UNIT 5
GREEDY ALGORITHMS

Iris Hui-Ru Jiang
Spring 2024

Department of Electrical Engineering
National Taiwan University

Outline

- Content:
  - Activity selection (Interval scheduling)
  - Elements of the greedy strategy
  - Knapsack problem
  - Huffman codes
  - Task scheduling
  - Minimum spanning trees (detailed in graph algorithms)
- Reading:
  - Chapter 15

Greedy Algorithms

- An algorithm is greedy if it builds up a solution in small steps, making the choice that looks best at each step to optimize some underlying criterion

- It's easy to invent greedy algorithms for almost any problem
  - Intuitive and fast
  - Usually not optimal

- It's challenging to prove greedy algorithms succeed in solving a nontrivial problem optimally
  - An exchange argument

Interval Scheduling

Activity selection

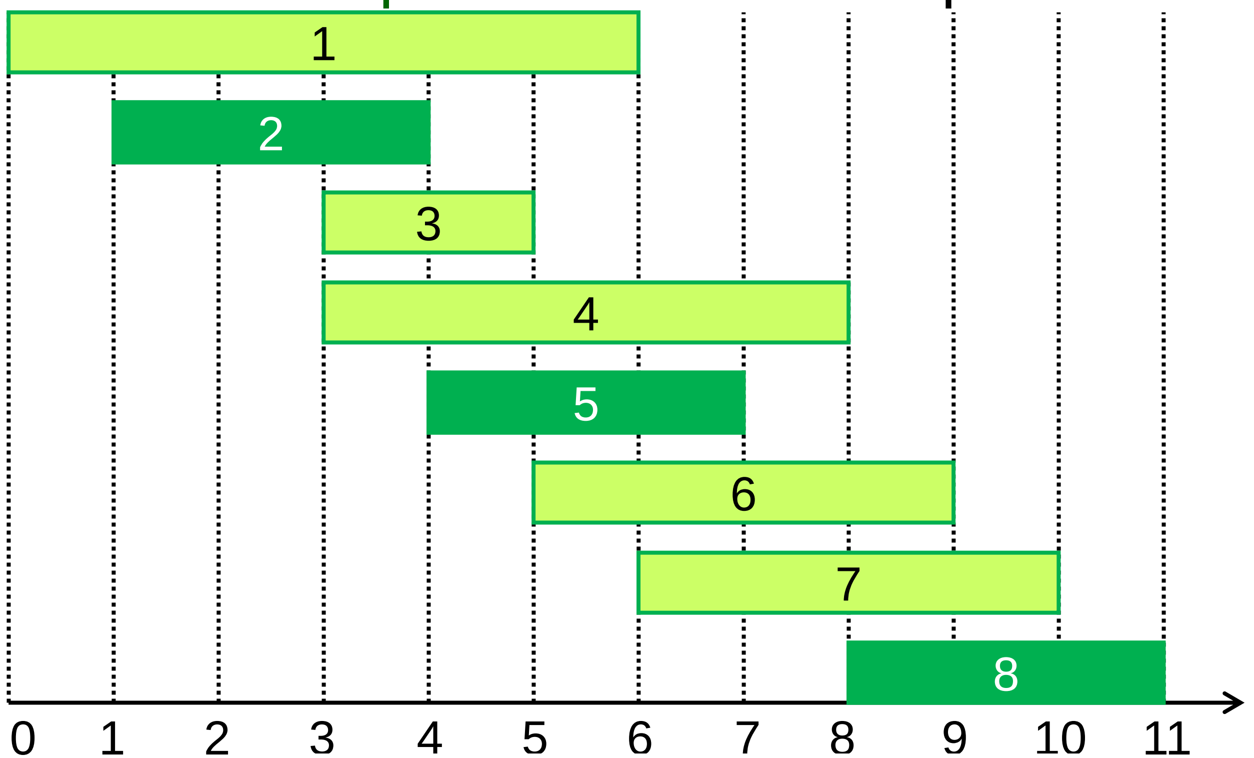
The Interval Scheduling Problem

- Given: Set of requests {1, 2, …, n}, i^{th} request corresponds to an interval with start time $s(i)$ and finish time $f(i)$
  - interval i: $[s(i), f(i))$

requests don't overlap

optimal

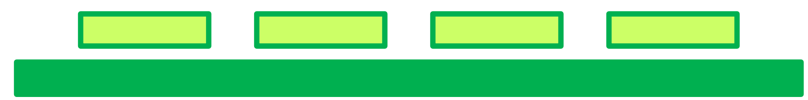
- Goal: Find a compatible subset of requests of maximum size



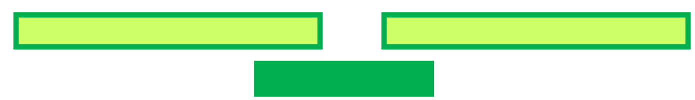
Maximum compatible subset {2, 5, 8}

Greedy Rule (Choice)

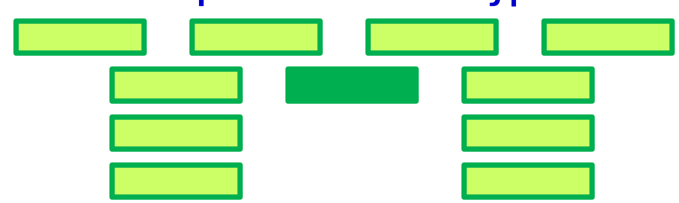
- Repeat
  - Use a simple rule to select a first request i_1
  - Once i_1 is selected, reject all requests incompatible with i_1
- Until run out of requests
- Q: How to decide a greedy rule for a good algorithm?
- A:
  1. Earliest start time: min $s(i)$



  2. Shortest interval: min $\{f(i) - s(i)\}$



  3. Fewest conflicts: $\min_{i=1..n} |\{j: j \text{ is not compatible with } i\}|$



  4. Earliest finish time: min $f(i)$

The Greedy Algorithm

- The 4^{th} greedy rule leads to the optimal solution
  - We first accept the request that finishes first
  - Natural idea: Free resource ASAP
- The greedy algorithm:

$\emptyset$: empty set = {}

Interval-Scheduling(R)
// R: undetermined requests; A: accepted requests
1. $A = \emptyset$
2. **while** (R is not empty) **do**
3. choose a request $i \in R$ with minimum $f(i)$ // greedy rule
4. $A = A + \{i\}$
5. $R = R - \{i\} - X$, where $X = \{j: j \in R$ and j is not compatible with $i\}$
6. **return** A

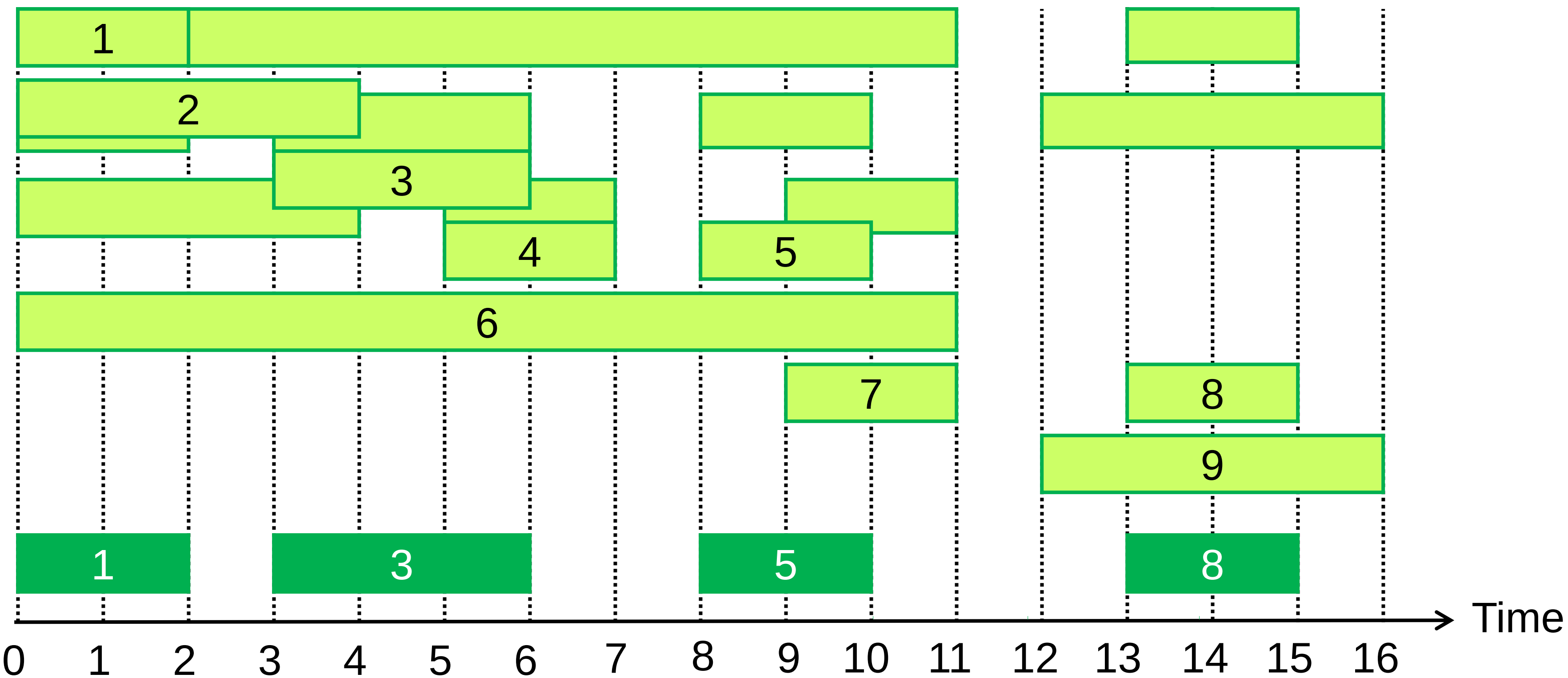
  - Q: Feasible?
  - A: Yes! Line 5
    - A is a compatible set of requests
  - Q: Optimal? Efficient?

Implementation

```
Greedy-Activity-Selector(s, f)
// Assume f_1 ≤ f_2 ≤ … ≤ f_n
n = s.length
A = {1}
k = 1
for i = 2 to n
    if s_i ≥ f_k
        A = A ∪ {i}
        k = i
return A
```

- Time complexity excluding sorting: $O(n)$
- **Theorem:** Algorithm Greedy-Activity-Selector produces solutions of maximum size for the activity-selection problem

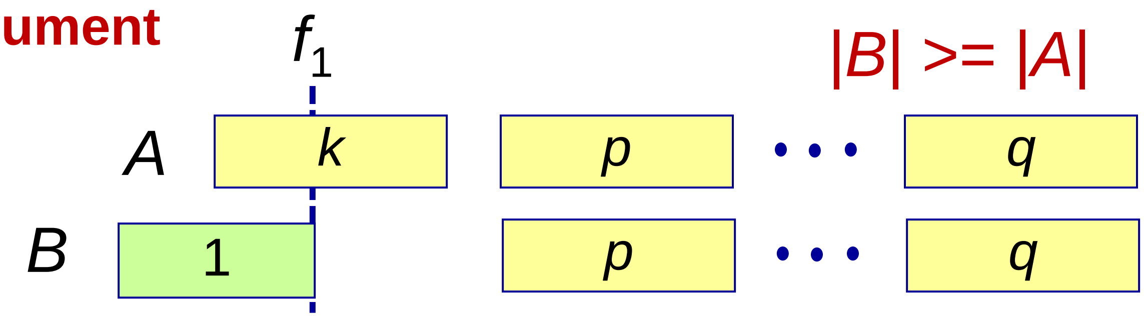
The Interval Scheduling Algorithm



Maximum compatible subset {1, 3, 5, 8}

Optimality Proofs

- **Greedy-choice property**: Suppose $A \subseteq S$ is an optimal solution. Show that if the first activity in A activity $k \neq 1$, then $B = A - \{k\} \cup \{1\}$ is an optimal solution.
  - **Exchange argument**



- **Optimal substructure**: If A is an optimal solution to S, then $A' = A - \{1\}$ is an optimal solution to $S' = \{i \in S: s_i \geq f_1\}$.
  - Exp: $A' = \{3, 5, 8\}$, $S' = \{3, 4, 5, 7, 8, 9\}$ in the previous slide
  - **Proof by contradiction**: If A' is not an optimal solution to S', we can find a “better” solution A'' (than A'). Then, $A'' \cup \{1\}$ would be a better solution than $A' \cup \{1\} = A$ to S, contradicting to the original claim that A is an optimal solution to S. (Activity 1 is compatible with all the tasks in A''.)

Elements of the Greedy Strategy

- When to apply greedy algorithms?
  - **Greedy-choice property:** A globally optimal solution can be arrived at by making a locally optimal (greedy) choice
    - Dynamic programming needs to check the solutions to subproblems
    - **Exchange argument**
  - **Optimal substructure:** An optimal solution to the problem contains within its optimal solutions to subproblems
    - E.g., if A is an optimal solution to S, then $A' = A - \{1\}$ is an optimal solution to $S' = \{i \in S: s_i \geq f_1\}$
    - **Proof by contradiction**
- Greedy ***heuristics*** do not always produce optimal solutions

Knapsack Problem

Greedy algorithms vs. dynamic programming

Greedy vs. Dynamic Programming

- Common: optimal substructure
- Difference: greedy-choice property

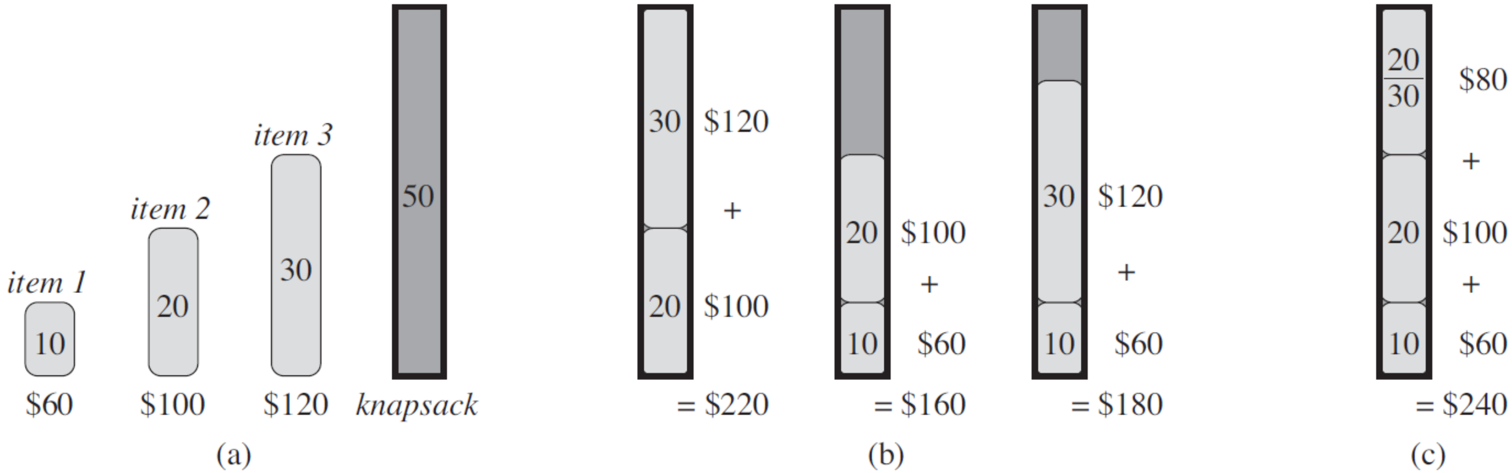
- One subproblem for greedy vs. several subproblems for DP

- Top down for greedy vs. bottom up for DP

- Beneath every greedy algorithm, there is always a more cumbersome DP solution

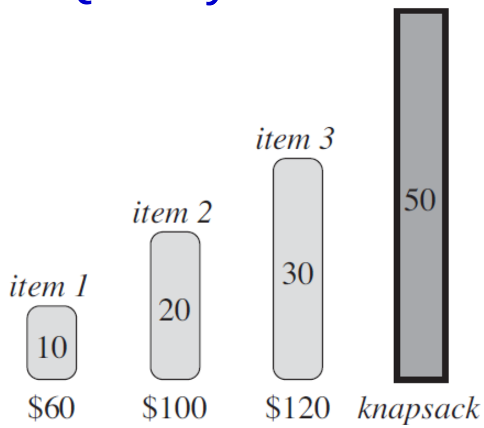
Knapsack Problem

- **Knapsack Problem:** Given n items, with i^{th} item worth v_i dollars and weighing w_i pounds, a thief wants to take as valuable a load as possible, but can carry at most W pounds in his knapsack
- **0-1 knapsack problem:** Each item is either taken or not
- **fractional knapsack problem:** Can take fraction of items
- **Ex: v** = (60, 100, 120), **w** = (10, 20, 30), W = 50

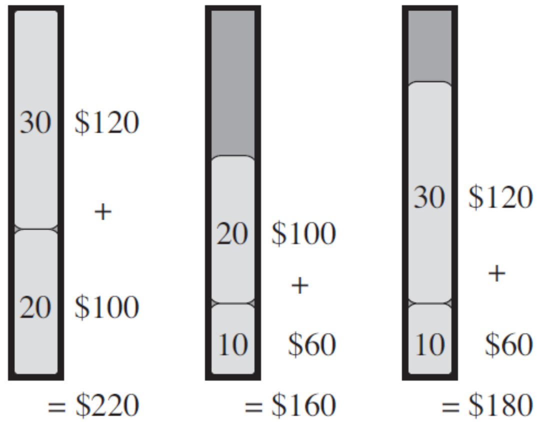


Greedy or DP?

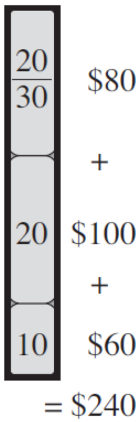
- Both problems exhibit the optimal substructure property
  - 0-1: If optimal solution includes item j, the remaining load must be the most valuable load weighing at most W-w_j (Fractional: DIY)
- Fractional version can be optimally solved by greedy algorithm, taking items in order of greatest value per pound
  - Does not work for the 0-1 version
- 0-1 knapsack problem can be solved in $O(nW)$ time by DP
  - Q: Polynomial time?



Item 1 has greatest value per pound



For 0-1 version, any solution with item 1 is not optimal!



Greedy alg. is optimal for fractional version

Huffman Codes

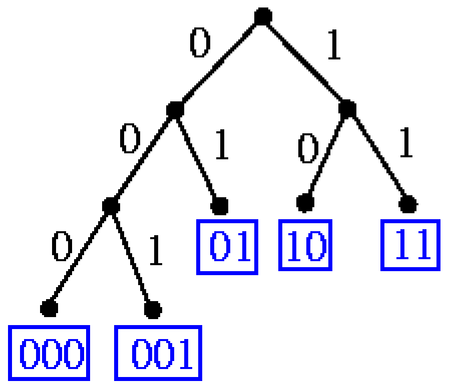
Coding

- Is used for data compression, instruction-set encoding, etc.
- **Binary character code:** character is represented by a unique binary string
  - **Fixed-length code (block code):** 3 bits for 6 characters
    a: 000, *b*: 001, …, *f*: 101 ⇒ *ace* ↔ <u>000</u> <u>010</u> <u>100</u>
  - **Variable-length code:** frequent characters ⇒ short codeword; infrequent characters ⇒ long codeword

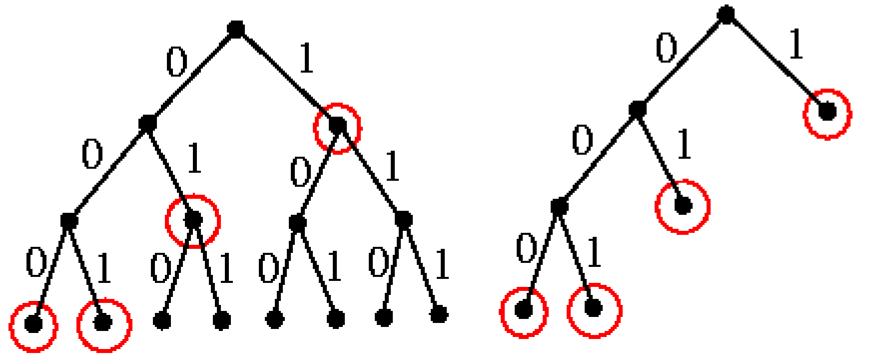
	a	b	c	d	e	f	cost / 100 characters
Frequency	45	13	12	16	9	5	
Fixed-length codeword	000	001	010	011	100	101	300
Variable-length codeword	0	101	100	111	1101	1100	224

Binary Tree vs. Prefix Code

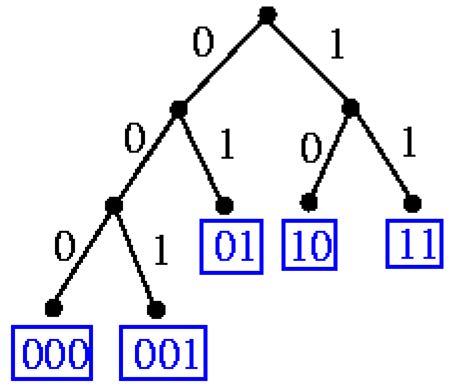
- **Prefix code:** No code is a prefix of some other code
  - Unique representation



binary tree –> prefix code



prefix code {1, 01, 000, 001}–> binary tree

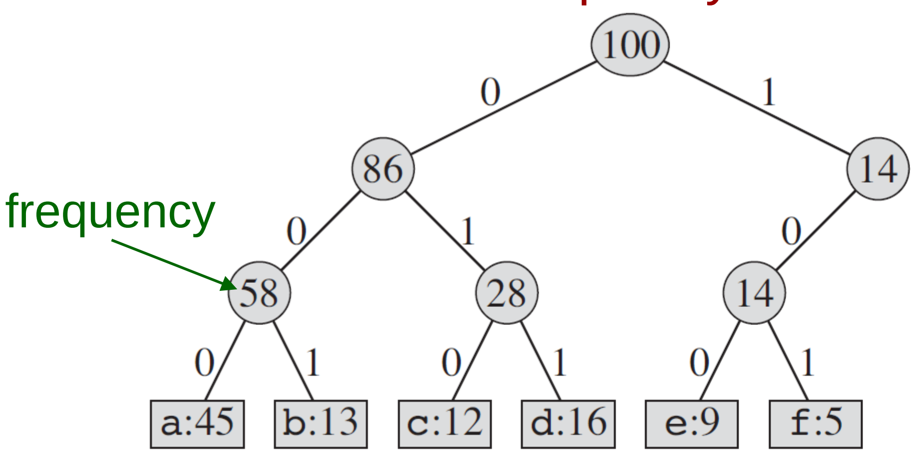


decoding: 01 10 000 000 001 11 11

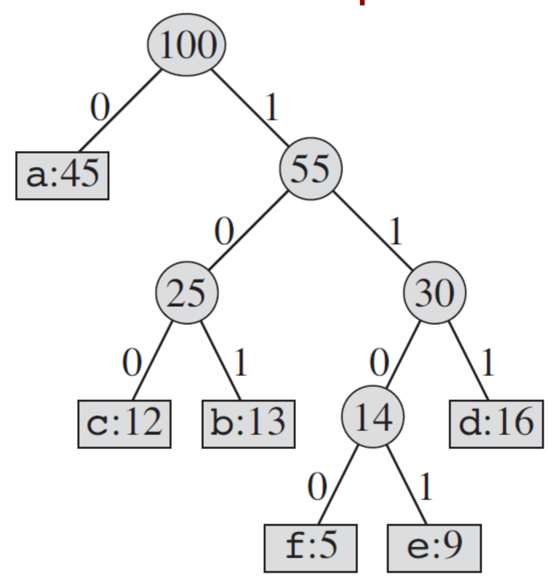
arrive at a leaf, start at root

Optimal Prefix Code Design

- **Coding Cost** of T: $B(T)=\sum_{c\in C} c.freq \cdot d_T(c)$
  - c: character in the alphabet C
  - $c.freq$: frequency of c
  - $d_T(c)$: depth of c's leaf (length of the codeword of c)
- **Code design:** Given $c_1.freq$, $c_2.freq$, …, $c_n.freq$, construct a binary tree with n leaves such that $B(T)$ is minimized
  - Idea: more frequently used characters use shorter depth



Fixed-length cost = 3 × 100 = 300

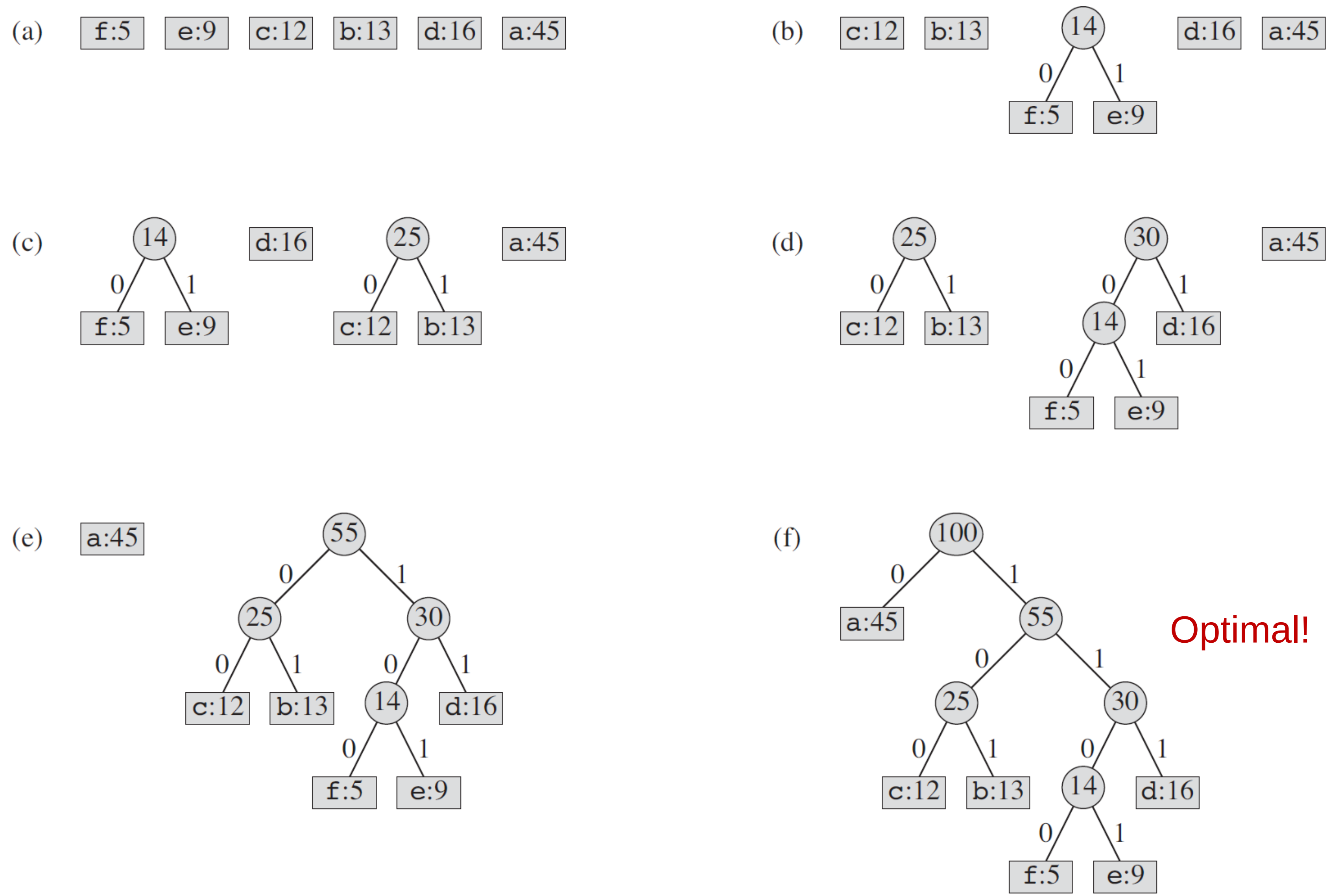


Variable-length cost = 224

Optimal code ⇒ *full* binary tree, where every nonleaf node has two children

Huffman's Procedure

- **Pair** two nodes with the least costs (lowest frequencies) at each step



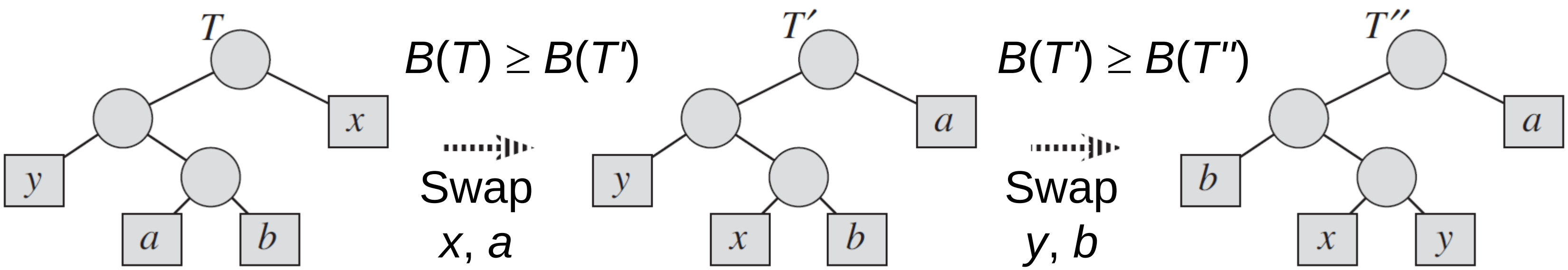
Huffman's Algorithm

```
Huffman(C)
// C: input characters; Q: min-priority queue
1. n = |C|
2. Q = C
3. for i = 1 to n -1
4.       Allocate a new node z
5.       z.left = x = Extract-Min(Q)
6.       z.right = y = Extract-Min(Q)
7.       z.freq = x.freq + y.freq
8.       Insert(Q, z)
9. return Extract-Min(Q)  //return the root of the tree
```

- Time complexity: $O(n \lg n)$
  - Build-Min-Heap needs $O(n)$ for binary min heap
  - Extract-Min(Q) needs $O(\lg n)$ by a **heap** operation
  - Requires initially $O(n \lg n)$ time to build a binary heap

Huffman's Algorithm: Greedy Choice

- **Greedy choice:** Let *x* and *y* be two characters with the lowest frequencies. ∃ an optimal prefix code for *C* where *x* and *y* have *the same length* and differ only in the last bit
  - *x* and *y* appear as sibling leaves of maximum depth in the new tree

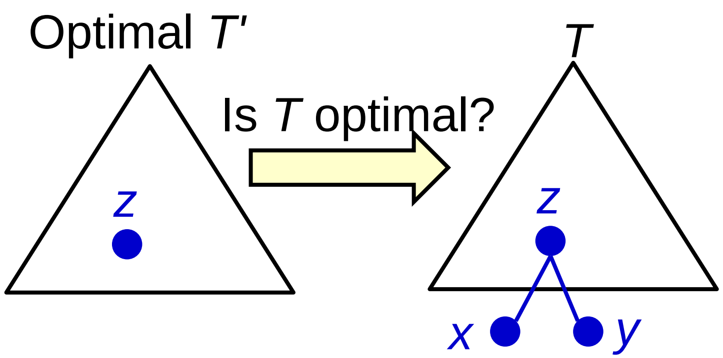


T is optimal   *T'* is optimal   *T''* is optimal

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

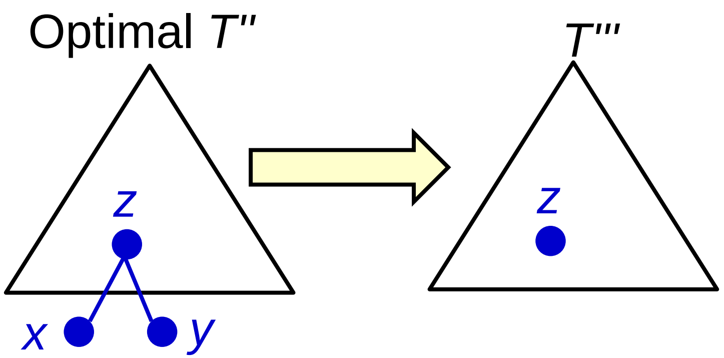
Huffman's Algorithm: Optimal Substructure

- **Optimal substructure:** $C' = C - \{x, y\} \cup \{z\}$. $z.freq = x.freq + y.freq$ of min frequency. Let T' represent an optimal prefix code over C'. Tree T obtained from T' represents an optimal prefix code for C



$z.freq = x.freq + y.freq$

$B(T) = B(T') + x.freq + y.freq$
$(d_T(x) = d_T(y) = d_{T'}(z) + 1)$

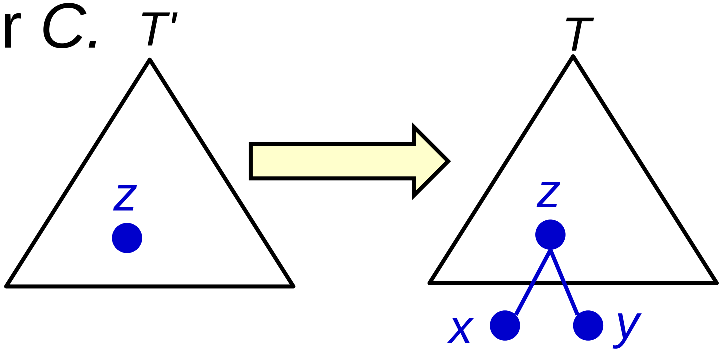


Proof by contradiction!!

Suppose T is not optimal,
$\exists$ optimal T'' such that
$B(T'') < B(T)$
$B(T''') = B(T'') - x.freq - y.freq$
$< B(T) - x.freq - y.freq$
$= B(T') \rightarrow\leftarrow$

Huffman's Algorithm: Optimality

- Lemma (optimal substructure): If T' is optimal for $C' = C - \{x, y\} \cup \{z\}$, then T is optimal for C.



- Huffman's algorithm gives an optimal cost prefix tree
- Proof by induction
  - Base case: $|C| = 2$, trivial.
  - Inductive hypothesis: Huffman's algorithm is optimal for any $|C| < n$
  - Inductive step:
    - Consider the case of $|C| = n$. We construct T' on C' where $|C'| = n$-1.
    - By inductive hypothesis, T' is optimal for C'. By the lemma above, T is then optimal for C.

Task Scheduling

Tasks with deadlines and penalties

Task Scheduling

- **The task scheduling problem:** Schedule unit-time tasks with deadlines and penalties s.t. the total penalty for missed deadlines is minimized
  - $S = \{1, 2, \ldots, n\}$ of n unit-time tasks
  - **Deadlines** $d_1, d_2, \ldots, d_n$ for tasks, $1 \le d_i \le n$
  - **Penalties** $w_1, w_2, \ldots, w_n$: w_i is incurred if task *i* misses deadline
- **Canonical form** of schedule: Early tasks precede the late tasks; early tasks are in order of monotonically increasing deadlines
- Set *A* of tasks is **independent** if ∃ a schedule with no late tasks
- $N_t(A)$: number of tasks in *A* with deadlines *t* or earlier, $t = 1, \ldots, n$

- Three equivalent statements for any set of tasks *A*
  1. *A* is independent
  2. $N_t(A) \le t$, $t = 1, 2, \ldots, n$
  3. If the tasks in *A* are scheduled in order of nondecreasing deadlines, then no task is late

Greedy Algorithm: Task Scheduling

- **The optimal greedy scheduling algorithm:**
  1. Sort penalties in nonincreasing order
  2. Find tasks of independent sets: no late task in the sets.
  3. Schedule tasks in a maximum independent set in order of nondecreasing deadlines
  4. Schedule other tasks (missing deadlines) at the end arbitrarily

Task	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

optimal scheduling: (2, 4, 1, 3, 7, 5, 6)
penalty: 30+20 = 50

$N_1(A) = 0 <= 1$
$N_2(A) = 1 <= 2$
$N_3(A) = 2 <= 3$
$N_4(A) = 4 <= 4$
$N_5(A) = 4 <= 5$
$N_6(A) = 5 <= 6$

$N_t(A) <= t$

Summary: Greedy Analysis Strategies

- An algorithm is greedy if it builds up a solution in small steps, making the choice that looks best at each step to optimize some underlying criterion
- It's challenging to prove greedy algorithms succeed in solving a nontrivial problem optimally

- **Greedy choice property**: Prove by an exchange argument: Gradually transform an optimal solution to the one found by the greedy algorithm without hurting its quality
- **Optimal substructure**: Prove by contradiction: An optimal solution to the problem contains within its optimal solution to the subproblem

Summary: Algorithmic Paradigms

- **Brute-force (Exhaustive search)**: Examine the entire set of possible solutions explicitly
  - A victim to show the efficiencies of the following methods
- **Greedy**: Build up a solution incrementally, myopically optimizing some local criterion
  - Always maintain **one** subproblem
- **Divide-and-conquer**: Break up a problem into subproblems, solve each subproblem independently, and combine solution to sub-problems to form solution to original problem
  - Subproblems have equal size
- **Dynamic programming**: Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems
  - The first step of DP: define the subproblem!