10-0: **Dynamic Programming**

- Hallmarks of Dynamic Programming
  - Optimal Program Substructure
  - Overlapping Subproblems
- If a problem has optimal program structure, there *may* be a faster method than dynamic programming

10-1: **Greedy Algorithms**

- Always takes the step that seems best in the short run
  - Locally Optimal Choice
- With some problems, this can lead to an optimal solution
  - Globally Optimal Solution

10-2: **Greedy Algorithms**

- Matrix Chain Multiplication
  - What would the locally optimal choice be?
  - Will that lead to a globally optimal solution?

10-3: **Greedy Algorithms**

- Matrix Chain Multiplication
  - What would the locally optimal choice be?
    - Choose k to minimize just $p_{i-1}p_kp_j$
    - (Don't consider how long subproblems take)
  - Will that lead to a globally optimal solution?
    - No!
    - Left as "an exercise to the reader"
- Need to be sure that the greedy solution is correct before you use it!

10-4: **Activity Scheduling**

- n activities to schedule $S = \{a_1, a_2, \ldots, a_n\}$
- Each activity has a start time and an end time
- Two activities are compatible if their times do not overlap
- Problem: Find a maximal subset S' of S such that all activities in S' are compatible with each other

10-5: **Activity Scheduling**

- Solution
  - Sort the activities by increasing end time
  - Go through the list in order, selecting each activity that is compatible with all previously selected activities

- Why does this work?

10-6: **Proving Greedy**

- To prove a greedy algorithm is correct:
  - Greedy Choice
    - At least one optimal solution contains the greedy choice
  - Optimal Substructure
    - An optimal solution can be made from the greedy choice plus an optimal solution to the remaining subproblem
- Why is this enough?

10-7: **Activity Selection**

- Activity Selection problem:
  - Prove Greedy Choice
  - Prove Optimal Substructure

10-8: **Proving Greedy Choice**

- Let a_1 be the activity that ends first – greedy choice.
- Let S be an optimal solution to the problem.
- If S contains a_1, then we are done.

10-9: **Proving Greedy Choice**

- Let a_1 be the activity that ends first – greedy choice.
- Let S be an optimal solution to the problem.
- If S does not contain a_1:
  - Let a_k be the first activity in S. Remove a_k from S to get S'.
  - Since no activity in S' conflicts with a_k, all activities in S' must start after a_k finishes.
  - Since a_1 ends at or before when a_k ends, all activities in S' start after a_1 finishes – and a_1 is compatible with all activities in S'
  - Add a_1 to S' to get S''. $|S''| = |S|$, and hence S'' is optimal, and contains a_1

10-10: **Proving Optimal Substructure**

- Proof by contradiction: Assume no optimal solution that contains the greedy choice has optimal substructure
- Let S be an optimal solution to the problem, which contains the greedy choice
- Consider $S' = S - \{a_1\}$. S' is not an optimal solution to the problem of selecting activities that do not conflict with a_1
- Let S'' be an optimal solution to the subproblem of picking activities that do not conflict with a_1.

- Consider $S''' = S'' \cup \{a_1\}$. S''' is a valid solution to the problem, $|S'''| = |S''| + 1 > |S'| + 1 = |S|$ (since S' is not optimal).
- S is thus not optimal, a contradiction

10-11: **Proving Optimal Substructure**

- Proof by contradiction: Assume no optimal solution that contains the greedy choice has optimal substructure
- Let S be an optimal solution to the problem, which contains the greedy choice

...

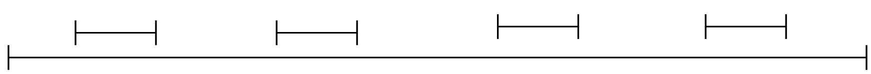
- S is thus not optimal, a contradiction

10-12: **Activity Scheduling**

- WARNING: Just because there is a greedy algorithm that leads to an optimal solution does not mean that *all* greedy solutions lead to an optimal solution
  - Picking the activity with the earliest start time can lead to a non-optimal solution

10-13: **Activity Scheduling**

- WARNING: Just because there is a greedy algorithm that leads to an optimal solution does not mean that *all* greedy solutions lead to an optimal solution
  - Picking the activity with the earliest start time can lead to a non-optimal solution

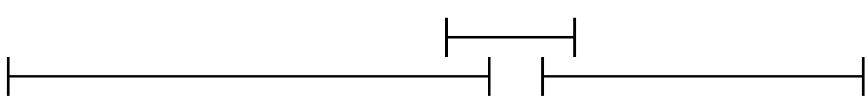


10-14: **Activity Scheduling**

- WARNING: Just because there is a greedy algorithm that leads to an optimal solution does not mean that *all* greedy solutions lead to an optimal solution
  - Picking the activity with the shortest duration can lead to a non-optimal solution

10-15: **Activity Scheduling**

- WARNING: Just because there is a greedy algorithm that leads to an optimal solution does not mean that *all* greedy solutions lead to an optimal solution
  - Picking the activity with the shortest duration can lead to a non-optimal solution

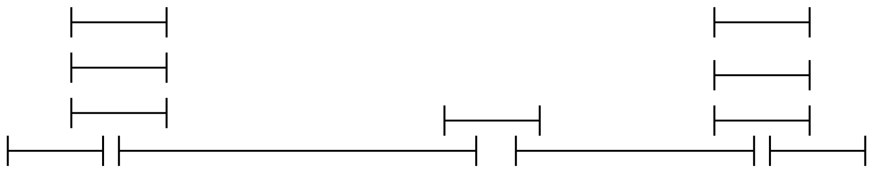


10-16: **Activity Scheduling**

- WARNING: Just because there is a greedy algorithm that leads to an optimal solution does not mean that *all* greedy solutions lead to an optimal solution
  - Picking the activity with the smallest # of conflicts can lead to a non-optimal solution

10-17: **Activity Scheduling**

- WARNING: Just because there is a greedy algorithm that leads to an optimal solution does not mean that *all* greedy solutions lead to an optimal solution
  - Picking the activity with the smallest # of conflicts can lead to a non-optimal solution



10-18: **Greedy Algorithms**

- Dynamic vs. Greedy
  - It can sometimes be difficult to tell when a Greedy Algorithm can be used, and when Dynamic Programming must be used
  - Subtle changes in a problem can kill greedy choice

10-19: **Knapsack Problem**

- Thief has a knapsack (backpack) that can hold k pounds
- n elements, each of which has a value and a weight
- Add items to the backpack to maximize total value
  - What are some greedy solutions?
  - Do they produce optimal solutions?

10-20: **Knapsack Problem**

- Pick most densely valued items first:

  Knapsack holds 100 pounds

Weight	Value	Value / Weight
60	70	7/6
50	50	1
45	45	1

- No other greedy algorithm works, either

10-21: **Fractional Knapsack**

- Thief has a knapsack (backpack) that can hold k pounds
- n elements, each of which has a value and a weight
- Add items to the backpack to maximize total value
  - This time you can take a fraction of any item
  - Like gold dust
- Is there a greedy algorithm for this problem? Can you prove it?

10-22: **0-1 Knapsack Problem**

- Standard version of the knapsack problem
  - Can't take fractional items
- Order of elements by increasing weight = order by decreasing value
- Is there a valid greedy algorithm for this problem?

10-23: **Driving Problem**

- Need to get across the country in a car
  - Gas tank holds enough gas for n miles
  - Have a chart with location of all gas stations on it
  - Want to make as few stops as possible
- How do we decide which stations to stop at?

10-24: **Job Scheduling**

- Series of jobs to execute on a uniprocessor machine
- Each job takes a different amount of time to complete
  - $j_1, j_2, \ldots, j_n$
- Want to minimize the average wait time
  - Same as minimizing the total wait time (why?)
- Algorithm?
- Correctness Proof?

10-25: **Huffman Coding**

- Standard encoding (ASCII)
  - Each letter uses the same number of bits
- We'd like to use fewer bits for more common letters, more bits for less common letters
  - Use less space overall for the file

10-26: **Huffman Coding**

- If different letters use a different # of bits, how do we determine which bits go with which letter?

10-27: **Huffman Coding**

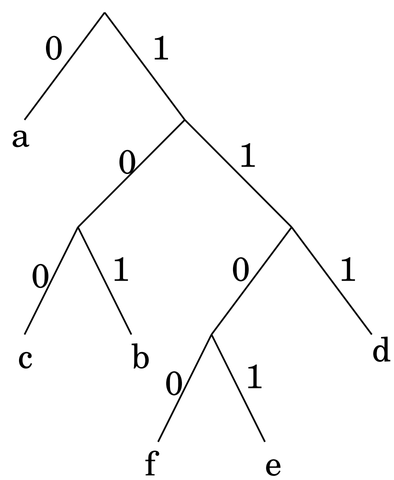
- If different letters use a different # of bits, how do we determine which bits go with which letter?
- Prefix Codes
  - No code is a prefix of any other code
  - Decoding is unambiguous

10-28: **Huffman Coding**

	a	b	c	d	e	f
Frequency	43K	12K	12k	16k	9k	5k
Fixed-Length	000	001	010	011	100	101
Variable-Length	0	101	100	111	1101	1100

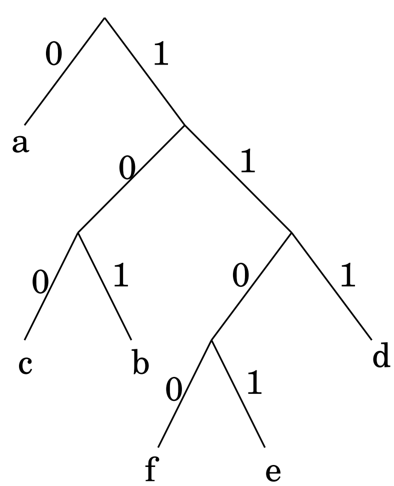
Input	Fixed-Length	Variable-Length
abc	000001010	0101100
fee	101100100	1100110111011
aaba	000000001000	001010

10-29: **Huffman Coding**



- abaac
- 11010010111000100

10-30: **Huffman Coding**



- abaac ⇒ 010100100
- 11010010111000100 ⇒ eaabfac

10-31: **Huffman Coding**

	a	b	c	d	e	f
Frequency	43K	12K	12k	16k	9k	5k
Fixed-Length	000	001	010	011	100	101
Variable-Length	0	101	100	111	1101	1100

- Total size of file in fixed-length encoding: 300K bits
- Total size of file in variable-length encoding: 224k bits

10-32: **Huffman Coding**

- Are fixed-length codes prefix codes?
  - Can we form a binary tree for fixed-length codes?
- What is the cost of a tree T for a specific file (given the frequency $f[c]$ of each character c in the file)?

10-33: **Huffman Coding**

- Are fixed-length codes prefix codes?
  - Can we form a binary tree for fixed-length codes?
- What is the cost of a tree T for a specific file (given the frequency $f[c]$ of each character c in the file)?

$$B(T) = \sum_{c \in T} f[c] * d_T(c)$$

($d_T(c)$ is the depth of the character c in the tree T) 10-34: **Huffman Coding**

- Build a tree to minimize $B(T) = \sum_{c \in T} f[c] * d_T(c)$
  - Create set of trees: one for each character in the input file
    - Each tree has a single node w/ character & frequency information
  - While > 1 tree in the set:
    - Take the two trees with the smallest frequency, t_1, t_2
    - Create a new root, with t_1 and t_2 as subtrees
    - $f[root] = f[t_1] + f[t_2]$

Letter	a	b	c	d	e	f
Frequency	3	7	40	20	15	13

10-35: **Huffman Coding**

- Do Huffman codes produce optimal trees?
  - Greedy Choice
  - Optimal Substructure

10-36: **Huffman Coding**

- Greedy Choice
  - Optimal tree T
  - Alphabet C, $f[c]$ = frequency of $c \in C$

- x, y two characters in C with lowest frequency
- a, b lowest-depth siblings in T
- Swap a with x, and b with y, to get T'

10-37: **Huffman Coding**

$$\begin{aligned}
B(T) - B(T') &= \sum_{c \in T} f[c] * d_T(c) - \sum_{c' \in T'} f[c']d_{T'}(c') \\
&= f[a](d_T(a) - d_{T'}(a)) + f[b](d_T(b) - d_{T'}(b)) \\
&\quad + f[x](d_T(x) - d_{T'}(x)) + f[y](d_T(y) - d_{T'}(y)) \\
&= f[a](d_T(a) - d_{T'}(a)) + f[x](d_T(x) - d_{T'}(x)) \\
&\quad + f[b](d_T(b) - d_{T'}(b)) + f[y](d_T(y) - d_{T'}(y)) \\
&= (f[a] - f[x])(d_T(a) - d_{T'}(a)) \\
&\quad + (f[b] - f(y))(d_T(b) - d_{T'}(b)) \\
&\geq 0
\end{aligned}$$

- $B(T') \leq B(T)$
- If T is optimal, T' is, too

10-38: **Huffman Coding**

- Optimal Substructure
  - Let T be optimal tree
  - x, y sibling nodes in T, z is the parent
  - Consider z to be a character with frequency $f[x] + f[y]$
  - $T' = T - \{x, y\}$ is an optimal prefix code for $C' = C - \{x, y\} \cup \{z\}$
  - Cost $B(T)$ in terms of cost $B(T')$:

10-39: **Huffman Coding**

- Cost $B(T)$ in terms of cost $B(T')$:
  - $\forall c \in C - \{x, y\}, d_T(c) = d_{T'}(c)$, so $f[c]d_T[c] = f[c]d_{T'}(c)$

$$\begin{aligned}
f[x]d_T(x) + f[y]d_T[y] &= (f[x] + f[y])(d_{T'}(z) + 1) \\
&= f[z]d_{T'}(z) + f[x] + f[y]
\end{aligned}$$

- $B(T) = B(T') + f[x] + f[y]$
- So, if T' is not optimal, neither is T

10-40: **Matroids**

- Matriod is a pair: $M = (S, I)$
  - S is a finite, nonempty set
  - I is a nonempty family of subsets of S, called “Independent subsets” of S such that:

- if $B \in I$ and $A \subseteq B$, then $A \in I$
  (Hereditary Property)
- If $A \in I$ and $B \in I$ and $|A| < |B|$, there is some element $x \in B$ such that $A \cup \{x\} \in I$
  (Exchange Property)

10-41: **Matroids**

- Originally, Matroids used to describe matrices
  - S = rows of a matrix
  - I = sets of linearly independent rows
    - Hence the name, independent subsets
  - Matrix matroids have both hereditary and exchange properties

10-42: **Example Matroids**

- S = edges of an undirected graph G
- I = Subsets of S that do not form a directed cycle

(Examples on board)

10-43: **Example Matroids**

- Undirected graphs / I = acyclic subsets
  - Hereditary property

10-44: **Example Matroids**

- Undirected graphs / I = acyclic subsets
  - Hereditary property
    - Trivial
    - If a graph is acyclic, any subset of edges will also be acyclic

10-45: **Example Matroids**

- Undirected graphs / I = acyclic subsets
  - Exchange Property
    - $A, B \in I$, $|A| < |B|$
    - A is a forest of $|V| - |A|$ trees (why?)
    - B is a forest of $|V| - |B|$ trees
    - Must be some edge in B that spans two different trees in A (why?)

10-46: **Weighted Matroids**

- Weighted Matroid:
  - Positive weight $w(x)$ for each element $x \in S$
  - Weight of any member of I is sum of weights of elements of I
  - Optimal subset of S is an element of I with maximal weight

- Problem: Find an optimal subset of S
  - What would greedy solution look like?
  - Does it work?

10-47: **Weighted Matroids**

Greedy(M, w)
  $A \leftarrow \{\}$
  sort $S[M]$ in non-increasing order by w
  for each $x \in S[M]$ (in non-decreasing order)
    if $A \cup \{x\} \in I[M]$
      $A \leftarrow A \cup \{x\}$
  return A

10-48: **Weighted Matroids**

- To show that a greedy algorithm is correct (produces optimal solutions) we need to show:
  - Greedy Choice
    - There exists a solution that contains the greedy choice
  - Optimal Substructure
    - Optimal solutions are composed of optimal solutions to subproblems

10-49: **Weighted Matroids**

- Greedy Choice
  - Let $\{x\}$ be independent element with largest weight
  - Show that there is some maximal matroid that contains x.
- What should we do?

10-50: **Weighted Matroids**

- Let $\{x\}$ be independent element with largest weight
- Let B be a maximal matroid
  - If B contains x, we are done
  - If B does not contain x, we can create a set A:
    - start with $A = \{x\}$
    - Use exchange property to add elements to A from b until $|A| = |B|$
    - weight(A) = weight(B) - weight(y) + weight(x)
      - y is element of B not added to A
      - weight(x) $\geq$ weight(y) (why?)

10-51: **Weighted Matroids**

- Optimal substructure

- Let x be first element chosen by Greedy from $M = (S, I)$
- Remaining subproblem: find maximal weight indep. subset of $M' = (S', I')$:
  - $S' = \{y \in S : \{x, y\} \in I\}$
  - $I' = \{B \subseteq S - \{x\} : B \cup \{x\} \in I\}$

10-52: **Weighted Matroids**

- If an optimization problem is finding a maximal weighted matroid, then greedy will work.
- Minimum Cost Spanning Tree (MST)
  - Undirected graph G, each edge k has a positive weight w_k
  - Find a spanning tree (connected, acyclic subset of edges) that has minimum cost
- Is the MST problem a maximal weighted matroid problem?

10-53: **Weighted Matroids**

- If an optimization problem is finding a maximal weighted matroid, then greedy will work.
- Minimum Cost Spanning Tree (MST)
  - Undirected graph G, each edge k has a positive weight w_k
  - Find a spanning tree (connected, acyclic subset of edges) that has minimum cost
- Is the MST problem a weighted matroid?
  - Want to find minimal total weight, not maximal
  - Replace each weight w_k with $w_0 - w_k$, where w_0 is larger than any weight on the graph
- Greedy solution will work (Kruskal's algorithm)

10-54: **Weighted Matroids**

- Example: Unit tasks with deadlines and penalties
  - Set $S = \{a_1, a_2, \ldots, a_n\}$ of n unit-time tasks
  - Set of n deadlines $d_1, \ldots d_n$
  - Set of n non-negative penalties $w_1, w_2, \ldots, w_n$
- Schedule all n tasks. Each task a_k that is completed after time d_k incurs penalty w_k.
- What is the optimal schedule (smallest overall penalty)?

10-55: **Weighted Matroids**

- Example: Unit tasks with deadlines and penalties
  - Any schedule can be re-arranged so that:
    - All on-time tasks are scheduled before all late tasks
    - On-time tasks are completed by order of deadline
  - To create a schedule, decide which tasks will be done on time, and which will be late. Then, order early tasks by increasing deadline, and late tasks afterwards in any order.

10-56: **Weighted Matroids**

- Example: Unit tasks with deadlines and penalties
  - S = set of tasks
  - I = set of subsets of tasks, where all tasks in I are early
- Hereditary Property?
- Exchange Property?

10-57: **Weighted Matroids**

- Example: Unit tasks with deadlines and penalties
  - S = set of tasks
  - I = set of subsets of tasks, where all tasks in I are early
- Hereditary Property
  - If we can schedule all elements in I on time, we can obviously schedule all elements of any subset of I in time as well.

10-58: **Weighted Matroids**

- Exchange Property
  - Let A and B be independent subsets, with $|B| > |A|$.
  - $N_T(A)$ be the number of tasks in A that have a deadline if t or earlier
  - Let k be the largest integer such that $N_k(B) \leq N_k(A)$
    - $N_0(B) = N_0(A) = 0$, so such a k must exist
  - $N_n(B) = |B|$, $N_n(A) = |A|$, so $N_n(B) > N_n(A)$
  - $k < n$, for all j in the range $k + 1 \ldots n$, $N_j(B) > N_j(A)$.
  - B contains more tasks with deadline $k + 1$ than A does
  - Add any task with deadline $k + 1$ to A from B