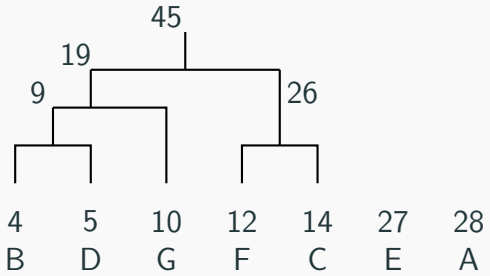
Huffman Trees



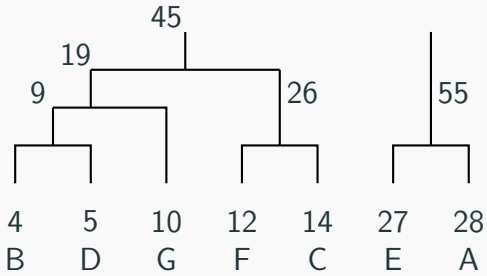
Huffman Trees



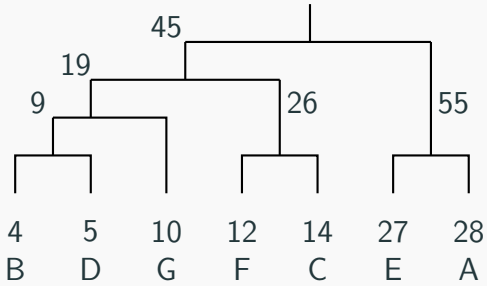
Huffman Trees



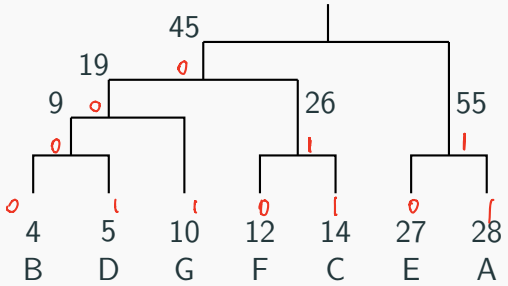
Huffman Trees



Huffman Trees



Huffman Trees



We end up with the trie from before!

E – 10

A – 11

B – 0000

The Greedy Method

Huffman is an example of the *greedy* method.

The Greedy Method

Huffman is an example of the *greedy* method.

- The goal is *optimisation*: many solutions are "acceptable" but we want to find the *best* one.

↓

give shortest compression

The Greedy Method

Huffman is an example of the *greedy* method.

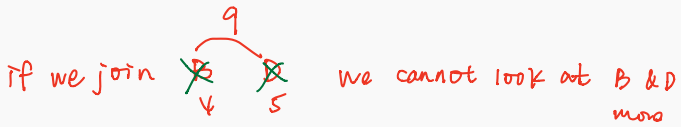
- The goal is *optimisation*: many solutions are "acceptable" but we want to find the *best* one.
- The problem can be divided into *local* subproblems.

eg. ASCII

The Greedy Method

Huffman is an example of the *greedy* method.

- The goal is *optimisation*: many solutions are "acceptable" but we want to find the *best* one.
- The problem can be divided into *local* subproblems.
- The greedy method builds a *global* solution through *local* and *irrevocable* solutions.



The Greedy Method

Huffman is an example of the *greedy* method.

- The goal is *optimisation*: many solutions are "acceptable" but we want to find the *best* one.
- The problem can be divided into *local* subproblems.
- The greedy method builds a *global* solution through *local* and *irrevocable* solutions.
- For Huffman, the greedy methods finds the optimal. Dijkstra and Prim are other examples.

The Greedy Method

Huffman is an example of the *greedy* method.

- The goal is *optimisation*: many solutions are "acceptable" but we want to find the *best* one.
- The problem can be divided into *local* subproblems.
- The greedy method builds a *global* solution through *local* and *irrevocable* solutions.
- For Huffman, the greedy methods finds the optimal. Dijkstra and Prim are other examples.
- But this is not always the case.

Summary

Summary

- Most data we store in our computer has *redundancy*.

Summary

- Most data we store in our computer has *redundancy*.
- Compression uses redundancy to reduce space.

Summary

- Most data we store in our computer has *redundancy*.
- Compression uses redundancy to reduce space.
- Huffman is based on variable-length encoding.

ASCII – fix length encoding

Summary

- Most data we store in our computer has *redundancy*.
- Compression uses redundancy to reduce space.
- Huffman is based on variable-length encoding.
- Tries to store codes.

↓ decompose keys

can be useful in any kind of symbols

Data Compression - In Practice

Data Compression - In Practice

- Huffman encoding provides the basis for many advanced compression techniques.

Data Compression - In Practice

- Huffman encoding provides the basis for many advanced compression techniques.
- Lempel-Ziv compression assigns codes to *sequences of symbols*: used in GIF, PNG and ZIP.

Huffman only assign code to characters

Data Compression - In Practice

- Huffman encoding provides the basis for many advanced compression techniques.
- Lempel-Ziv compression assigns codes to *sequences of symbols*: used in GIF, PNG and ZIP.
- For sequential data (audio/video), an alternative is linear prediction: *predict* the next frame given the previous ones. Used in FLAC.

eg. audio / video frame

lossless compression

we can recover the data with 100% accuracy

Data Compression - In Practice

- Huffman encoding provides the basis for many advanced compression techniques.
- Lempel-Ziv compression assigns codes to *sequences of symbols*: used in GIF, PNG and ZIP.
- For sequential data (audio/video), an alternative is linear prediction: *predict* the next frame given the previous ones. Used in FLAC.
- **Lossy compression:** JPEG, MP3, MPEG and others. Also employ Huffman (among other techniques).

lose information

Data Compression - In Practice

- Huffman encoding provides the basis for many advanced compression techniques.
- Lempel-Ziv compression assigns codes to *sequences of symbols*: used in GIF, PNG and ZIP.
- For sequential data (audio/video), an alternative is linear prediction: *predict* the next frame given the previous ones. Used in FLAC.
- **Lossy compression:** JPEG, MP3, MPEG and others. Also employ Huffman (among other techniques).

Next lecture: how to trade memory for speed and get an $\Theta(n)$ worst case sorting algorithm...