Codeword Encoding--Arithmetic Encoding

Computer Science Department
CS4481b/9628b: Image Compression
Winter 2017
Instructor: Mahmoud R. El-Sakka
Office: MC-419
Email: elsakka@csd.uwo.ca
Phone: 519-661-2111 x86996

Arithmetic Encoding

- Unlike the variable-length codes, the arithmetic encoding scheme generates non-block codes
- In arithmetic encoding scheme, a one-to-one correspondence between source symbols and codewords does not exist
- Instead, an entire string of source symbols is assigned a single arithmetic codeword
- The codeword is defined as a real number (a tag) in a sub-interval between [0 and 1)
- *Is it possible to assign a unique sub-interval to each distinct string of symbols? WHY?*

Arithmetic Encoding

- As the number of symbols in the message increases, the sub-interval used to represent it becomes smaller and the number of bits required to represent any number in the sub-interval becomes larger
- Each symbol in the message reduces the size of the sub-interval in accordance with its probability of occurrence
- Arithmetic encoding is especially useful when dealing with:
  - small alphabets, such as binary, and
  - alphabets with highly skewed probabilities

Arithmetic Encoding

- Example 1:
  - Consider having 4 symbols, A_1, A_2, A_3, and A_4
  - The probabilities of these symbols are 0.2, 0.2, 0.4, and 0.2, respectively
  - It is required to encode the string $A_1A_2A_3A_3A_4$

Arithmetic Encoding

- Based on the information provided, we can say that:
  - $P(A_1) = 0.2$, $P(A_2) = 0.2$, $P(A_3) = 0.4$, and $P(A_4) = 0.2$,
- Assume that X is a one-to-one mapping function, where

$$X(A_i) = i$$

- As a result of this mapping, we can say that
  - the *probability density function*, also called *pdf*, of X is

$$P(X = i) = P(A_i)$$

  - the *cumulative density function*, also called *cdf*, of X is

$$F_X(i) = F_X(A_i) = P(X \le i) = \sum_{k=1}^{i} P(X = k) = \sum_{k=1}^{i} P(A_k)$$

This means that:

$F_X(0) = 0$, $F_X(1) = 0.2$, $F_X(2) = 0.2+0.2 = 0.4$,
$F_X(3) = 0.2+0.2+0.4 = 0.8$, and $F_X(4) = 0.2+0.2+0.4+0.2 = 1.0$

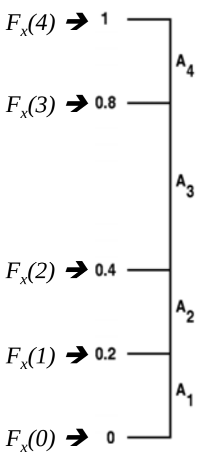
Arithmetic Encoding

- The encoding process starts by dividing the unit interval, i.e., $[0, 1)$, into sub-intervals of the form

$$[F_X(i-1), F_X(i)),$$

where $i = 1, \ldots, number\_of\_symbols$
I.e., [0.0, 0.2), [0.2, 0.4), [0.4, 0.8) and *[0.8, 1)*
- Note that:
  - The intersection between any two sub-intervals are always empty, i.e., these sub-intervals are disjoint from each other
  - The union of all sub-intervals equals $[0, 1)$
- We associate the sub-interval $[F_X(i-1), F_X(i))$ with the symbol A_i

Arithmetic Encoding

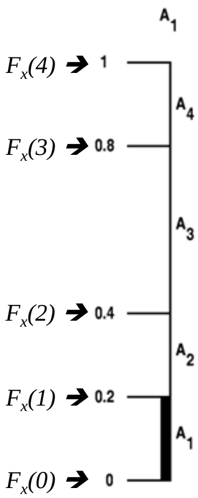


An arithmetic encoding example:
Divide the *current interval* according to symbols' *cdf*

Arithmetic Encoding

- The appearance of the first symbol in the sequence restrict the current interval containing the tag to one of these sub-intervals
- Any number in this new sub-interval is enough to encode the symbol

Arithmetic Encoding

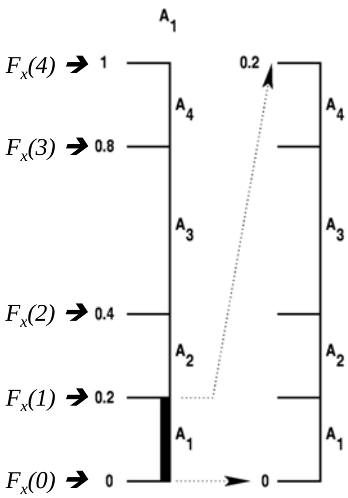


An arithmetic encoding example:
Find the *sub-interval* which is corresponding to the *symbol* to be encoded

Arithmetic Encoding

- This new sub-interval is now partitioned in exactly the same proportions as the original interval

Arithmetic Encoding

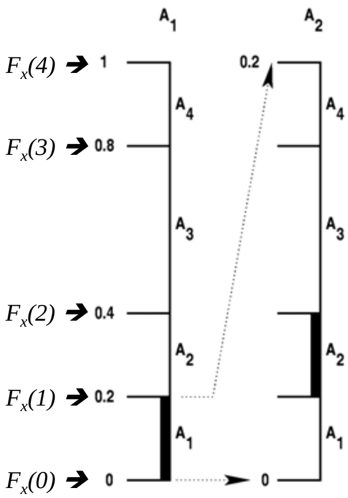


An arithmetic encoding example:
Update the *current interval* to be this *sub-interval*
Divide the *current interval* according to symbols' *cdf*

Arithmetic Encoding

- The appearance of the second symbol in the sequence restrict the sub-interval containing the tag to one of these sub-subintervals
- Any number in this new sub-sub-interval is enough to encode the first and second symbols

Arithmetic Encoding

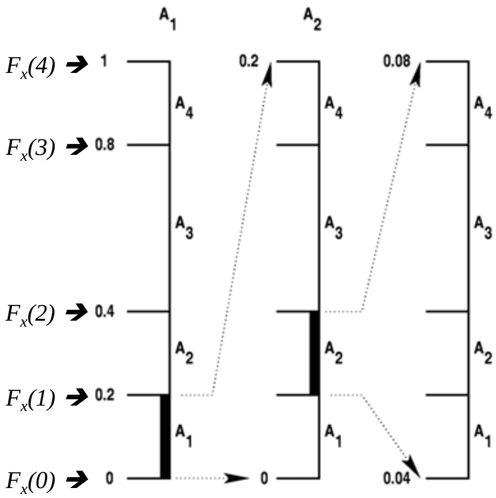


An arithmetic encoding example:
Find the *sub-interval* which is corresponding to the *symbol* to be encoded

Arithmetic Encoding

- Each succeeding symbol causes the tag to be restricted to a sub-interval that is further partitioned in the same proportions.

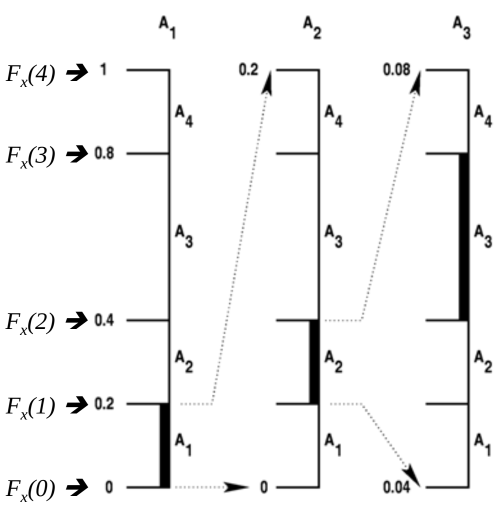
Arithmetic Encoding



An arithmetic encoding example:
Update the *current interval* to be this *sub-interval*
Divide the *current interval* according to symbols' *cdf*

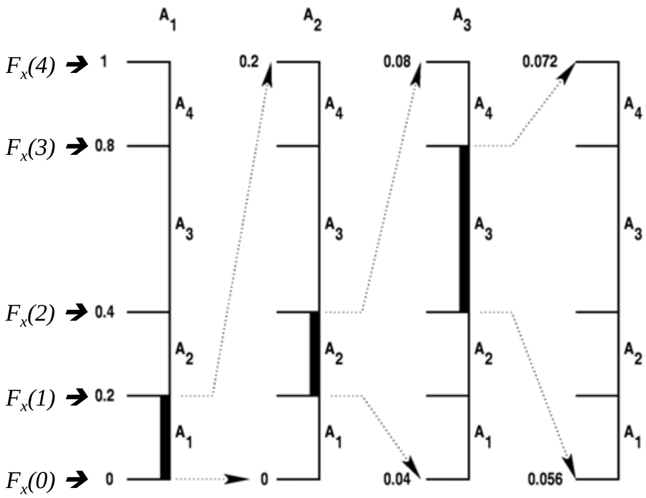
Topic 04: Codeword Encoding--Arithmetic Encoding

Arithmetic Encoding



An arithmetic encoding example:
Find the *sub-interval* which is corresponding to the *symbol* to be encoded

Arithmetic Encoding

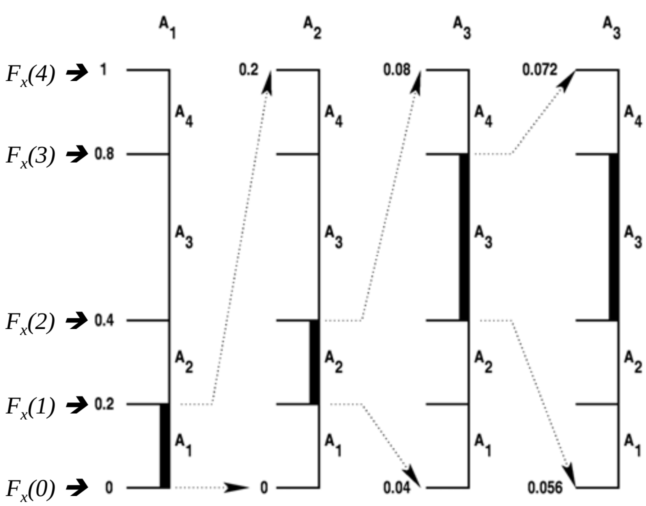


An arithmetic encoding example:

Update the *current interval* to be this *sub-interval*

Divide the *current interval* according to symbols' *cdf*

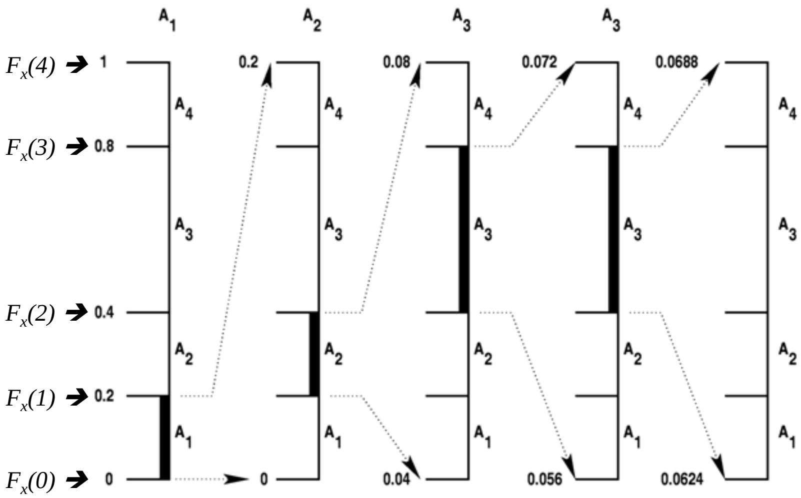
Arithmetic Encoding



An arithmetic encoding example:

Find the *sub-interval* which is corresponding to the *symbol* to be encoded

Arithmetic Encoding



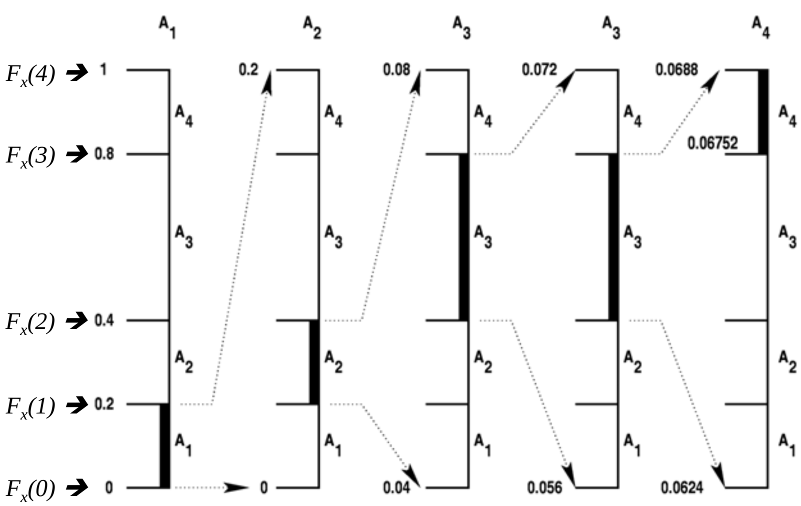
An arithmetic encoding example:

Update the *current interval* to be this *sub-interval*

Divide the *current interval* according to symbols' *cdf*

Topic 04: Codeword Encoding--Arithmetic Encoding

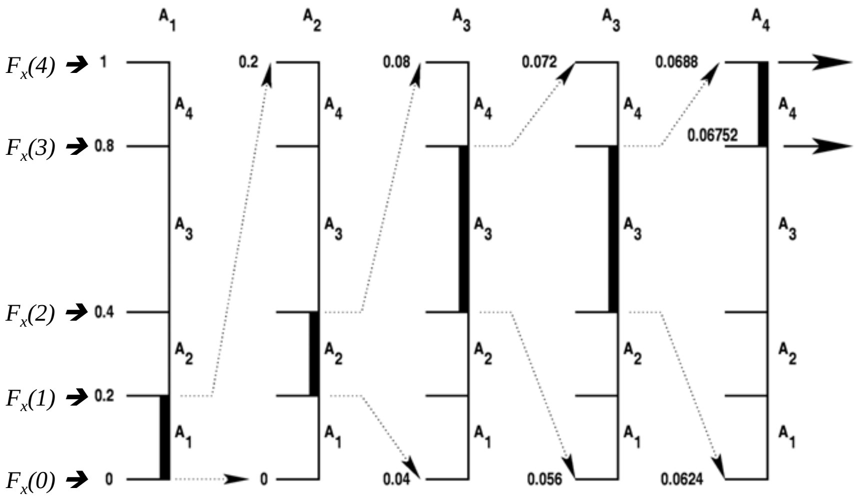
Arithmetic Encoding



An arithmetic encoding example:

Find the *sub-interval* which is corresponding to the *symbol* to be encoded

Arithmetic Encoding



An arithmetic encoding example:
The entire string has been decoded
Any tag number in [0.06752, 0.0688) represents $A_1A_2A_3A_3A_4$

Arithmetic Encoding

- **Step 1:** Initialize the *current interval* to [0, 1)
- **Step 2:** <u>*Repeat*</u>:
- **Step 2.1:** Divide the *current interval* according to symbols' *cdf*
- **Step 2.2:** Find the *sub-interval* which is correspond to the *symbol* to be encoded
- **Step 2.3:** Update the *current interval* to be this *sub-interval*
- **Step 3:** <u>*Until*</u> the entire string has been encoded
- **Step 4:** Select any tag number from the *current interval* to encode the symbols

Arithmetic Encoding

- The sequence of symbols to be encoded can be represented as:
  $$A = A^{(1)}A^{(2)} \ldots A^{(n)},$$
  where i in $A^{(i)}$ means the order of the symbol in the sequence.
- In the previous example, the sequence was $A_1A_2A_3A_3A_4$ hence,
  - $A^{(1)} = A_1$
  - $A^{(2)} = A_2$
  - $A^{(3)} = A_3$
  - $A^{(4)} = A_3$
  - $A^{(5)} = A_4$
- Do not get confused between
  - the superscript (i.e., the order of the symbol within the sequence) and
  - the subscript (the symbol itself).

Arithmetic Encoding

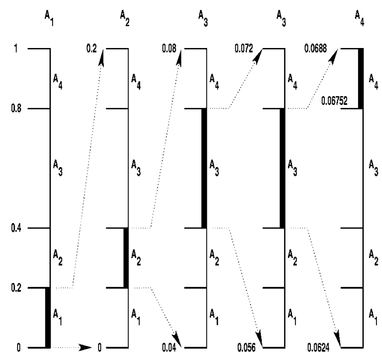
- The upper and lower limits of the interval containing the tag for any sequence $A = A^{(1)}A^{(2)} \ldots A^{(n)}$ can be calculated as follows:
  - Initially, $\text{lower}^{(0)} = 0$ and $\text{upper}^{(0)} = 1$
  - $\text{lower}^{(1)} = \text{lower}^{(0)} + [\text{upper}^{(0)} - \text{lower}^{(0)}] \times F_X(A^{(1)}-1)$
  - $\text{upper}^{(1)} = \text{lower}^{(0)} + [\text{upper}^{(0)} - \text{lower}^{(0)}] \times F_X(A^{(1)})$
  - $\text{lower}^{(2)} = \text{lower}^{(1)} + [\text{upper}^{(1)} - \text{lower}^{(1)}] \times F_X(A^{(2)}-1)$
  - $\text{upper}^{(2)} = \text{lower}^{(1)} + [\text{upper}^{(1)} - \text{lower}^{(1)}] \times F_X(A^{(2)})$

  …….
  - $\text{lower}^{(i)} = \text{lower}^{(i-1)} + [\text{upper}^{(i-1)} - \text{lower}^{(i-1)}] \times F_X(A^{(i)}-1)$
  - $\text{upper}^{(i)} = \text{lower}^{(i-1)} + [\text{upper}^{(i-1)} - \text{lower}^{(i-1)}] \times F_X(A^{(i)})$

  …….
  - $\text{lower}^{(n)} = \text{lower}^{(n-1)} + [\text{upper}^{(n-1)} - \text{lower}^{(n-1)}] \times F_X(A^{(n)}-1)$
  - $\text{upper}^{(n)} = \text{lower}^{(n-1)} + [\text{upper}^{(n-1)} - \text{lower}^{(n-1)}] \times F_X(A^{(n)})$

Arithmetic Encoding

- For example,
  - Note that, in this example:
    - $A^{(1)} = A_1$, $A^{(2)} = A_2$, $A^{(3)} = A_3$, $A^{(4)} = A_3$, and $A^{(5)} = A_4$
    - $F_X(0) = 0$,
      $F_X(1) = 0.2$,
      $F_X(2) = 0.2+0.2 = 0.4$,
      $F_X(3) = 0.2+0.2+0.4 = 0.8$, and
      $F_X(4) = 0.2+0.2+0.4+0.2 = 1.0$
  - $\text{lower}^{(0)} = 0$ and $\text{upper}^{(0)} = 1$
  - $\text{lower}^{(1)} = \text{lower}^{(0)} + [\text{upper}^{(0)} - \text{lower}^{(0)}]\times F_X(A^{(1)}-1)$
  - $\text{lower}^{(1)} = 0 + [1 - 0]\times F_X(A_1-1) = 0 + 1\times F_X(0) = 0$
  - $\text{upper}^{(1)} = \text{lower}^{(0)} + [\text{upper}^{(0)} - \text{lower}^{(0)}]\times F_X(A^{(1)})$
  - $\text{upper}^{(1)} = 0 + [1 - 0]\times F_X(A_1) = 0 + 1\times F_X(1) = 0.2$
  - $\text{lower}^{(2)} = \text{lower}^{(1)} + [\text{upper}^{(1)} - \text{lower}^{(1)}]\times F_X(A^{(2)}-1)$
  - $\text{lower}^{(2)} = 0 + [0.2 - 0]\times F_X(A_2-1) = 0 + 0.2\times F_X(1) = 0.04$
  - $\text{upper}^{(2)} = \text{lower}^{(1)} + [\text{upper}^{(1)} - \text{lower}^{(1)}]\times F_X(A^{(2)})$
  - $\text{upper}^{(2)} = 0 + [0.2 - 0]\times F_X(A_2) = 0 + 0.2\times F_X(2) = 0.08$
  - ……

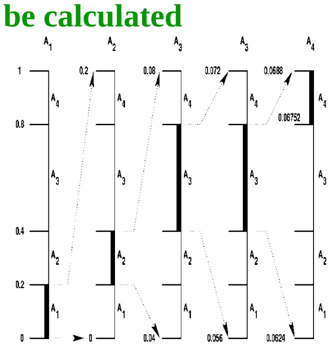


Arithmetic Encoding

- **There is no need to generate all intervals to encode a symbol**
- **You only generate the limits of the interval that you will use**
- **The only information required by the tag generator procedure is the *cdf***
- The tag can be generated in a sequential fashion
- **There is no need for joint probabilities to be calculated**

$F_X(4) = 1.0$
A_4
$F_X(3) = 0.8$
A_3
$F_X(2) = 0.4$
A_2
$F_X(1) = 0.2$
A_1
$F_X(0) = 0.0$

Initial interval → [0.0,1.0)
After encoding A_1 → [0.0, 0.2)
After encoding A_2 → [0.04, 0.08)
After encoding A_3 → [0.056, 0.072)
After encoding A_3 → [0.0624, 0.0688)
After encoding A_4 → [0.06752, 0.0688)



Arithmetic Encoding

- Which number should be selected to represent the interval?
  - Any number within the specified interval can be used to represent the given string of sequence
  - However, the binary representation of some numbers is infinitely long, e.g., $(1/3)_{10} = (0.01010101010....)_2$
  - If the final interval is [0.3, 0.55), then the best binary number to represent this interval is 0.5, i.e., $(0.1)_2$

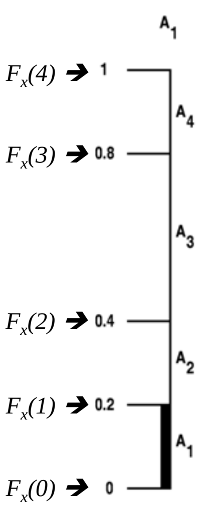
Arithmetic Decoding

- The generated tag is useless unless we can also decipher it with reasonable computational cost
- The idea is to just mimic the encoder in order to successfully do the decoding
- Our decoding strategy is to decode the elements in the sequence in such a way that the upper and lower limits always contain the tag value

Arithmetic Decoding

- Let us try to decode the previous example, where that selected tag is 0.068
  - We start with the interval *[0, 1)*
  - Divide the *current interval* according to symbols' *cdf*
  - The interval containing the 0.068 tag value is [0.0, 0.2), hence the first decoded symbol is A_1

Arithmetic Decoding

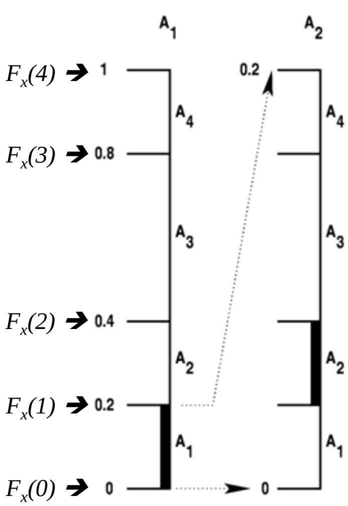


An arithmetic Decoding example:
Find the *sub-interval* which includes the *tag number and*
Decode the *symbol* corresponding to this *sub-interval*

Arithmetic Decoding

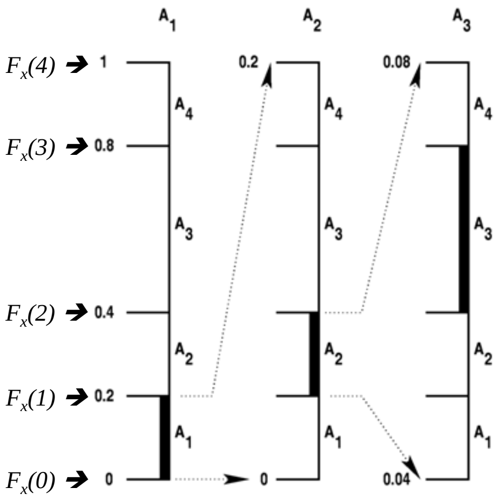
- This new sub-interval is now partitioned in exactly the same proportions as the original interval
- The sub-interval containing the 0.068 tag value is [0.04, 0.08), hence the seconf decoded symbol is A_2

Arithmetic Decoding



An arithmetic Decoding example:
Find the *sub-interval* which includes the *tag number and*
Decode the *symbol* corresponding to this *sub-interval*

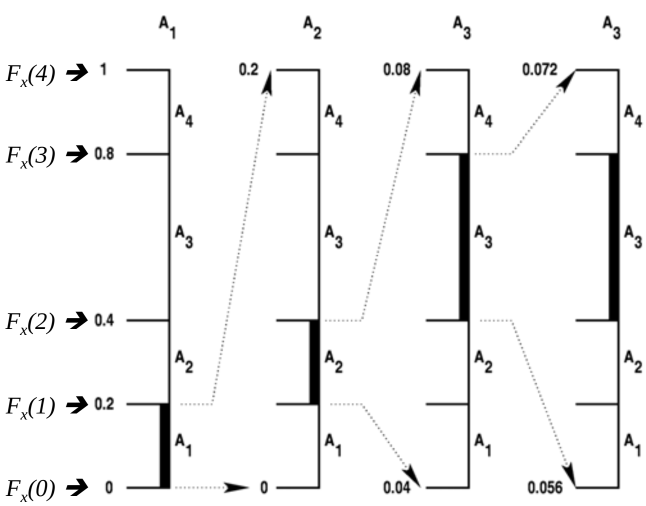
Arithmetic Decoding



An arithmetic Decoding example:
Find the *sub-interval* which includes the *tag number and*
Decode the *symbol* corresponding to this *sub-interval*

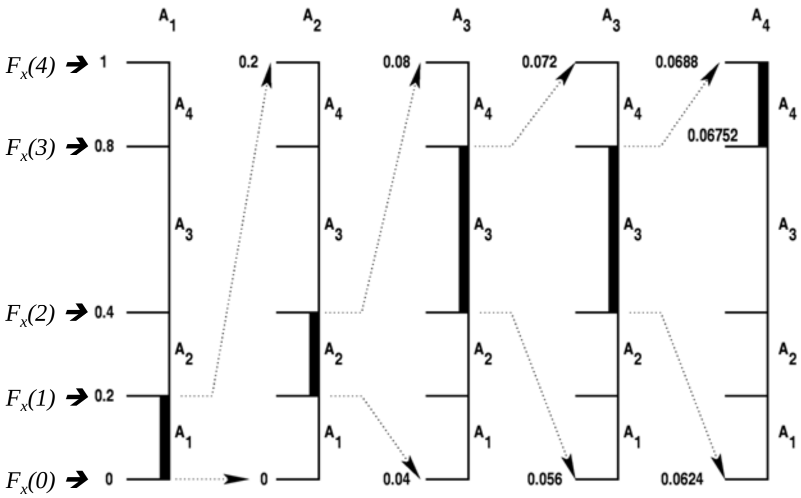
Topic 04: Codeword Encoding--Arithmetic Encoding

Arithmetic Decoding



An arithmetic Decoding example:
Find the *sub-interval* which includes the *tag number and*
Decode the *symbol* corresponding to this *sub-interval*

Arithmetic Decoding



An arithmetic Decoding example:
Find the *sub-interval* which includes the *tag number and*
Decode the *symbol* corresponding to this *sub-interval*

Arithmetic Decoding

- **Step 1:** Initialize the *current interval* to [0, 1)
- **Step 2:** *Repeat*:
- **Step 2.1:** Divide the *current interval* according to symbols' *cdf*
- **Step 2.2:** Find the *sub-interval* which includes the *encoded tag value*
- **Step 2.3:** Decode the *symbol* corresponding to this *sub-interval*
- **Step 2.4:** Update the *current interval* to be this *sub-interval*
- **Step 3:** *Until* the entire string has been decoded

Arithmetic Decoding

- For each symbol to be decoded, we should find k such that:

$$F_x(A^{(k)}-1) \leq \frac{tag - lower^{(k-1)}}{upper^{(k-1)} - lower^{(k-1)}} < F_x(A^{(k)})$$

Where $A^{(k)}$ is the symbol number k in the decoded sequence

Arithmetic Decoding

$$F_x(A^{(k)}-1) \leq \frac{tag - lower^{(k-1)}}{upper^{(k-1)} - lower^{(k-1)}} < F_x(A^{(k)})$$

$F_X(4) = 1.0$
A_4
$F_X(3) = 0.8$
A_3
$F_X(2) = 0.4$
A_2
$F_X(1) = 0.2$
A_1
$F_X(0) = 0.0$

$lower^{(0)} = 0$, $upper^{(0)} = 1$, tag = 0.068

$(Tag - lower^{(0)}) / (upper^{(0)} - lower^{(0)}) = (0.068 - 0)/(1 - 0) = 0.068 \rightarrow A_1$
$lower^{(1)} = lower^{(0)} + [upper^{(0)} - lower^{(0)}] \times F_X(A_1 - 1) = 0$
$upper^{(1)} = lower^{(0)} + [upper^{(0)} - lower^{(0)}] \times F_X(A_1) = 0.2$

$(Tag - lower^{(1)}) / (upper^{(1)} - lower^{(1)}) = (0.068 - 0)/(0.2 - 0) = 0.34 \rightarrow A_2$
$lower^{(2)} = lower^{(1)} + [upper^{(1)} - lower^{(1)}] \times F_X(A_2 - 1) = 0.04$
$upper^{(2)} = lower^{(1)} + [upper^{(1)} - lower^{(1)}] \times F_X(A_2) = 0.08$

$(Tag - lower^{(2)}) / (upper^{(2)} - lower^{(2)}) = (0.068 - 0.04)/(0.08 - 0.04) = 0.7 \rightarrow A_3$
$lower^{(3)} = lower^{(2)} + [upper^{(2)} - lower^{(2)}] \times F_X(A_3 - 1) = 0.056$
$upper^{(3)} = lower^{(2)} + [upper^{(2)} - lower^{(2)}] \times F_X(A_3) = 0.072$

$(Tag - lower^{(3)}) / (upper^{(3)} - lower^{(3)}) = (0.068 - 0.056)/(0.056 - 0.072) = 0.75 \rightarrow A_3$
$lower^{(4)} = lower^{(3)} + [upper^{(3)} - lower^{(3)}] \times F_X(A_3 - 1) = 0.0624$
$upper^{(4)} = lower^{(3)} + [upper^{(3)} - lower^{(3)}] \times F_X(A_3) = 0.0688$

$(Tag - lower^{(4)})/(upper^{(4)} - lower^{(4)}) = (0.068 - 0.0624)/(0.0688 - 0.0624) = 0.875 \rightarrow A_4$
$lower^{(5)} = lower^{(4)} + [upper^{(4)} - lower^{(4)}] \times F_X(A_4 - 1) = 0.06752$
$upper^{(5)} = lower^{(4)} + [upper^{(4)} - lower^{(4)}] \times F_X(A_4) = 0.0688$

Arithmetic Decoding

- The decoder may know the length of the encoded string
  - Explicitly, i.e., through some sort of side information, or
  - Implicitly, i.e., hidden information in the application itself

  In this case, the decoding process is stopped when that many symbols have been obtained

The Precision Issue

- *Is it possible to encode the whole British encyclopedia using just a number?*
- Note that, each succeeding interval, i.e., sub-interval, is contained in the preceding interval
- An undesirable consequence of this process is that the intervals get smaller and smaller and require higher precision as the string gets longer
- Theoretically speaking, there are infinite numbers in the interval [0, 1)
- *However, in practice the number of numbers that can be uniquely represented on a machine is limited by the maximum number of bits used to represent the number*
- In order to uniquely represent all of the sub-intervals, an increasing precision is needed, as the length of the encoded string increases
- *How can we overcome this precision problem?*

The Precision Issue

- To overcome this precision problem we should deal with binary codes directly during the encoding/decoding processes
  - An interval rescaling scheme is needed
  - This rescaling scheme must preserve the already encoded information

Rescaling Scheme (Encoding)

- As the interval becomes narrower, there are three possibilities:
  - the interval is entirely confined to the lower half of the unit interval, i.e., [0, 1/2)
  - the interval is entirely confined to the upper half of the unit interval, i.e., [1/2, 1)
  - the interval contains the midpoint of the unit interval and it is contained in the interval [1/4, 3/4)
- Once the interval is confined to either the upper, or lower, half of the unit interval, it is forever confined to that half of the unit interval, i.e., the most significant bit of the binary representation
  - of all numbers in the interval [0, 1/2) is 0 and
  - of all numbers in the interval [1/2, 1) is 1
- *How to convert a decimal fraction to binary fraction?*

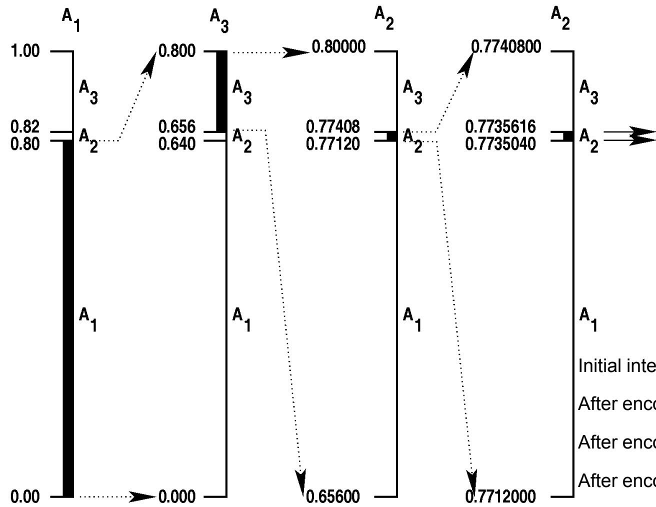
Rescaling Scheme (Encoding)

- Therefore, without waiting to see what the rest of the sequence looks like
  - This most significant bit can be sent/stored in the compressed file
  - Both the encoder ***and decoder*** can re-scale the interval as follow: from [0, 1/2) to [0, 1) or from [1/2, 1) to [0, 1)
  - As soon as we perform either of these mapping, all information about the most significant bit is lost; however, this should not matter, since we have already stored/sent this bit to the decoder
  - This procedure is repeated each time we have a similar situation

Rescaling Scheme (Encoding)

- Example 2:
  - Consider having 3 symbols, A_1, A_2, and A_3
  - The probabilities of these symbols are 0.8, 0.02, and 0.18, respectively
  - It is required to encode the string $A_1A_3A_2A_2$

Rescaling Scheme (Encoding)

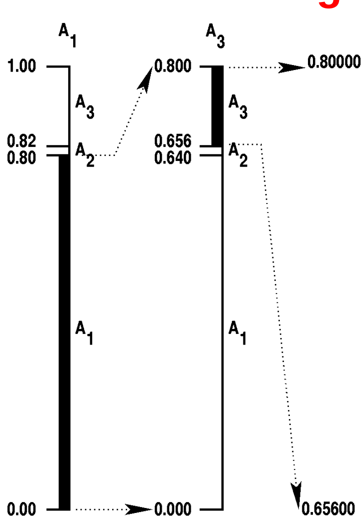


$F_X(3) = 1.0$
A_3
$F_X(2) = 0.82$
A_2
$F_X(1) = 0.8$
A_1
$F_X(0) = 0.0$

Initial interval ➔ [0.0,1.0)
After encoding A_1➔ [0.0, 0.8)
After encoding A_3➔ [0.656, 0.8)
After encoding A_2➔ [0.7712, 0.77408)
After encoding A_2➔ [0.773504, 0.7735616)

An arithmetic encoding example

Rescaling Scheme (Encoding)



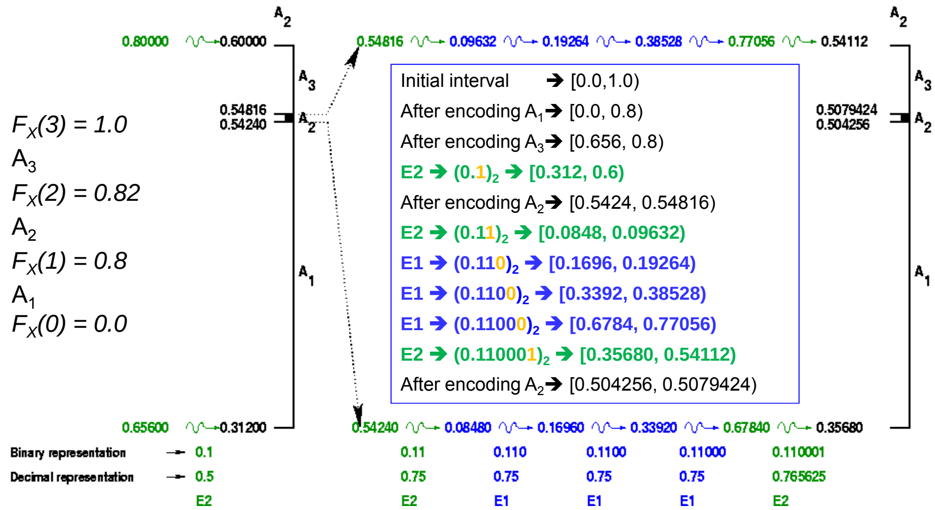
$F_X(3) = 1.0$
A_3
$F_X(2) = 0.82$
A_2
$F_X(1) = 0.8$
A_1
$F_X(0) = 0.0$

To be continued in the next pages

Initial interval ➔ [0.0,1.0)
After encoding A_1➔ [0.0, 0.8)
After encoding A_3➔ [0.656, 0.8)

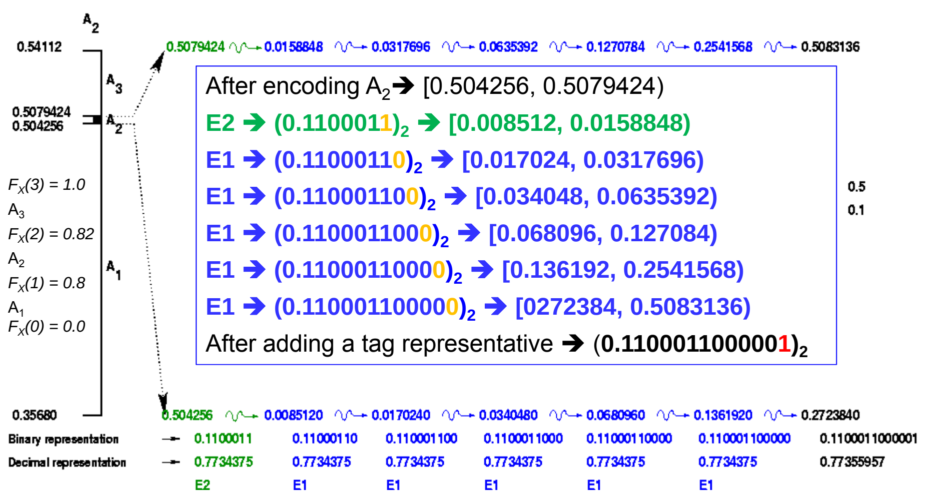
An arithmetic encoding example: the [1/4, 3/4) → [0, 1) mapping was **not applied**

Rescaling Scheme (Encoding)



An arithmetic encoding example: the [1/4, 3/4) → [0, 1) mapping was **not applied**

Rescaling Scheme (Encoding)



An arithmetic encoding example: the [1/4, 3/4) → [0, 1) mapping was **not applied**

Rescaling Scheme (Encoding)

- In the last example, rescaling is applied 12 times
- Without rescaling,
  - the final interval is [0.7735040, 0.7735616)
  - the length of the final interval is 0.0000576
- With rescaling,
  - the final interval is [0.2723840, 0.5083136)
  - the length of the final interval is 0.2359296
- The ratio between final intervals is 0.2359296 / 0.0000576 = 4096 = 2^{12}
  i.e., the final interval has been enlarged by $2^{\text{number of rescaling applications}}$
- The bits that we have sent during the rescaling process represent the tag itself

Topic 04: Codeword Encoding--Arithmetic Encoding

Rescaling Scheme (Decoding)

- Note that:
  - Before decoding a symbol, the decoder makes sure that there are enough bits to unambiguously decode this symbol
  - Based on the smallest interval, the decoder can determined how many bits it needs before it can start the decoding procedure
    - If *P(x)* is the probability of the smallest interval, the minimum number of bits required to start decoding is

$$\left\lceil \log_2\left(\frac{1}{P(x)}\right)+1\right\rceil+2$$

    - In the previous example, the probability of the smallest interval was 0.02, hence the minimum number of bits required to start decoding is 7+2 bits, since $0.02 = 2^{-5.644}$
  - The decoder keeps mimicking the rescaling process, which the encoder did
  - This process continued until all symbols are decoded

Rescaling Scheme (Decoding)

- Example 3:
  - Consider having 3 symbols, A_1, A_2, and A_3
  - The probabilities of these symbols are 0.8, 0.02, and 0.18, respectively
  - The encoded bit-stream is $(0.1100011000001)_2$
  - It is required to decode 4 symbols using the above encoded bit-stream

Rescaling Scheme (Decoding)

- To decode, we need to
  - Initialize all ***cdf*** values, i.e.,

    $F_X(0) = 0.0, F_X(1) = 0.8, F_X(2) = 0.82, \text{ and } F_X(3) = 1.0$
  - Read to first *n* bits of the compressed file to initialize the ***tag***
  - Initialize the lower and upper limits to 0 and 1, respective

    $lower^{(0)} = 0$ and $upper^{(0)} = 1$
  - Repeat the following steps until decoding all required symbols
    - Decode a symbol using:

      $$F_x(A^{(k)} - 1) \le \frac{tag - lower^{(k-1)}}{upper^{(k-1)} - lower^{(k-1)}} < F_x(A^{(k)})$$

      I.e., find the value of $A^{(k)}$ that satisfies the above equation
    - Adjust the limits using:

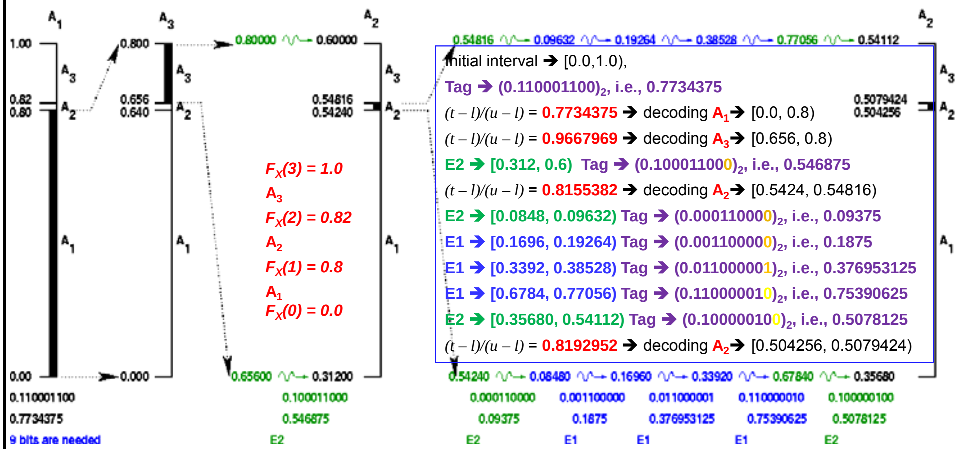
      $lower^{(k)} = lower^{(k-1)} + [upper^{(k-1)} - lower^{(k-1)}] \times F_X(A^{(k)} - 1)$

      $upper^{(k)} = lower^{(k-1)} + [upper^{(k-1)} - lower^{(k-1)}] \times F_X(A^{(k)})$
    - If possible, apply the rescaling procedures and adjust the limits and the tag

Rescaling Scheme (Decoding)

$lower^{(k)} = lower^{(k-1)} + [upper^{(k-1)} - lower^{(k-1)}] \times F_X(A^{(k)}-1)$
$upper^{(k)} = lower^{(k-1)} + [upper^{(k-1)} - lower^{(k-1)}] \times F_X(A^{(k)})$

$$F_x(A^{(k)}-1) \leq \frac{tag - lower^{(k-1)}}{upper^{(k-1)} - lower^{(k-1)}} < F_x(A^{(k)})$$



An arithmetic decoding example: the encoded bit-stream is $(0.1100011000001)_2$

the [1/4, 3/4) → [0, 1) mapping was **not applied**