Greedy Algorithms

for Optimization Problems

Alexandra Stefan

Optimization Problems

- *Knapsack problem:*
  - Thief in a store has a backpack. Can only steal as much as fits in his backpack. What objects should he pick to make the most money? Given data: W-knapsack capacity, N – number of different types of items; value and weight (v_i,w_i) of each item.
  - Versions – see next page
- *Job Scheduling (a.k.a. Interval Scheduling)*
  - Given N jobs with (start_time, end_time, value) pick a set of jobs that do not overlap and give the most money.
    - Variation: all jobs have the same value
- *Hufmann coding*
  - File compression: encode symbols in a text file s.t. the file has the smallest size possible.
- *Graphs – Minimum Spanning Tree (MST-Prim's Algorithm)*

- Terminology:
  - *Problem* - general
  - *Variations of a problem* – additional specification to the general problem
  - *Instance of a problem* – specific data (what I use in examples are instances of the problem being discussed problem)

Greedy Method for Optimization Problems

- Optimization problem – multiple possible solutions, pick the one that gives the most value (or lowest cost)
- Greedy:
  - Method:
    - **Pick a criterion that reflects the measure you are optimizing for (value or cost)**
      E.g. for Huffman minimize the storage (cost), for Knapsack maximize the money (value)
    - **take the action that is best now** (out of the current options) according to your criterion (i.e. pick a local optimal). **You commit to it, it may limit your future choices and cause you to not find the global optimum.**
  - **It may cause you to miss the optimal solution**
  - **You build the solution as you go. No need to back trace it**.
- Examples:
  - Knapsack
    - Optimal answer for the easy (fractional) version using ratio, but not for the others.
  - Interval scheduling
    - Optimal answer for the easy (same value jobs) version (using job that finishes first) , but not for the others.
  - Huffman codes
    - Optimal solution: pick the two smallest weight trees.
    - Used for file compression. Prefix codes: no code is also a prefix of another code.
    - Huffman tree to decode (the code for a character, x, takes you to the leaf with x)

The Knapsack Problem

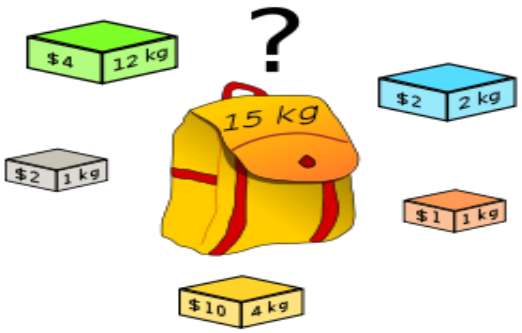


Image from Wikipedia

- A thief breaks into a store.
- The maximum total weight that he can carry is *W*.
- There are *N* types of items at the store.
- Each type t_i has a value v_i and a weight w_i.
- What is the maximum total **value** that he can carry out?
- What items should he pick to obtain this maximum value?

Typical version:
0/1
(only one of each object)

Can either pick object i, or not pick it.

Unbounded
(unlimited number of each object)
Can pick any object, any number of times.

Bounded
(limited number of each object)
Can pick object i, at most c_i times.
(Not covered in this class)

All versions have:	
N	number of different types of objects
W	the maximum capacity (kg)
$v_1, v_2, \ldots, v_N$	Value for each object. ($$)
$w_1, w_1, \ldots, w_N$,	Weight of each object. (kg)

Fractional
For each item can take the whole quantity, or a fraction of the quantity.

The only variation that Greedy can solve optimally

Variations we will discuss here:

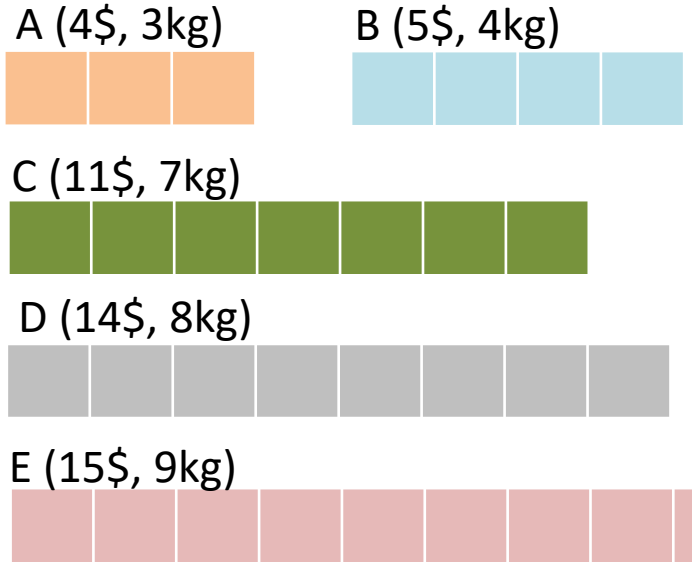
0/1 non-fractional	0/1 fractional
Unbounded non-fractional	Unbounded fractional

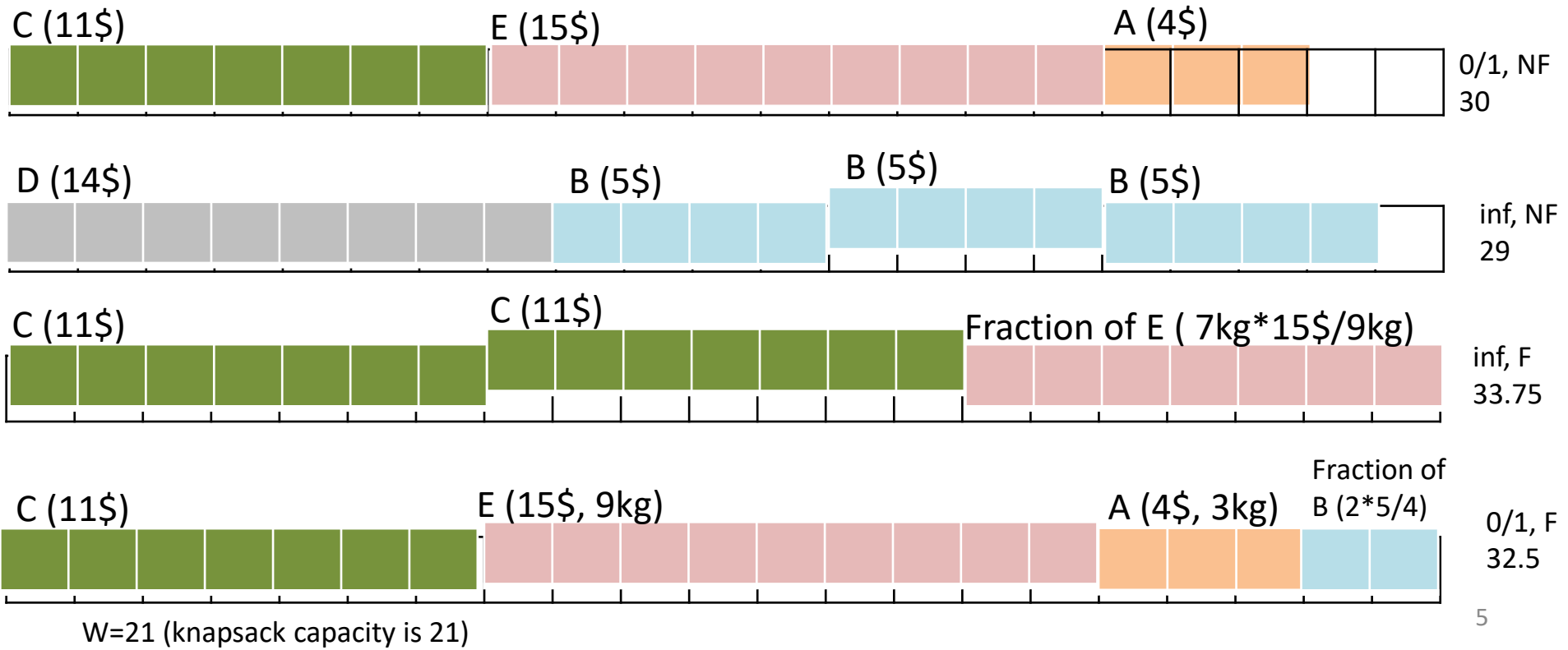
All four versions example – NOT optimal

Item	A	B	C	D	E
Value ($)	4	5	11	14	15
Weight (kg)	3	4	7	8	9

W=21 (i.e. the Knapsack capacity is 21)

Price per kilogram = value/weight.
E.g. for B we have 5$/4kg = 1.25$/kg .
(Useful for the fractional version in calculating the money made from using a fraction of an item. See the examples for the fractional problems below.)





Knapsack – Greedy solution

- What would a greedy thief do?
  - Criterion: value/weight ratio
  - Method:
    - Sort in decreasing order of value/weight ratio.
    - Pick as many of the largest ratio as possible. After that, try to take as many of the next ratio as possible and so on.
  - Does NOT give an optimal solution. See next page.

- Would any of the 4 variations be solved optimally using Greedy? (Prove or give counter-example)
  - Yes. The fractional version.

Worksheet (make copies)

Item	A	B	C	D	E
Value	4	5	11	14	15
Weight	3	4	7	8	9
Ratio	4/3= 1.3	5/4= 1.25	11/7= 1.57	14/8= 1.75	15/9= 1.67

Reordered decreasing by **ratio**: **D, E, C, A, B**

1.75, 1.67, 1.57, 1.3, 1.25

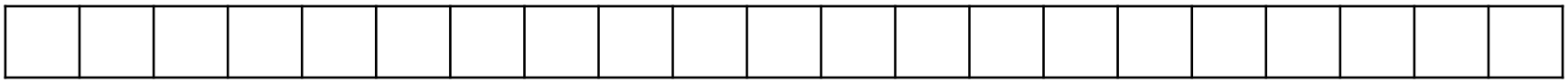
D (1.75=14/8)

E (1.67=15/9)

C (1.57=11/7)

A (1.3=4/3)

B (1.25=5/4)



W=21 (knapsack capacity is 21)

0/1 Not Fractional, Ratio

Item	A	B	C	D	E
Value	4	5	11	14	15
Weight	3	4	7	8	9
Ratio	4/3= 1.3	5/4= 1.25	11/7= 1.57	14/8= 1.75	15/9= 1.67

Reordered decreasing by **ratio**: **D, E, C, A, B**

1.75, 1.67, 1.57, 1.3, 1.25

D (1.75=14/8)

E (1.67=15/9)

C (1.57=11/7)

A (1.3=4/3)

B (1.25=5/4)

We can only pick entire objects, must skip those that do not fit. We have only one of each object.

D (1.75=14/8) E (1.67=15/9) A (1.3=4/3)

W=21 (knapsack capacity is 21)

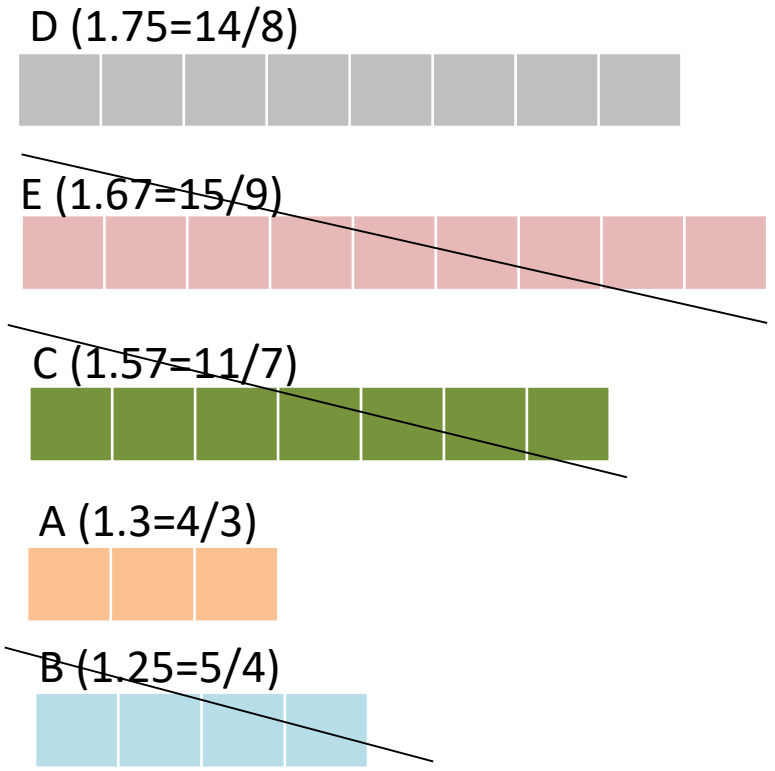
Total value = value(D) + value(E) + value(A) =14 + 15 + 4 =33

Unbounded Not Fractional, RATIO

Item	A	B	C	D	E
Value	4	5	11	14	15
Weight	3	4	7	8	9
Ratio	4/3= 1.3	5/4= 1.25	11/7= 1.57	14/8= 1.75	15/9= 1.67

Reordered decreasing by **ratio**: **D, E, C, A, B**

1.75, 1.67, 1.57, 1.3, 1.25



We can only pick entire objects, must skip those that do not fit. We have unlimited number of objects, thus we can pick as many of D as we can fit in.

W=21 (knapsack capacity is 21)

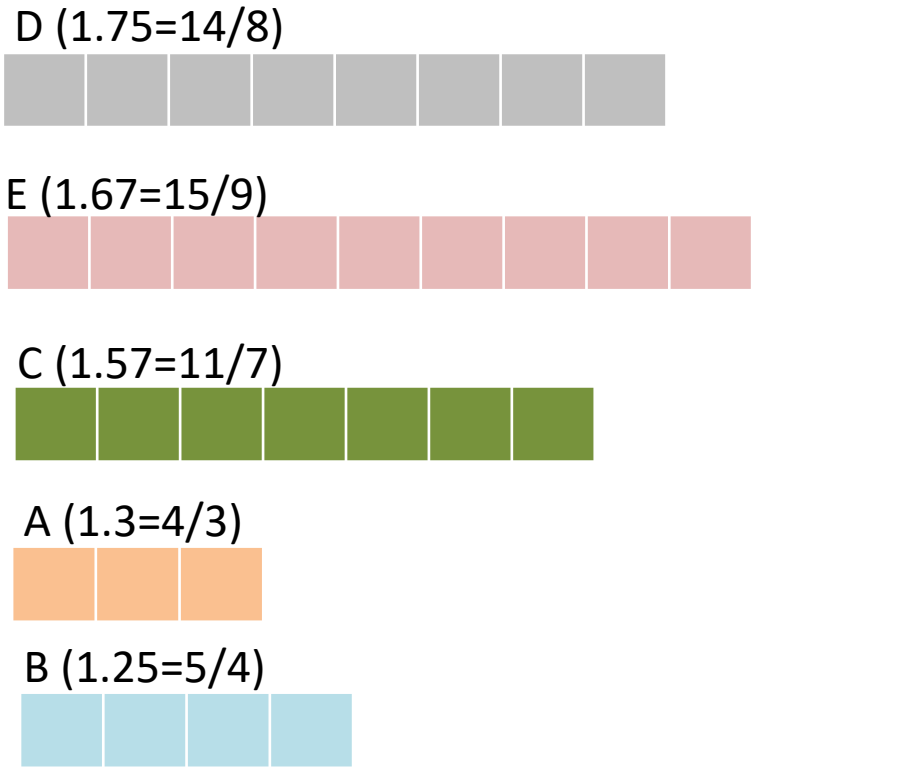
Total value = value(D) + value(D) + value(A) =14 + 14 + 4 =32
(D was selected twice)

Unbounded Fractional, RATIO - Solution

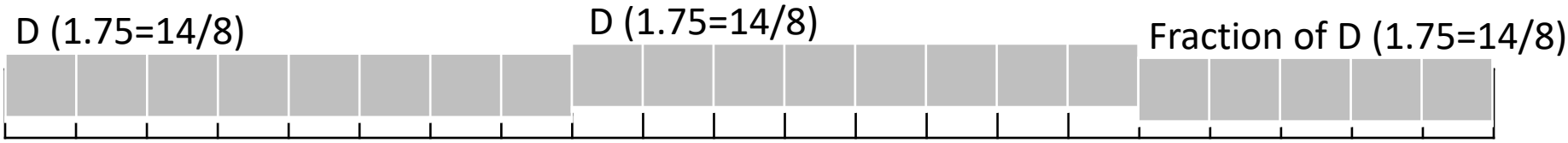
Item	A	B	C	D	E
Value	4	5	11	14	15
Weight	3	4	7	8	9
Ratio	4/3= 1.3	5/4= 1.25	11/7= 1.57	14/8= 1.75	15/9= 1.67

Reordered decreasing by **ratio**: **D, E, C, A, B**

1.75, 1.67, 1.57, 1.3, 1.25



We can pick a **fraction** of an object, must skip those that do not fit. We have unlimited number of objects, thus we can pick only D (entire objects and a fraction).



W=21 (knapsack capacity is 21)

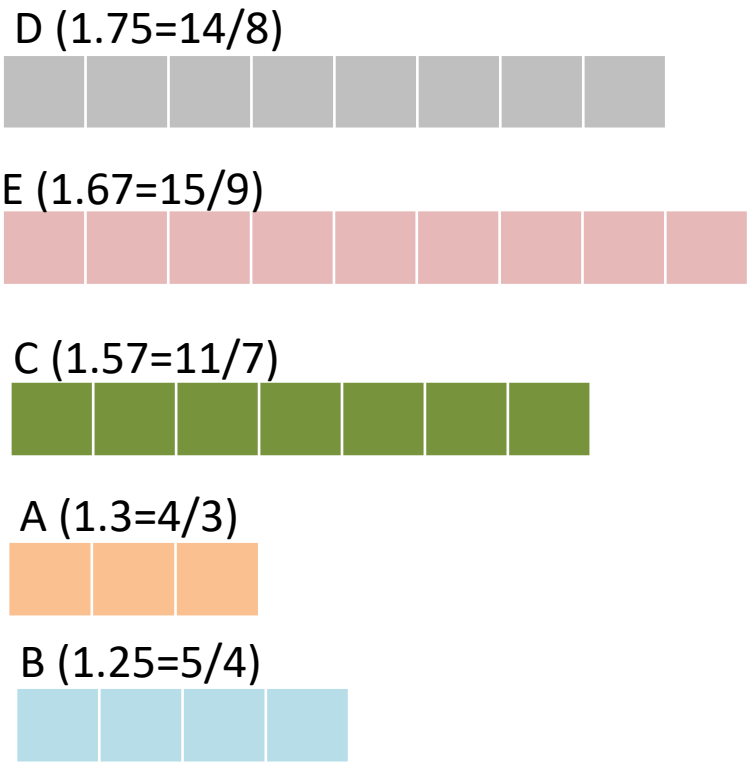
Total value = value(D) + value(D) + remining_weight*(value(D)/weight(D)) =
= capacity_kg* ($/kg for D) = 21kg*(14$/8kg) = 36.75$

0/1 Fractional, RATIO - Solution

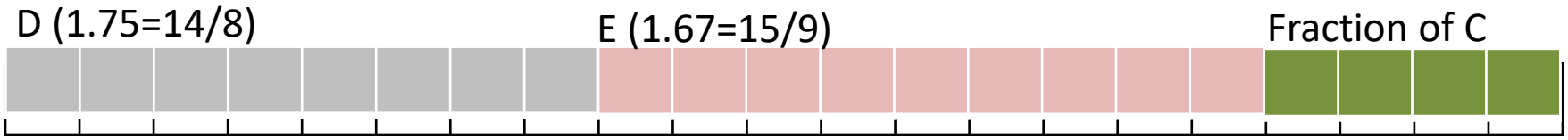
Item	A	B	C	D	E
Value	4	5	11	14	15
Weight	3	4	7	8	9
Ratio	4/3= 1.3	5/4= 1.25	11/7= 1.57	14/8= 1.75	15/9= 1.67

Reordered decreasing by **ratio**: **D, E, C, A, B**

1.75, 1.67, 1.57, 1.3, 1.25



We can pick a **fraction** of an object, must skip those that do not fit. We have only ONE of each object.



W=21 (knapsack capacity is 21)

Total value = value(D) + value(E) + remining_weight*(value(C)/weight(C)) =

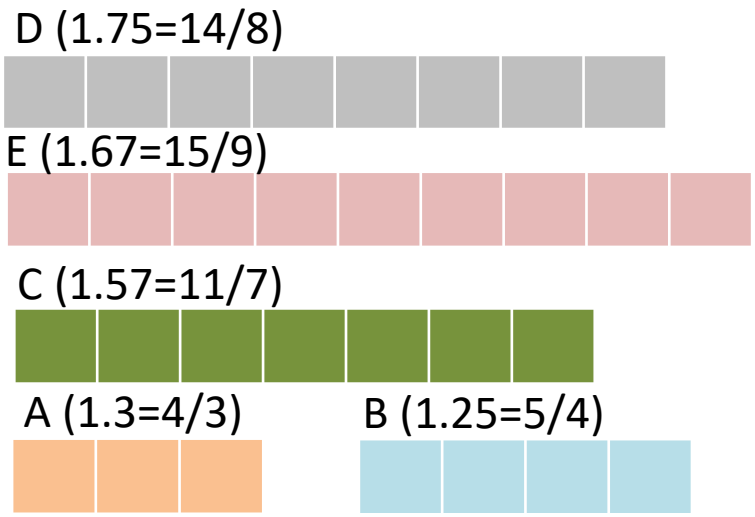
= 14$ + 15$ + 4kg* (11$/7kg)= 35.28$

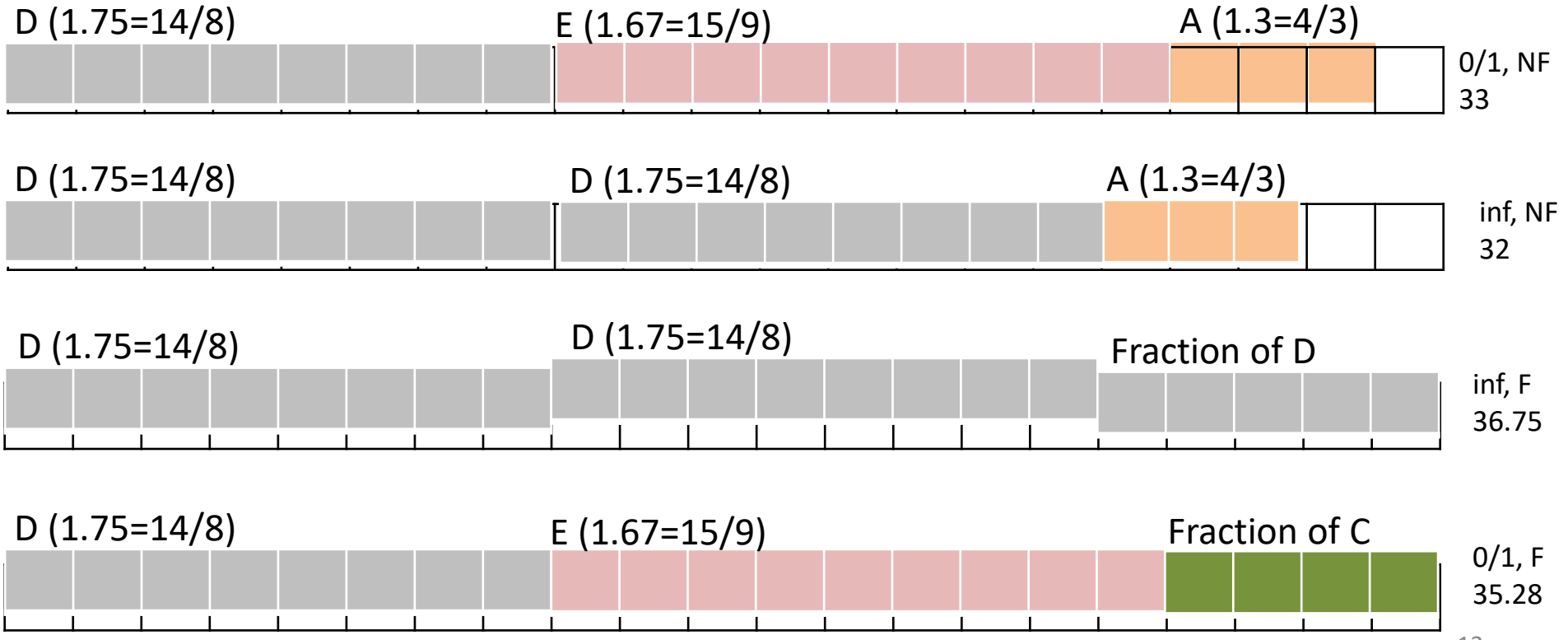
All four versions, Ratio

Item	A	B	C	D	E
Value	4	5	11	14	15
Weight	3	4	7	8	9
Ratio	4/3= 1.3	5/4= 1.25	11/7= 1.57	14/8= 1.75	15/9= 1.67

Reordered decreasing by **ratio**: **D, E, C, A, B**

1.75, 1.67, 1.57, 1.3, 1.25





W=21 (knapsack capacity is 21)

Greedy for Knapsack – Criterion: Ratio

W = 21 (Knapsack capacity: 21)

Item	A	B	C	D	E
Value	4	5	11	14	15
Weight	3	4	7	8	9
Ratio	4/3= 1.3	5/4= 1.25	11/7 =1.57	14/8 =1.75	15/9 =1.67

Reordered decreasing by **ratio**: **D, E, C, A, B**

Best available item now: C, weight of C: 7
Remaining weight: 4
Want to use all remaining space with a fraction of D or C => Value calculation:
Remaining_weight*(value/kg)

Unbounded, not fractional

Picked	D	D	A
Remaining weight	13 (=21-8)	5 (=13-8)	2 (=5-3)

Value: **14+14+4 = 32**

Unbounded, fractional

Picked	D	D	Fraction of of D
Remaining weight	13 (=21-8)	5 (=13-8)	0 (=5-5)

Value: **14+14+5*(14/8) = 36.75**

0/1, not fractional

Picked	D	E	A
Remaining weight	13 (=21-8)	4 (=13-9)	1 (=4-3)

Value: **14+15+4 = 33**

0/1, fractional

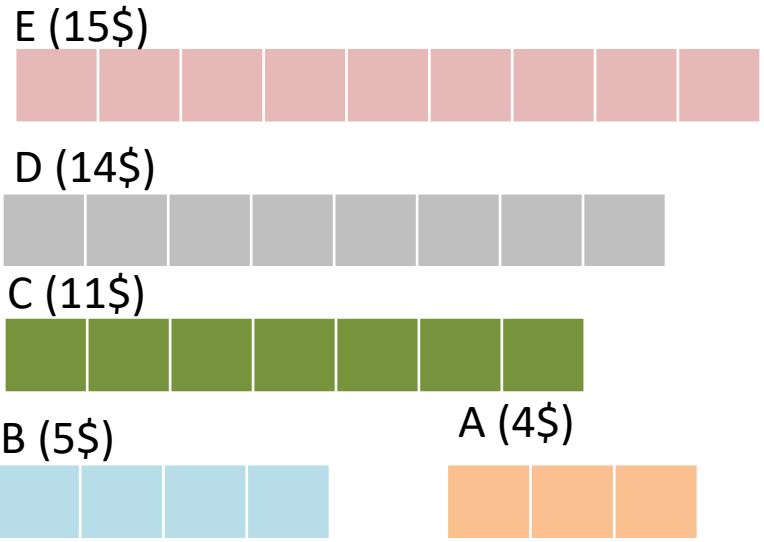
Picked	D	E	Fraction of C
Remaining weight	13 (=21-8)	4 (=13-9)	0 (=4-4)

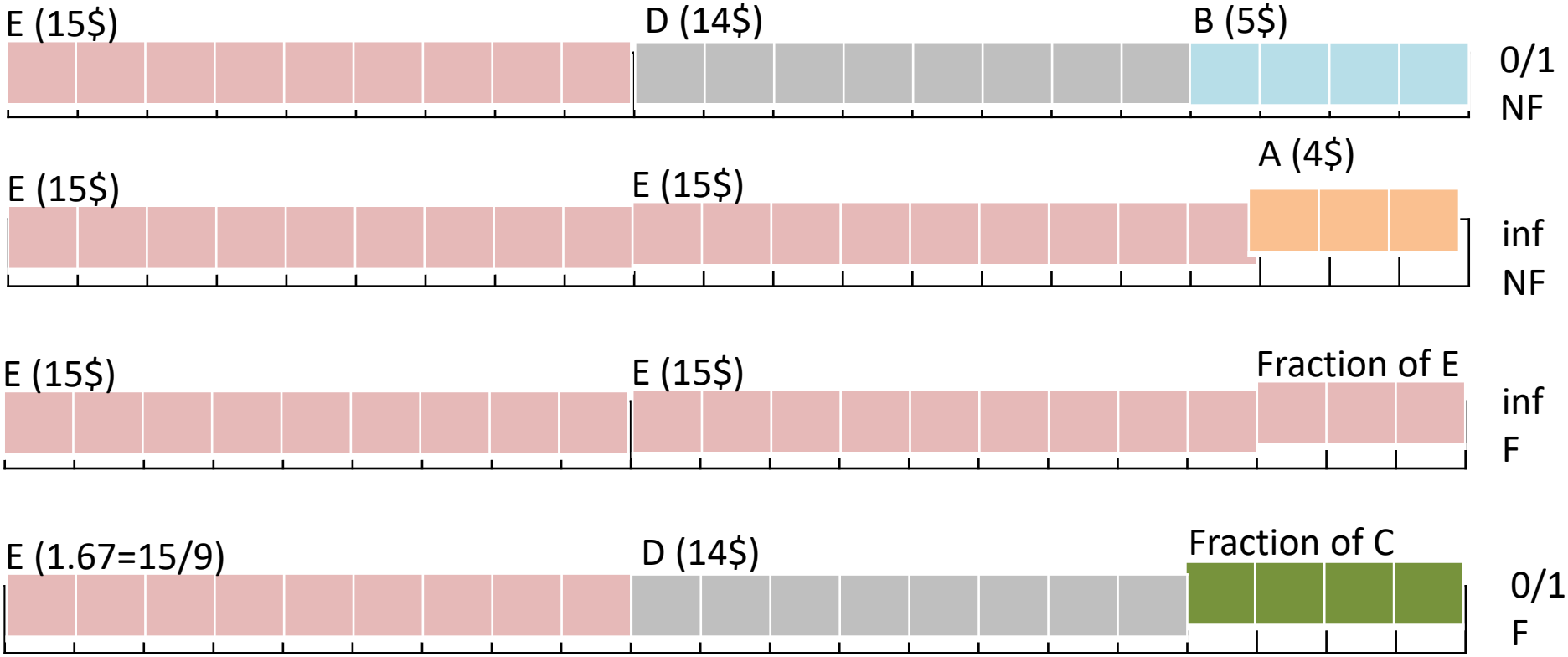
Value: **14+15+ 4*(11/7) = 35.28**

All four versions, VALUE

Item	A	B	C	D	E
Value	4	5	11	14	15
Weight	3	4	7	8	9

Reordered decreasing by **value**: **E, D, C, B, A**





W=21 (knapsack capacity is 21)

Greedy for Knapsack – Criterion: Value

Item	A	B	C	D	E
Value	4	5	11	14	15
Weight	3	4	7	8	9

Reordered decreasing by **value**: **E, D, C, B, A**

W = 21 (Knapsack capacity: 21)

Best item: E, weight of E: 9
Remaining weight: 3
Want to use all remaining space with a fraction of D or C => Value calculation: Remaining_weight*(value/kg)

Unbounded , not fractional

Picked	E	E	A
Remaining weight	12 (=21-9)	3 (=12-9)	0 (=3-3)

Value: **15+15+4 = 34**

Unbounded, fractional

Picked	E	E	1/3 of E
Remaining weight	12 (=21-9)	3 (=12-9)	0 =3-(1/3)*9

Value: **15+15+(1/3)*15 = 35**

0/1 , not fractional

Picked	E	D	B
Remaining weight	12 (=21-9)	4 (=12-8)	0 (=4-4)

Value: **15+14+5 = 34**

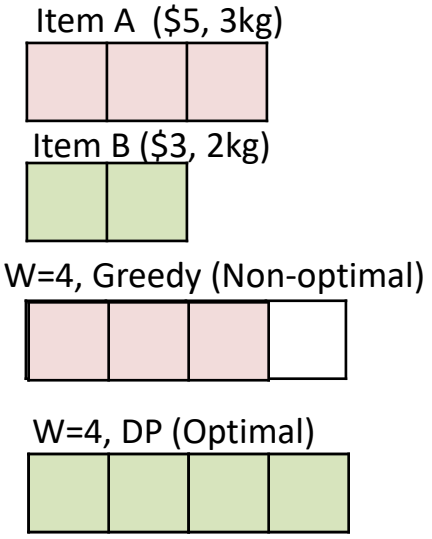
0/1, fractional

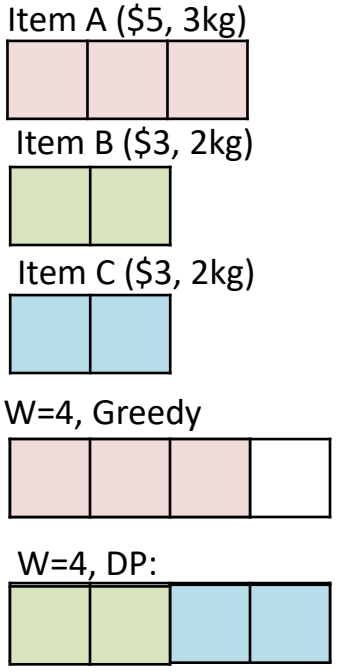
Picked	E	D	4/7 of C
Remaining weight	12 (=21-9)	4 (=12-8)	0 =4-(4/7)*7

Value: **15+14+(4/7)*11 = 35.28**

Greedy may NOT find the optimum solution for Unbounded and 0/1 Knapsack - Proof by counter example

- Goal: construct an **Unbounded** Knapsack instance where Greedy (with the **ratio**) does not give the optimal answer.
- Intuition: We want Greedy to pick only one item, when in fact two other items can be picked and together give a higher value:
  - Item A: 3kg, $5 => total value 5
  - Item B: 2kg (can fit 2), $3 => total value 6
  - Knapsack max weight: 4
  - !!! You must double-check that Greedy would pick item A => check the ratios: 5/3 > 3/2 (1.66 > 1.5).
    - *If item A had value 4, Greedy would also have picked B.*
- Same example can be modified to work for **0/1 Knapsack**:
  - Item A: 3kg, $5
  - Item B: 2kg, $3
  - Item C: 2kg, $3





Weighted Interval Scheduling
(a.k.a. Job Scheduling)

Weighted Interval Scheduling
(a.k.a. Job Scheduling)

Problem - Version 1 - jobs with different values

Given n jobs where each job has a start time, finish time and value, (s_i, f_i, v_i) *select a subset of them that do not overlap and give the largest total value*.

- What would be a greedy criterion? _____
- Greedy algorithm? (Is it optimal?)

E.g.:
(start, end, value)
(6, 8, $2)
(2, 5, $6)
(3, 11, $5)
(5, 6, $3)
(1, 4, $5)
(4, 7, $2)

Problem - Version 2 - jobs with same value

- All jobs have the same value (e.g. $10).
  - The job value will be just a multiplication factor
  - Only start and finish time, (s_i, f_i), will affect the solution
- ***maximize number of non-overlapping jobs***
- What would be a greedy criterion? _____
- Greedy algorithm? (Is it optimal?)

E.g.:
(start, end)
(6, 8)
(2, 5)
(3, 11)
(5, 6)
(1, 4)
(4, 7)

Which of the 2 versions is harder?

Greedy gives an optimal solution to one of them. Can you guess which one?

Weighted Interval Scheduling
(a.k.a. Job Scheduling)

Preprocessing:

- Sort jobs in increasing order of their finish time (and give them an ID for easy reference).

- Possible criteria:
  - Ratio: value/duration
  - Largest value
  - Shortest duration
  - Starts first
  - Finishes last

  - Finishes first
  - Starts last

E.g.:
(start, end, value)
(6, 8, $2)
(2, 5, $6)
(3, 11, $5)
(5, 6, $3)
(1, 4, $5)
(4, 7, $2)

After preprocessing:
JobId (start, end)
1 (1, 4, $5)
2 (2, 5, $6)
3 (5, 6, $3)
4 (4, 7, $2)
5 (6, 8, $2)
6 (3, 11, $5)

Class work – Applying Greedy

- Both problem version

- Criterion: job that finishes first
- algorithm,
  - Sort by finish time - O(NlgN)
  - Repeat as long as there are still jobs available - (Θ(N))
    - Pick the one that finishes first, J, and
    - Remove the ones it overlaps with - Go though all remaining jobs

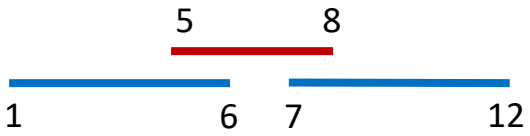
- optimal or not (if not, can we build a counter example?)
  - For VERSION 1 (different values) – not optimal
  - For VERSION 2 (same value) – Yes, optimal

Version 1:
After preprocessing:
JobId (start, end)
1 (1, 4, $5) - picked
~~2 (2, 5, $6)~~
3 (5, 6, $3)-picked
~~4 (4, 7, $2)~~
5 (6, 8, $2) - picked
~~6 (3, 11, $5) X~~

Version 2
After preprocessing:
JobId (start, end)
1 (1, 4) - picked
~~2 (2, 5)~~
3 (5, 6)-picked
~~4 (4, 7)~~
5 (6, 8) - picked
~~6 (3, 11) X~~

Interval Scheduling Greedy Criteria

- Problem version: *All jobs have the SAME value*. => maximize number of jobs you can pick.



- Criteria that gives **optimal solution**:
  - job that ***finishes first***
  - job that ***starts last***
  - (See book for proof if interested – proof by contradiction, CLRS page 415)

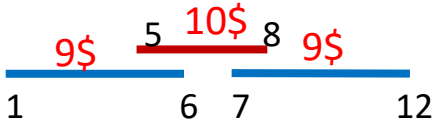
- Criteria that gives **non**-optimal solution:
  - Shortest duration
  - Least overlaps
  - Starts first
  - Finishes last

Summary and Counter Examples

- With values (job = (start, finish, value)):
  - Greedy solution – none optimal
  - DP - optimal
- Without values (or same values) (job = (start, end)):
  - Greedy solution – Some optimal, some not (based on criterion used)
    - (CLRS proof at page 418, proof of Theorem 16.1)
- Which of the two versions is more general?
  - Is one a special case (or special instance) of the other?
  - If you have a program to solve problems of one type, can you easily use it to solve problems of the other type? Which type should the program solve (with value, or without value)?

Jobs with values

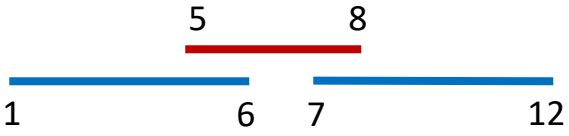
Example showing that Greedy with ***largest value*** does not give an optimal solution.



Greedy will pick the red job. Nothing else fits.
Better (optimal): the 2 blue jobs.

Jobs with the same value

Example showing that *Shortest duration*
Does not give an optimal solution.



Greedy will pick the red job. Nothing else fits.
Better (optimal): the 2 blue jobs.

Huffman code

Huffman code

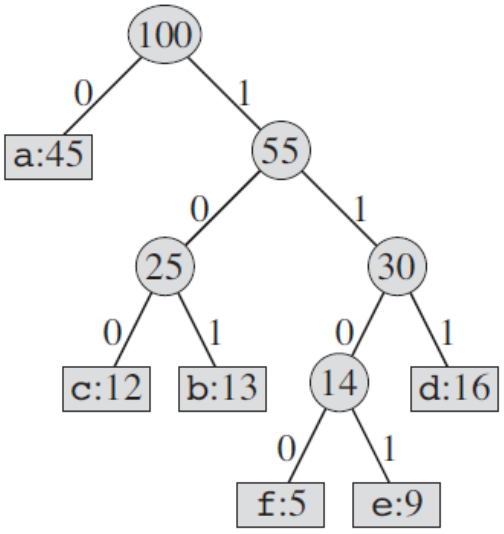
- Application: file compression
- Example from CLRS:
  - File with 100,000 characters.
  - Characters: a,b,c,d,e,f
  - Frequency in thousands (e.g. the frequency of b is 13000):
  - Goal: binary encoding that requires less memory.

	a	b	c	d	e	f	File size after encoding
Frequency (thousands)	45	13	12	16	9	5	-
Fix-length codeword	000	001	010	011	100	101	
Variable-length codeword	0	101	100	111	1101	1100	

Huffman codes

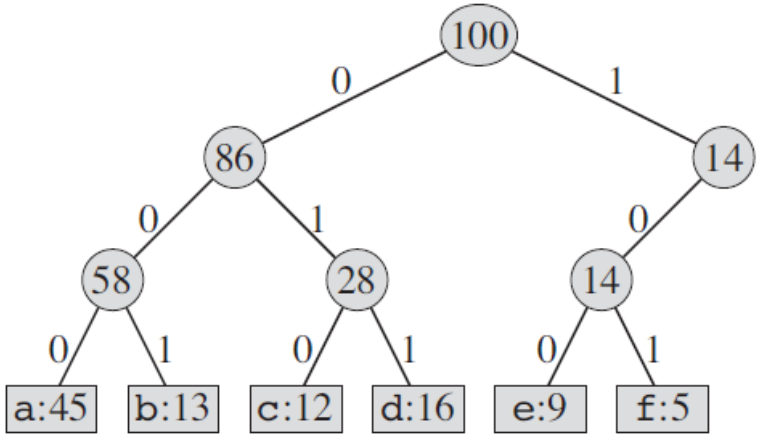
- Internal nodes contain the sum of probabilities of the leaves in that subtree.

Optimal prefix codeword tree (Huffman tree) – optimal encoding



a->0
b->101
c->100
d->111
e->1101
f->1100

Fixed-length codeword tree



Compute the number of bits needed for the whole file using each of these encodings.

Number of bits in code

45,000*1 + 13,000 * 3 + 12,000*3 + 16,000*3 + 9,000 * 4 + 5,000 * 4 = 224,000

100,000 * 3 = 300,000

(Images from CLRS.)

Building the Huffman Tree

1. Make 'leaves' with letters and their frequency and arrange them in **increasing** order of frequency.
2. Repeat until there is only one tree:
   1. Create a new tree from the two leftmost trees (with the smallest frequencies) and
   2. Put it in its place (in increasing order of frequency).
3. Label left/right branches with 0/1

Encoding of char = path from root to leaf that has that char

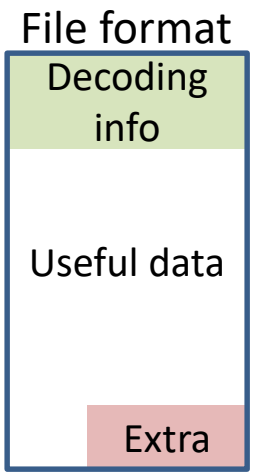
Tree property: Every internal node has two children

See book or lecture video for the step-by-step solution.
In exam or hw, you must use this method. Make sure that you always put the smallest child node to the left.

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

Glancing at implementation issues and options

- Good reference: https://www2.cs.duke.edu/csed/poop/huff/info/
- File Compression with Huffman coding - Practical Issues
  - Given an encoded file, how will you know when it ended?
    - Typically files are stored by the OS in sizes that are multiples of a specific value, so even though your actual file takes 121 bits, on disk 128 bits are written and all that will be read when you open the file. You need to know that after 121 you are done.
    - a) encode length of useful data in the file or
    - b) need a new symbol to indicate end of file. Note that you must add this symbol (with count 1) to your set of distinct symbols used in building the Huffman tree, because you must be able to recognize it (decode it from bits into the symbol).
  - Given an encoded file, how will you know how to decode it?
    - The encoding tree is not standard, but it depends on the given file =>
    - Must record some information to know how to decode the file along with the file (at the beginning) =>
      - Store the Huffman tree (0- inner node, 1-leaf, after 1 read 8 bits and decode to symbol)
      - Store enough info to regenerate the tree (e.g. char and frequency pairs).



- Build the tree efficiently
  - Using a heap: Heaps are efficient (lgN) for finding min, removing min, inserting new item
    - With heap: O(NlgN)
    - With sort and reinsert (like on paper): $O(N^2)$
  - Using 2 sorted queues (https://en.wikipedia.org/wiki/Huffman_coding)

Greedy Algorithms

- Greedy algorithms do not always guarantee optimal solution. It depends on the problem.
- Difference between Greedy and Dynamic Programming:
  - In DP typically you solve all the problems and then make your choice. You will compute two or more answers for the current problem (entire problem) and pick the best of those.
  - In greedy, you make the greedy choice and you are left with only one problem.

Problem/Greedy optimal or not	Greedy gives optimal solution	Greedy does NOT give the optimal solution
Knapsack	- 0/1 fractional, - unbounded fractional Criterion: value/weight ratio	- 0/1 Not fractional - Unbounded not fractional
Job Scheduling	- All jobs have the SAME VALUE Criterion: job that finishes first	- Jobs have different values
Hufmann	Pick the two trees with smallest weight.	-