Overview of Optimization Problems and Greedy Algorithms

When There Are Multiple Solutions To A Problem,
And We Want To Find An Optimal Solution

And Sometimes Greedy Choices Suffice

Optimization Problems

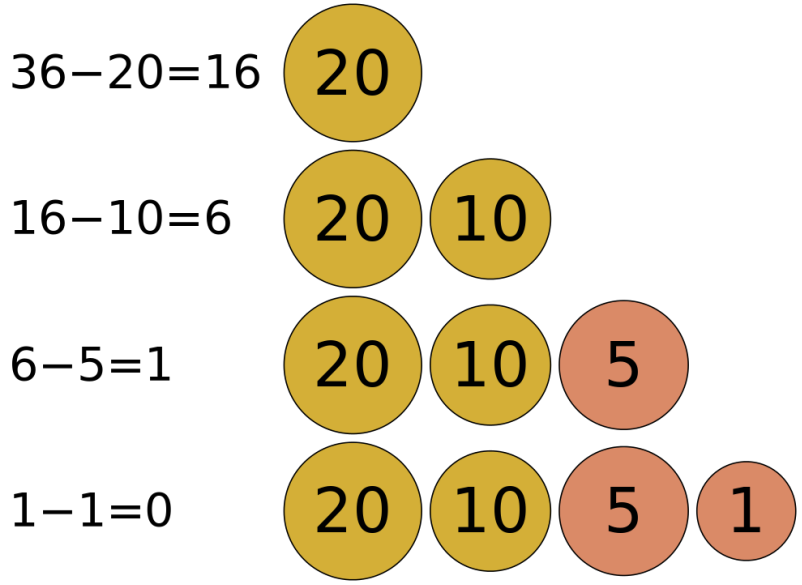
- Contrary to problems we've seen so far, there can be multiple solutions to an optimization problem.
  - E.g., making coin changes for a change amount in US currency
- Each solution has a value (cost or benefit).
  - E.g., # coins to the above problem
- We wish to find a solution with the optimal value (minimum cost or maximum benefit).
  - E.g., the way to make changes with minimum # coins
- "An" optimal solution, as opposed to "the" optimal solution
  - Because there might be multiple solutions that achieve the optimal value.

Solving An Optimization Problem

- We make a set of choices in order to arrive at an optimal solution.
- Sometimes such a choice making can be performed in a sequence.
  - E.g., which type of coin to return, one at a time.
  - Could be very many combinations of sequences that need to be examined
    - “Brute-force” search, which usually results in exponentially many trials
- Not all optimization problems exhibit such a choice sequence property.
  - E.g., maximum subarray sum problem in CLRS 4.1 we have studied in earlier module

Greedy Algorithms

- Sometimes a simple, heuristic choice at each step could give us an optimal solution
  - E.g., return the highest amount of coin type possible for the current change amount.
  - It's a "greedy" choice that looks best at the moment.
  - Makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- Algorithm is easy, but it's not easy to prove it's correct!
- Not all optimization problems can be solved this way!



https://en.wikipedia.org/wiki/Greedy_algorithm

Greedy algorithms determine minimum number of coins to give while making change. These are the steps a human would take to emulate a greedy algorithm to represent 36 cents using only coins with values {1, 5, 10, 20}. The coin of the highest value, less than the remaining change owed, is the local optimum.

Activity-Selection Problem

First Example Of An Optimization Problem With Very Simple Greedy Algorithm

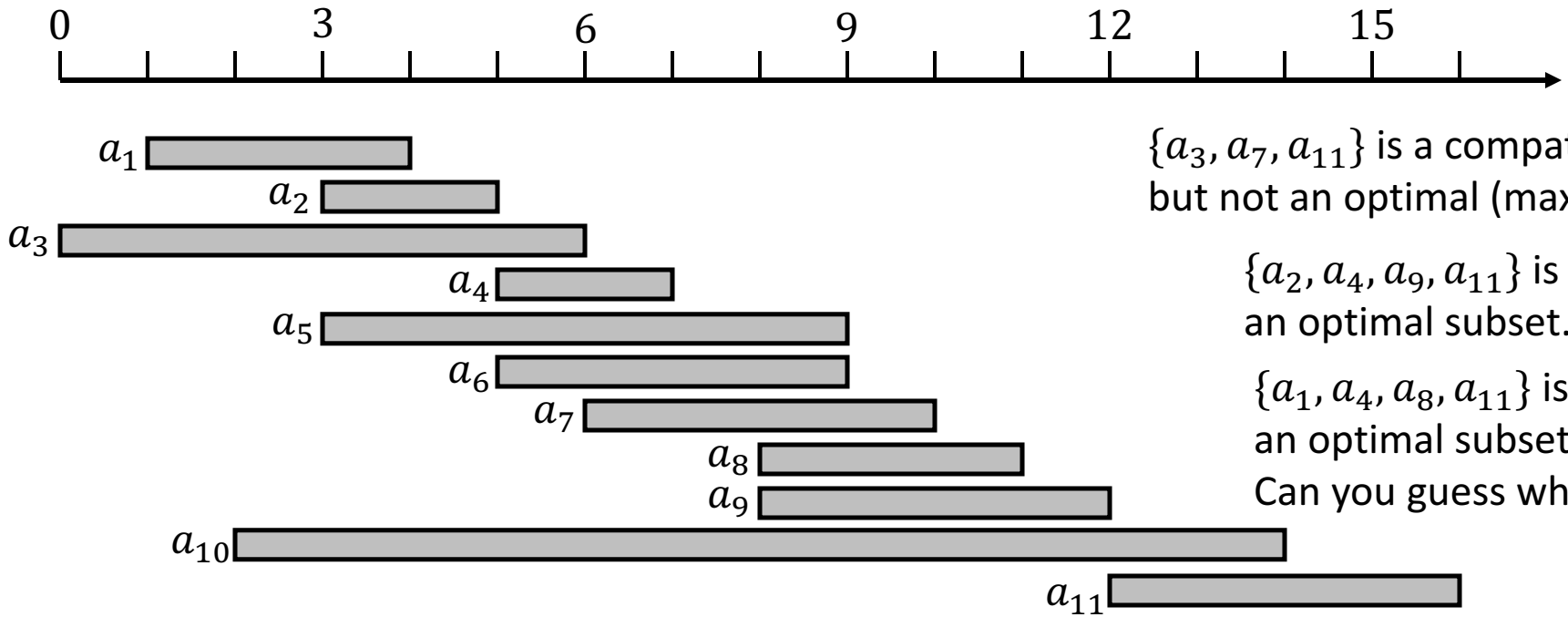
Problem Description

- Given a set S of n activities ($S = \{a_1, a_2, \ldots, a_n\}$), each a_i of which has a start time (s_i) and a finish time (f_i),
- We want to find a maximum-size subset of mutually compatible (non-overlapping) activities.
  - Rationale: These activities need to use a shared resource (e.g., lecture hall) and we want to maximize the utility of such a shared resource.
  - Note that we are maximizing the *number of activities*, *NOT the time occupied* by the chosen activities.
    - This is significant, because the greedy algorithm that'll be presented won't work for maximizing the time occupied!

Example (CLRS pp.415)

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

a_i's are sorted in monotonically increasing order of finish time for some reasons we'll see soon.



$\{a_3, a_7, a_{11}\}$ is a compatible (non-overlapping) subset, but not an optimal (maximum) subset.

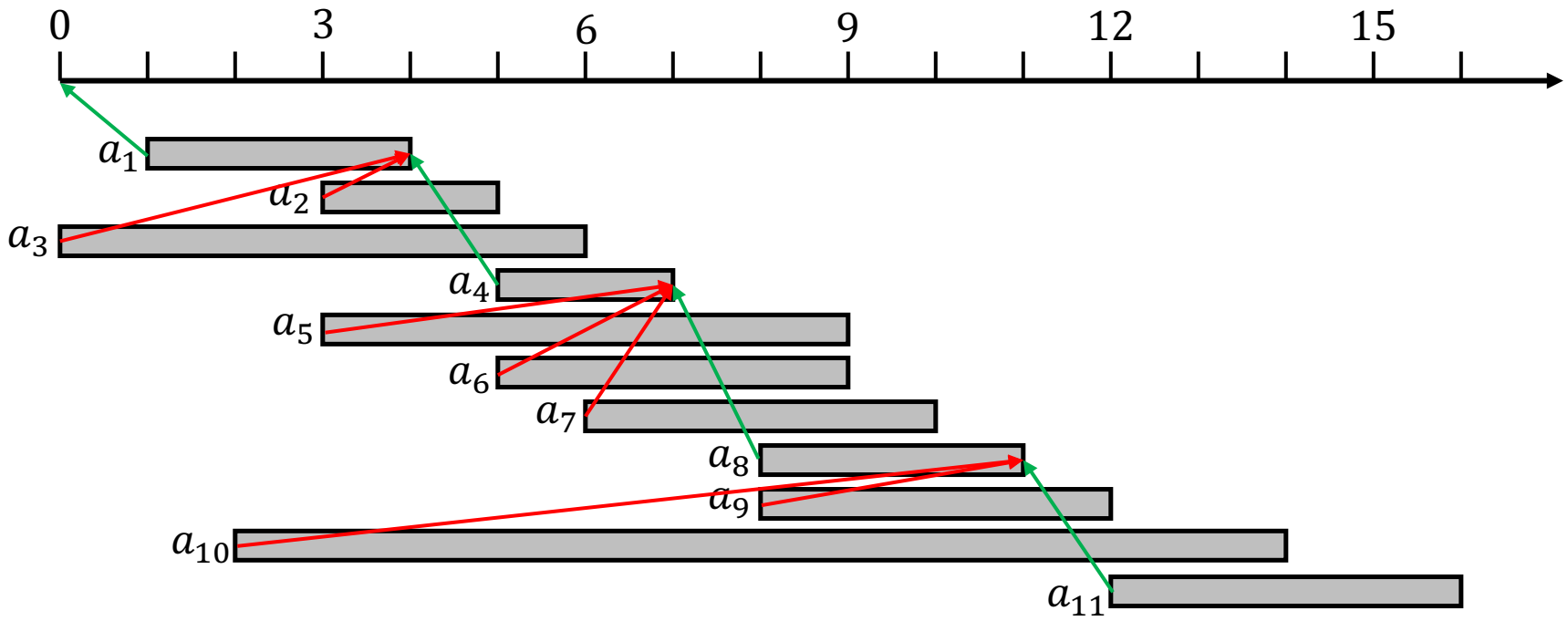
$\{a_2, a_4, a_9, a_{11}\}$ is a compatible subset, and also an optimal subset.

$\{a_1, a_4, a_8, a_{11}\}$ is a compatible subset, and also an optimal subset with somewhat greedy choices. Can you guess what's greedy (locally optimal)?

Greedy (Locally Optimal) Choices In Activity Selection Problem

- At each iteration,
  - Keep the current time as the finish time of the last activity picked.
    - It's 0 initially (no activity picked).
  - Pick the earliest finishing AND also compatible activity.
    - Of course can't pick an incompatible (overlapping) activity.
    - And pick the earliest finishing compatible activity, hoping that'll give other remaining activities more chances to be picked.
- It's straightforward to implement this idea.
  - RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n) in pp. 419 and GREEDY-ACTIVITY-SELECTOR(s, f) in pp. 421
  - Iterative implementation is better in terms of lower constant factor in $\Theta(n)$ time complexity, and constant space (memory) complexity ($\Theta(1)$) versus linear space complexity ($\Theta(n)$) of recursive implementation.

Greedy Activity Selection Visualization



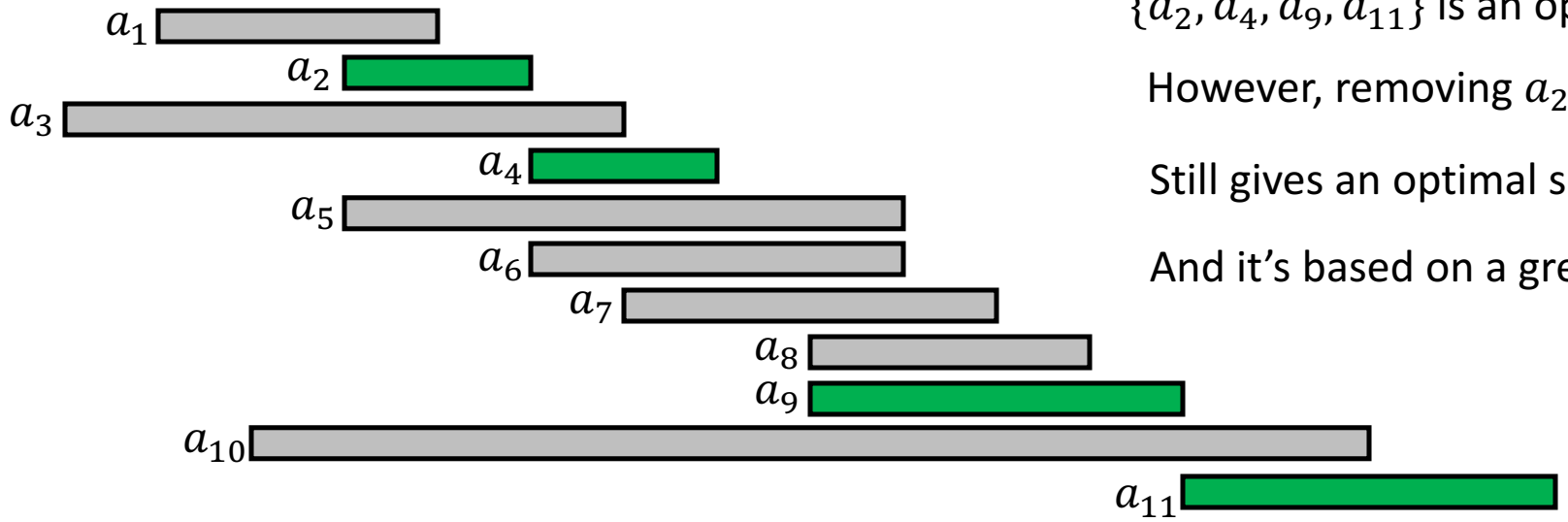
Easily obtained a solution $\{a_1, a_4, a_8, a_{11}\}$, but how can we be sure that it's optimal?

Before Proving Correctness,

- Time complexity of the greedy activity selection algorithm
- The greedy activity selection part is simply $\Theta(n)$.
  - RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n): At most one recursive call per every i
  - GREEDY-ACTIVITY-SELECTOR(s, f): For loop from 2 to n + const
- However, there's a preprocessing part.
  - Activities must be sorted in monotonically increasing order of finish time
  - This takes $\Theta(n \log_2 n)$!
- Overall, $\Theta(n \log_2 n)$, as it dominates $\Theta(n)$.

Correctness Proof Of Greedy Algorithm

- "Cut-and-paste" strategy
  - Assume an optimal solution that doesn't follow the greedy selection property
  - Show that a modified solution is also optimal. Modified in such ways that:
    - "Cut" the element in the assumed non-greedy optimal solution that satisfies the greedy selection property among the elements in the assumed optimal solution.
    - "Paste" the actual greedy selection element among all elements to the cut solution.
- E.g. with the activity selection problem example:



$\{a_2, a_4, a_9, a_{11}\}$ is an optimal solution.

However, removing a_2 and adding a_1

Still gives an optimal solution $\{a_1, a_4, a_9, a_{11}\}$,

And it's based on a greedy selection (a_1)!

Formal Proof

- Need to generalize the above observation to:
  - All cases (use a_m, a_j instead of a_1, a_2)
  - All sub-problems: Not just for the original entire problem $\{a_1, a_2, \dots, a_n\}$, but also for any sub-problem $\{a_k, a_{k+1}, \dots, a_n\}$: Called S_k in Theorem 16.1.
- Carefully read and understand proof of Theorem 16.1 in pp.418
  - And be prepared to prove yourself for other greedy algorithms
  - Note: You should be also prepared to write correct greedy activity selection algorithm pseudocode (GREEDY-ACTIVITY-SELECTOR(s, f)or RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)).

Elements of Greedy Strategy

Insights Into Greedy Algorithms

Greedy Strategy

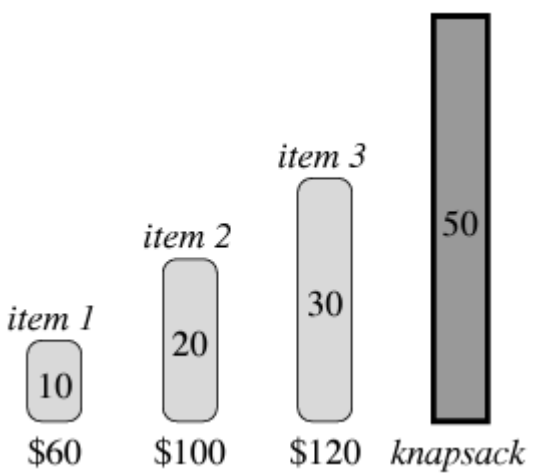
- A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices.
  - At each decision point, the algorithm makes the choice that seems best at the moment.
- Strategy
  - The optimization problem ➔ Make a choice + one sub-problem to solve.
  - Prove that there is always an optimal solution to the original problem that includes the greedy choice, so that the greedy choice is always safe.
  - Show optimal substructure:
    - The greedy choice + optimal solution to the sub-problem = optimal solution to the original problem

Proof of Greedy Choice's Correctness

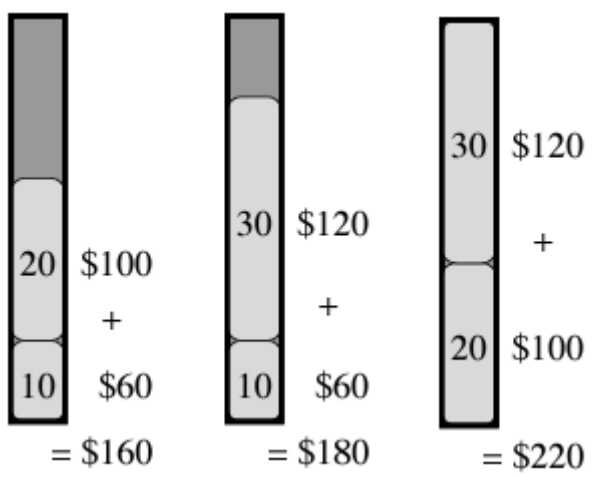
- "Cut-and-paste" strategy
  - Assume a globally optimal solution to some sub-problem.
  - Then show how to modify the solution to substitute the greedy choice for some other choice, and assert that the modified solution is still optimal to the sub-problem.
- Often preprocessing is key
  - E.g., sorting activities in finish time order.
  - Allows us to easily prove the assertion that the modified solution is still optimal.
  - Optimal substructure property is also easily proven by this kind of preprocessing.

Greedy Strategy Doesn't Always Work

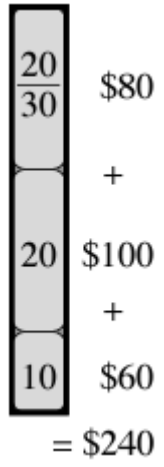
- Not all optimization problems allow correct greedy choices.
  - In fact, optimization problems that can be solved using greedy algorithms are much fewer than those that can't.
  - E.g., 0-1 knapsack vs fractional knapsack (Fig. 16.2 in CLRS pp. 427)



Items and knapsack



0-1 knapsack



Fractional knapsack

  - Give rise to "dynamic programming"

Huffman Coding Algorithm

Another Famous Greedy Algorithm For Compression

Compression Problem

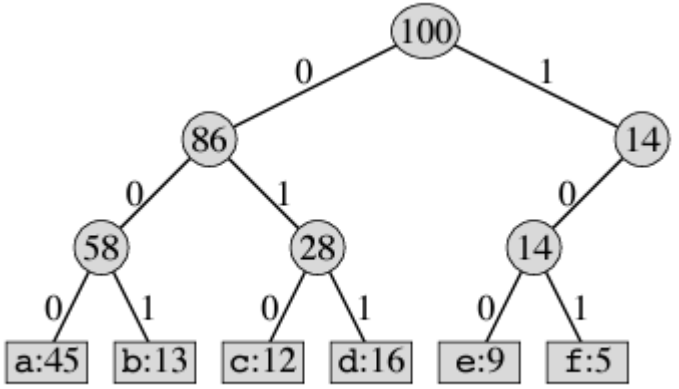
- Encoding symbols with codeword
- Fixed-length codeword: Length of every codeword is fixed
  - E.g., ASCII: A=1000001, B=1000010, …
  - Could be inefficient if some symbols occur very rarely (e.g., Z, Q, …).
  - Easy/straightforward decoding, though.
- Variable-length codeword: Codeword lengths vary.
  - We can assign shorter codeword for more frequent symbol, reducing the overall encoded text length.
    - E.g., E=0, Z=111111111111 (longer codeword for less frequent symbol unavoidable)
  - Frequency analysis is key!
  - Also important is decodability!

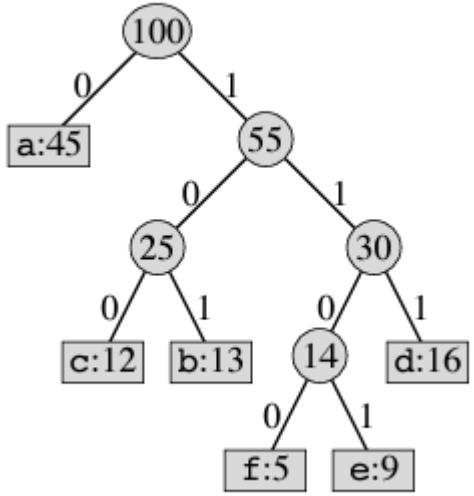
Prefix Codes

- No codeword is a prefix of some other codeword.
  - E.g., a=0, b=101, c=100, d=111, e=1101, f=1100
    - 0 is not a prefix of any of 101, 100, 111, 1101, 1100. 101 is not a prefix of any of 1101, 1100, …
  - Not prefix codes: a=00, b=01, c=001, d=010
    - 00 is a prefix of 001, 01 is prefix of 010.
- Without proof, we accept:
  - A prefix code can always achieve the optimal data compression rate among any character code.
  - Without loss of generality, we can restrict our attention to prefix codes only.

Encoding & Decoding Of Prefix Codes

- Encoding: Straightforward. Just concatenation.
- Decoding: Almost straightforward.
  - The initial codeword of any encoded file is unambiguous.
    - Because no codeword is a prefix of any other.
  - And then we keep repeating the process after translating/removing the initial codeword.
- Prefix code tree will be very useful.
  - The number in a leaf node corresponds to the symbol's frequency.
  - The number in an internal node corresponds to the sum of frequencies of all symbols in the subtree rooted at the internal node.



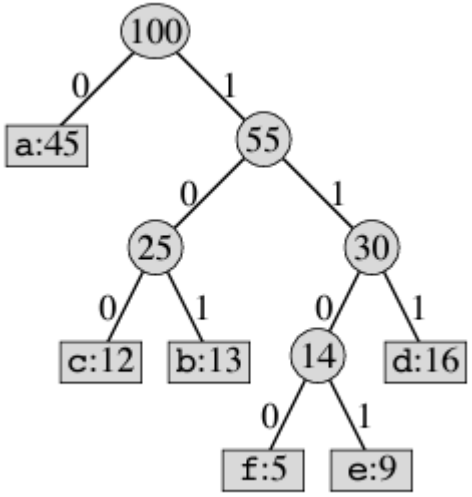


Optimal Coding (Compression) Problem

- Given a set of symbols and their frequencies in a file,
  - Find a prefix code that gives the least number of bits required to encode the file (smallest encoded/compressed file size)
- With a prefix code tree representation as in previous slide,

$$B(T) = \sum_{c \in C} freq(c) \cdot d_T(c)$$

  - T is the prefix code tree. c is each symbol (character). C is the set of all symbols.
  - $d_T(c)$ is the depth of the leaf holding c in the tree T.
  - $B(T)$ is defined as the *cost* of the tree T.
  - And problem is to find a tree with minimum cost.

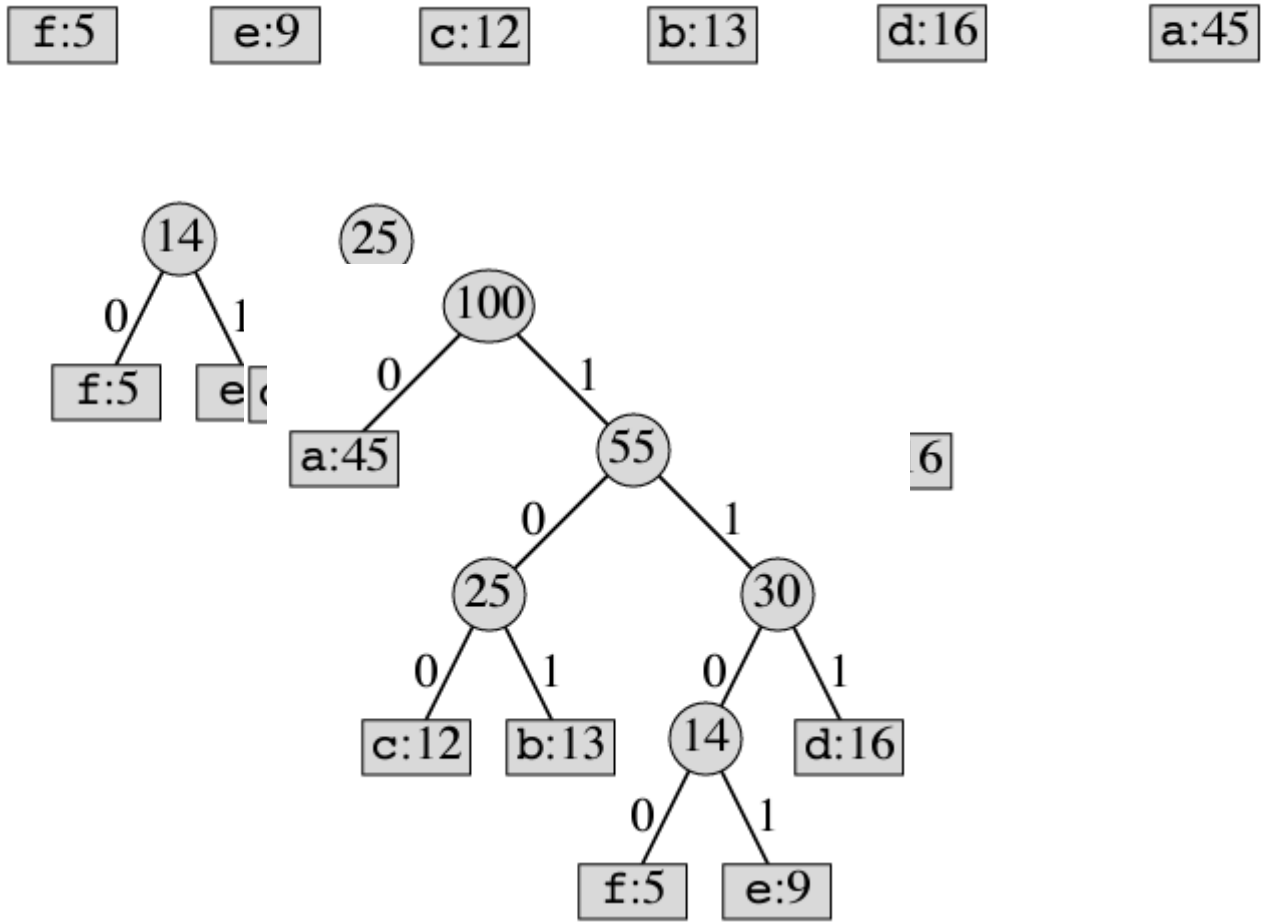


Huffman Code

- Huffman invented a *greedy* algorithm that constructs an optimal prefix code.
- “Greediness” characteristic could be a little confusing:
  - NOT: Assigning shorter codeword to more frequent symbol
  - BUT: Adding one more bit to two least frequent symbol sets
    - Any symbol set is represented as a full binary tree.
    - Adding one more bit is to take the two trees, making them as subtrees of a new root, whose frequency value is the sum of the two children’s frequency values.
  - It’s greedy in that it penalizes less frequent symbols (adding one more bit), rather than it incentivizes more frequent symbols.
    - But these are actually just the two sides of a single coin.
- Restricting to full binary tree is OK, because any optimal prefix code always results in a full binary tree (proof is an textbook exercise).

Huffman Code Construction Example

Sort frequencies into a min-heap

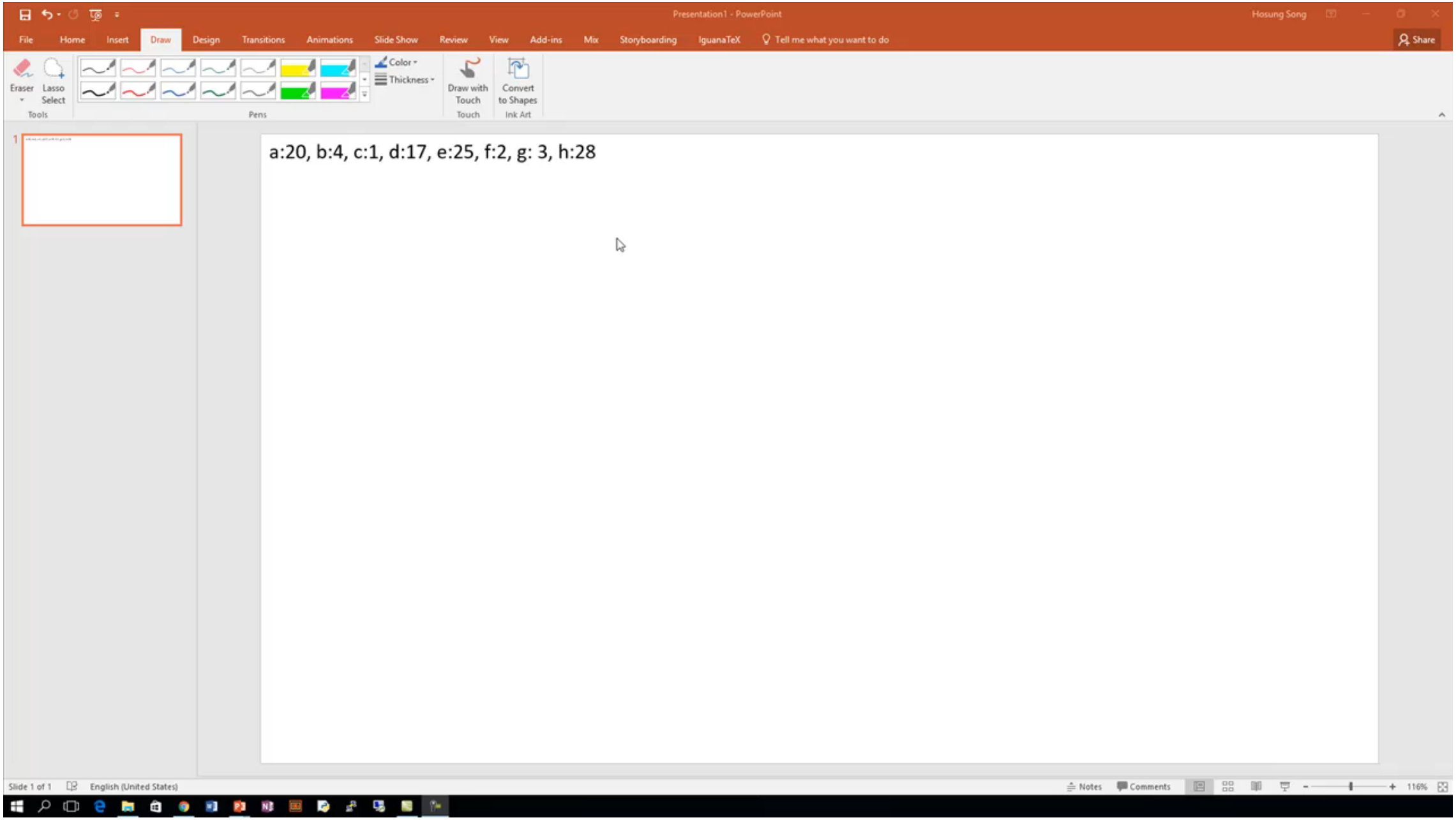


Extract two least frequency occurring subtrees

Answer is when only one tree is left in min-heap.
a (45): 0
b (13): 101
c (12): 100
d (16): 111
e (9): 1100
f (5): 1101

Assigning 0 & 1 to tree edges are not important (just different encoding), but by convention, we assign
0 to lowest frequency symbol set, and 1 to next lowest frequency symbol set.

Screencast: Huffman Code Construction



Before Proceeding To Correctness Proof

- Reminder: You should be able to write HUFFMAN(C) pseudocode in CLRS pp. 431 on your own.
- Time complexity
  - $n - 1$ iterations
  - Each iteration has 3 heap operations, each of which takes $O(\log_2 n)$.
  - Therefore, $O(n \log_2 n)$.
  - In this problem, the preprocessing step is quicker than the actual processing.
    - BUILD-MIN-HEAP() from Ch. 6, which is $O(n)$.

Correctness of Huffman's Algorithm

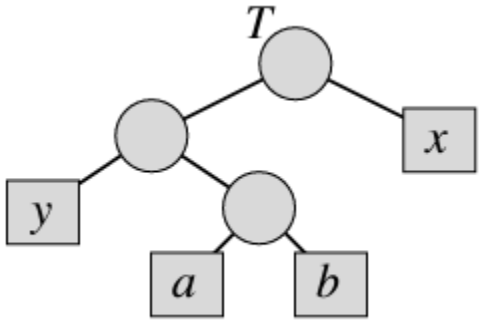
- Need to show two properties hold:
  - Greedy-choice property
    - Show that there always exists an optimal solution that corresponds to the greedy choice of the algorithm.
    - "Cut-and-paste" technique
      - Assume a general optimal solution, cut some nodes, paste greedy choice symbols, and show that it's still optimal.
  - Optimal substructure property
    - Show that the optimal solution to the sub-problem after the greedy choice is made is part of the optimal solution to the original problem.
    - Also "cut-and-paste" technique is used, but it's used for deriving a contradiction.

Greedy-Choice Property of Huffman Coding

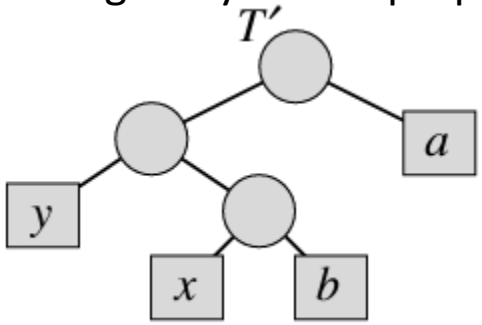
- Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies.
- Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit. (Lemma 16.2)
- Proof flow (pattern):
  - Assume a general optimal solution (T).
  - Transform T into another optimal solution T' and then T'', which satisfies the greedy choice property.

Greedy Choice Property (Lemma 16.2)

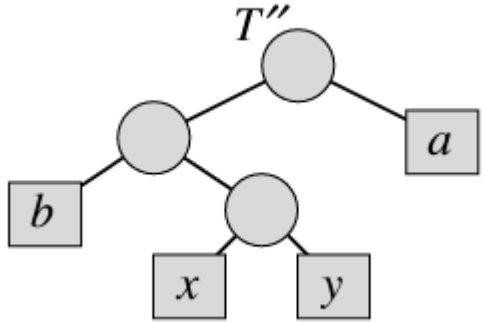
Assume a general optimal solution not satisfying the greedy choice property



Transform it closer to the desired optimal solution satisfying greedy choice property



Transform it to the final desired optimal solution satisfying greedy choice property



Show it's still optimal.

Show it's still optimal (similar to previous), and argue that this optimal solution satisfies the greedy choice property.

$$
\begin{aligned}
& B(T) - B(T') \\
&= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&= (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\geq 0 \,,
\end{aligned}
$$

Optimal Substructure Property of Huffman Coding Algorithm

- Optimal solution to the sub-problem after greedy choice is made:
  - T' for $C' = (C - \{x, y\}) \cup \{z\}$ with $z.freq = x.freq + y.freq$
- Combine it with the greedy choice:
  - T from T' by replacing z with an internal node having x and y as children
- Claim/prove: T is an optimal prefix code for C.
- Proof flow (pattern):
  - Establish relationship between T and T'.
  - Prove by contradiction: Suppose T is not an optimal prefix code for C, derive that T' is not optimal (find another tree with lower cost, using "cut-and-paste"), which is a contradiction, thus T must be optimal.

Optimal Substructure Property

- It's not hard to see $B(T) = B(T') + x.freq + y.freq$
  - x and y's depths are just z's depth + 1
- Contradiction argument: Suppose T is not optimal.
  - Then there must be an optimal tree T'' such that $B(T'') < B(T)$.
  - From earlier lemma, we can safely assume that x and y are siblings in T''.
  - Then let's replace x, y and their parent in T'' with a leaf z (whose $z.freq = x.freq + y.freq$), and call it T'''.
  - Now show that $B(T''') < B(T')$, which is a contradiction (to T' being optimal):
    - $B(T''') = B(T'') - x.freq - y.freq$
    - $< B(T) - x.freq - y.freq$
    - $= B(T')$
  - Therefore, T must be optimal!