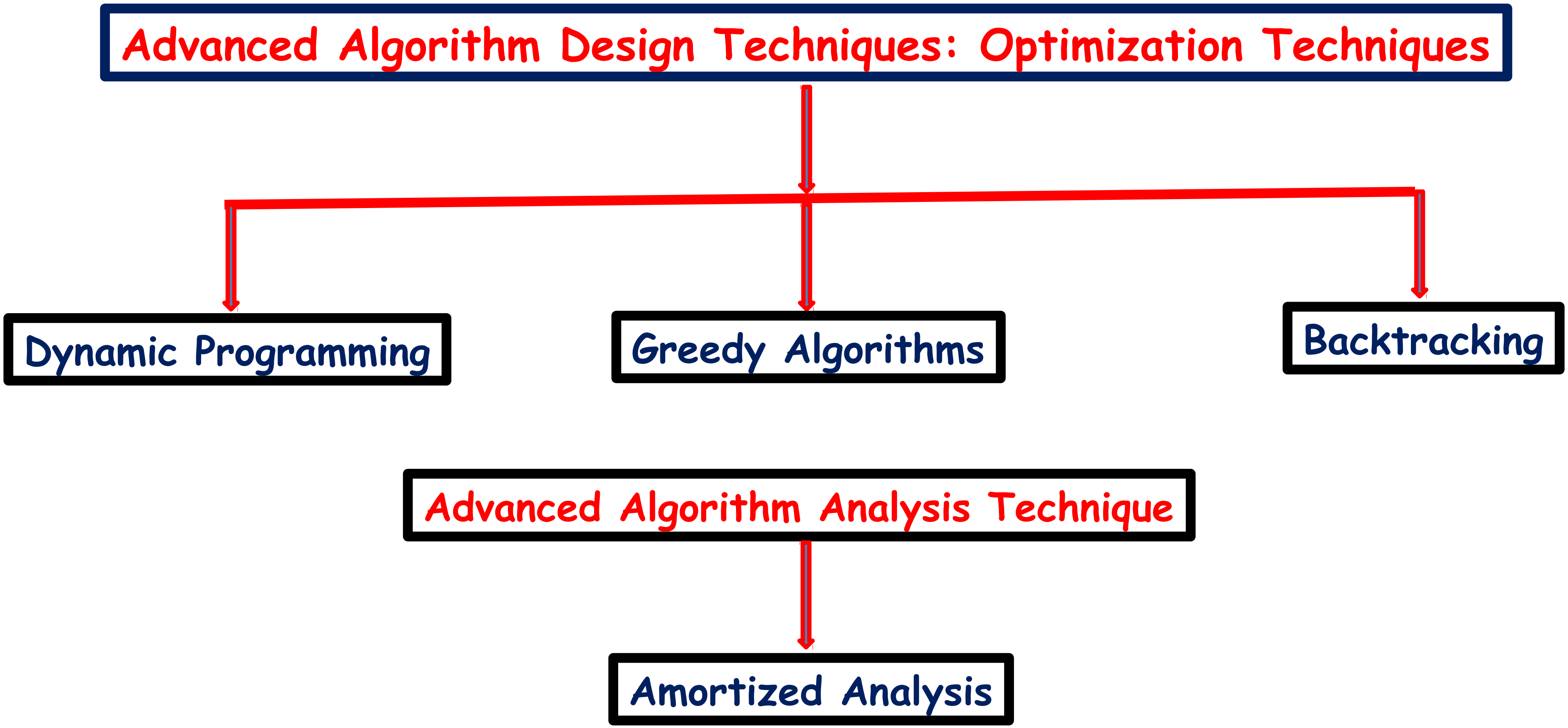
Learn DAA : From B K Sharma

TCS-503: Design and Analysis of Algorithms

Advanced Design and Analysis Techniques: Greedy Algorithms

Learn DAA: From B K Sharma



Advanced Algorithm Design Techniques: Optimization Techniques

Dynamic Programming

Optimal Substructure Property + Overlapping Substructure Property

Programming means "Tabular Method", not computer programming.

Works for more problems

In between, not as fast as greedy

Greedy Algorithms

Greedy Choice Property + Optimal Substructure Property

Can be applied to few problems

but gives fast algorithms

Backtracking

Can be applied to almost all problems

but gives very slow algorithms

Why Greedy Algorithm?

No direct solution available

Easy-to-implement solutions to complex, multi-step problems.

When Greedy Algorithm?

When the problem has:

Greedy Choice Property + Optimal Substructure Property → a.k.a. Elements of GA

↓

A good clue that a greedy strategy will solve the problem.

Greedy Choice Property

When we have a choice to make, make the one that looks best right now.

A locally greedy choice will lead to a globally optimal solution.

Make a locally optimal choice in hope of getting a globally optimal solution.

A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

How Greedy Algorithm? What are the Steps of Greedy Algorithm?

1. Formulate the optimization problem in the form:

   we make a choice and we are left with one sub-problem to solve.

2. Show that the greedy choice can lead to an optimal solution:

   so that the greedy choice is always safe.

3. Demonstrate that an optimal solution to original problem =

   greedy choice + an optimal solution to the sub-problem

4. Make the greedy choice and solve top-down.

5. May have to preprocess input to put it into greedy order : e.g. Sorting activities by finish time.

Problems to be solved using Techniques of Greedy Algorithm.

- Fractional Knapsack Problem
- Activity Selection Problem
- Huffman Code

Widely used For compressing data (assume data to be a sequence of characters)

Very efficient (saving 20-90%)

Use a table to keep frequencies of occurrence of characters.

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

Learn DAA: From B K Sharma

Problems to be solved using Techniques of Greedy Algorithm.

Fractional Knapsack Problem

Activity Selection Problem

Huffman Code

Output binary string.

"Today's weather is nice"

"001 0110 0 0 100 1000 1110"

Huffman Code

Example:

A file of 100,000 characters.
Containing only 'a' to 'f'

	Frequency	Fixed-length codeword	Variable-length codeword
'a'	45000	000	0
'b'	13000	001	101
'c'	12000	010	100
'd'	16000	011	111
'e'	9000	100	1101
'f'	5000	101	1100

Assign short codewords to frequent characters and long codewords to infrequent characters

eg. "abc" = "000001010"

300,000 bits

eg. "abc" = "0101100"

1*45000 + 3*13000 + 3*12000 + 3*16000 + 4*9000 + 4*5000
= 224,000 bits

Huffman Code

Binary tree whose leaves are the given characters.

The path from the root to the character, where 0 means "go to the left child" and 1 means "go to the right child"

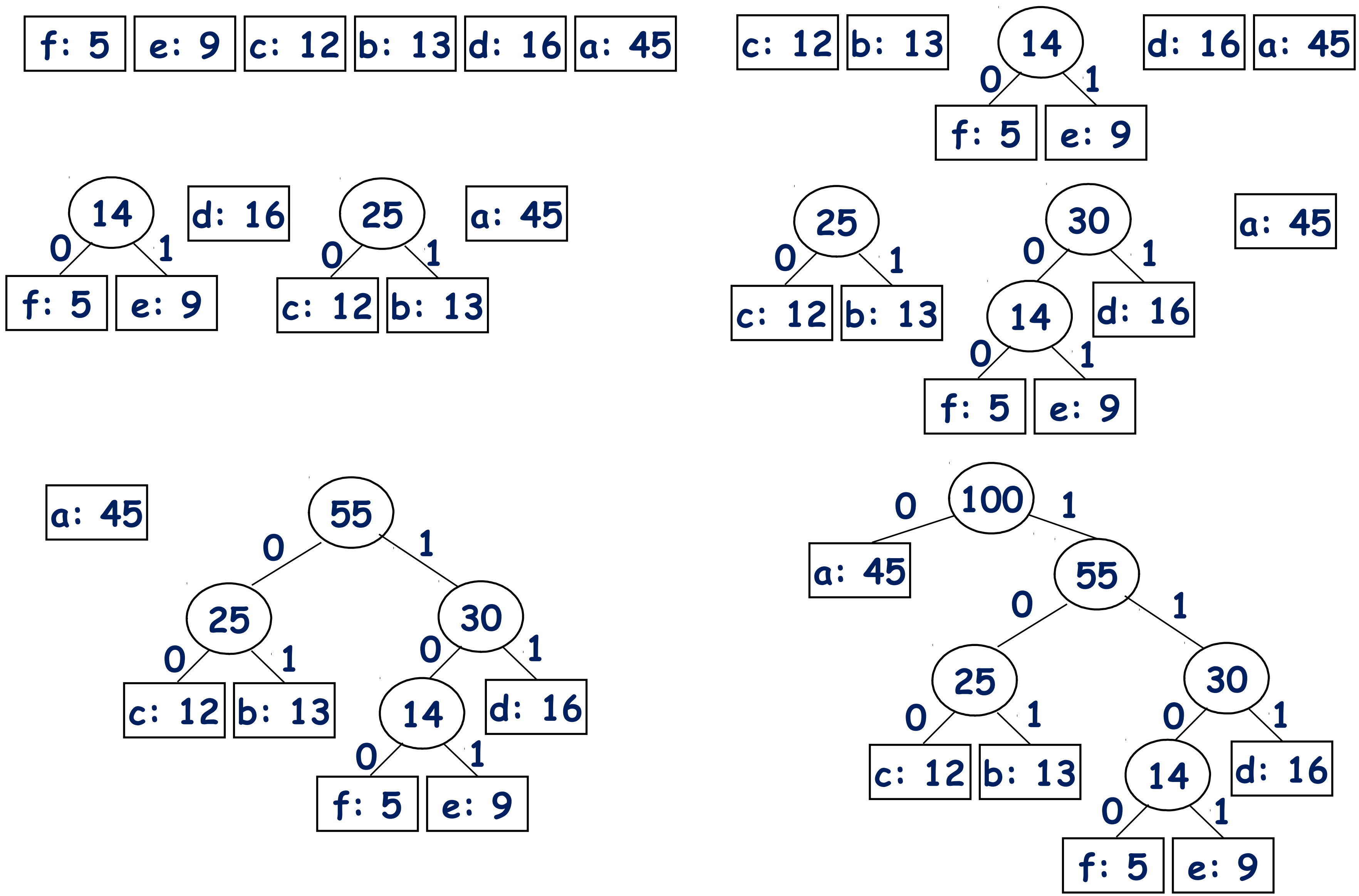
Q: A min-priority queue f:5 → e:9 → c:12 → b:13 → d:16 → a:45

Building Huffman Code

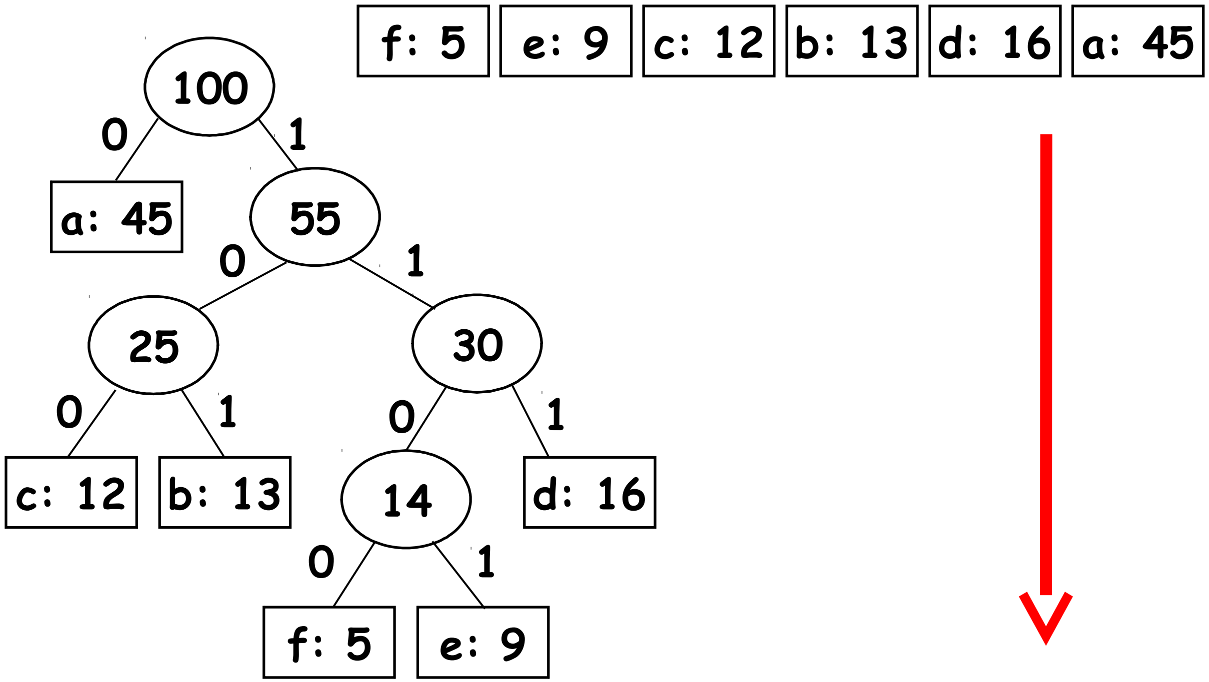
Alg.: HUFFMAN(C)

```
n ← | C |
Q ← C
for i ← 1 to n - 1
        do allocate a new node z
                left[z] ← x ← EXTRACT-MIN(Q)
                right[z] ← y ← EXTRACT-MIN(Q)
                f[z] ← f[x] + f[y]
                INSERT (Q, z)
return EXTRACT-MIN(Q)
```

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



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



a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

abc = 0 · 101 · 100 = 0101100

a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

0 0 101 11011100 = 0 · 0 · 101 · 1101 · 1100 = aabef

Variable-Length Codes

E.g.: Data file containing 100,000 characters

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

- **Assign short codewords to frequent characters and long codewords to infrequent characters**
- **a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100**

Prefix Codes

- **Prefix codes:**
  - **Codes for which no codeword is also a prefix of some other codeword**
  - **Better name would be "prefix-free codes"**
- **We can achieve optimal data compression using prefix codes**

Encoding with Binary Character Codes

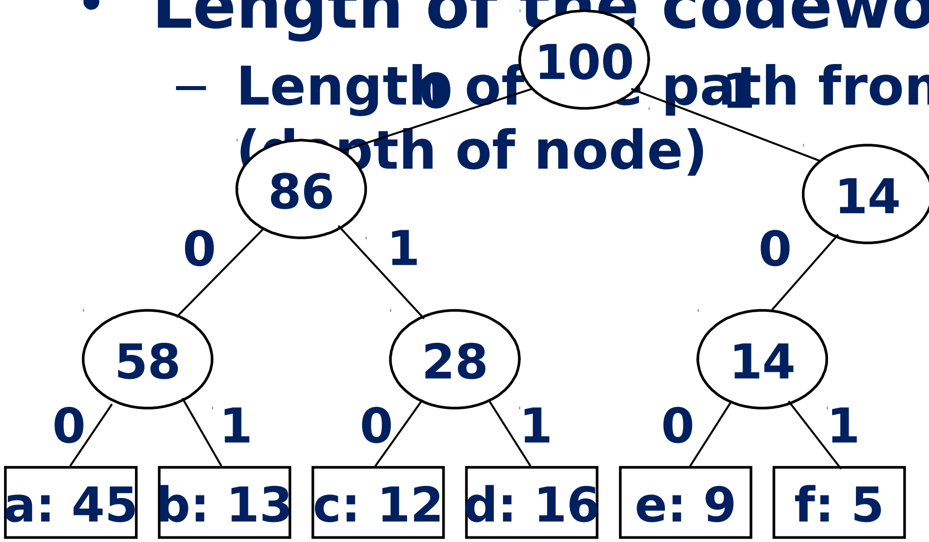
- Encoding
  - Concatenate the codewords representing each character in the file
- E.g.:
  - a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100
  - abc = 0 · 101 · 100 = 0101100

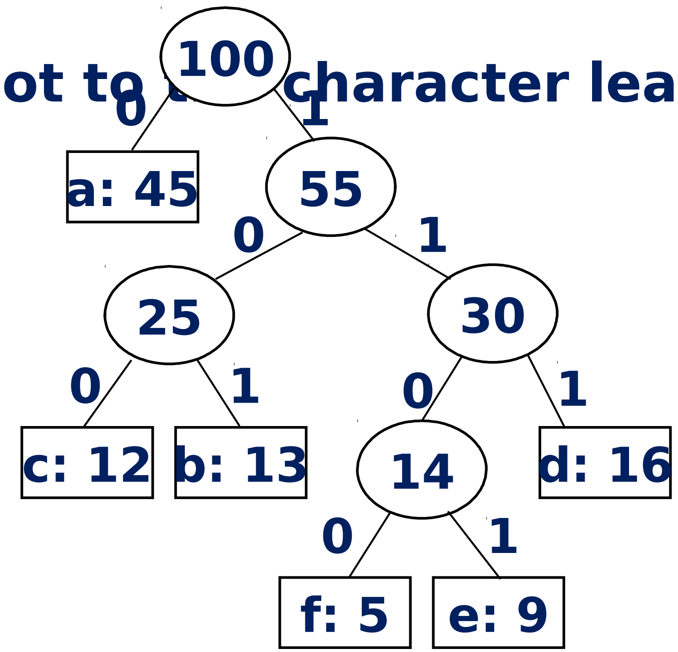
Decoding with Binary Character Codes

- **Prefix codes simplify decoding**
  - No codeword is a prefix of another ⇒ the codeword that begins an encoded file is unambiguous
- **Approach**
  - Identify the initial codeword
  - Translate it back to the original character
  - Repeat the process on the remainder of the file
- **E.g.:**
  - a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100
  - 001011101 = $0 \cdot 0 \cdot 101\,1101$ = aabe

Prefix Code Representation

- Binary tree whose leaves are the given characters
- Binary codeword
  - the path from the root to the character, where 0 means “go to the left child” and 1 means “go to the right child”
- Length of the codeword
  - Length of the path from root to the character leaf (depth of node)





Problems Definitions

- **Fractional Knapsack Problem**
  - **A thief rubbing a store finds n items: the i-th item is worth v_i dollars and weights w_i pounds (v_i, w_i integers)**
  - **The thief can only carry W pounds in his knapsack**
  - **The thief can take fractions of items**
  - **Which items and how much should the thief take to maximize the value of his load?**
- **Activity Selection Problem:**
  - **n activities $a_1, a_2, \ldots\ldots\ldots a_n$ with start time s_i and finish time f_i of each activity a_i are given, $S = \{a_1, \ldots, a_n\}$ set of activities.**
  - **Select the largest possible set of non-overlapping (*mutually compatible*) activities.**
- **Huffman Code:**
  - **Data file containing n characters (e.g. 100,000 characters)**
  - **Use the frequencies of occurrence of characters to build an optimal way of representing each character**

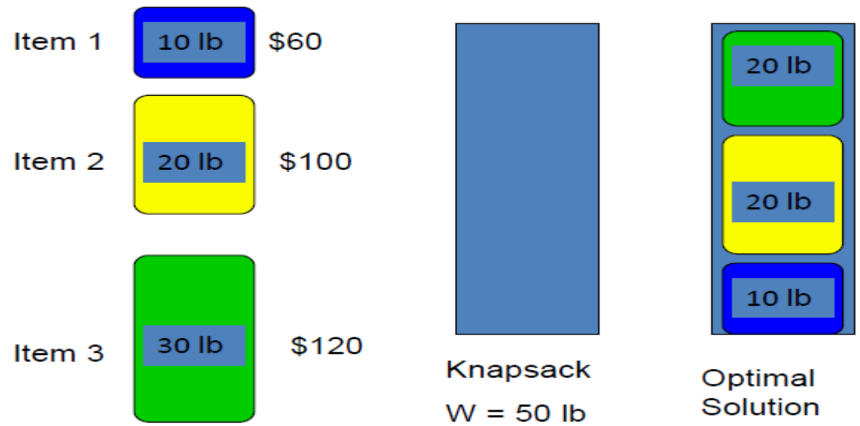
Characters	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

Compatible Activities(Non-overlapping Activities)

- **Activities a_i and a_j are compatible if the intervals (s_i, f_i) and (s_j, f_j) do not overlap**

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$$



4 *20= $80

+

5 *20 $100

+

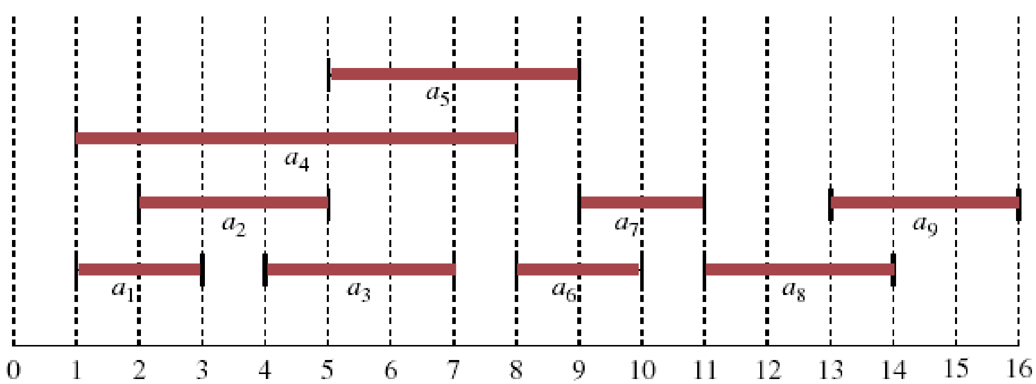
6*10 $60

$240

$6/pound$5/pound$4/pound

Example: *S* sorted by finish time:

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



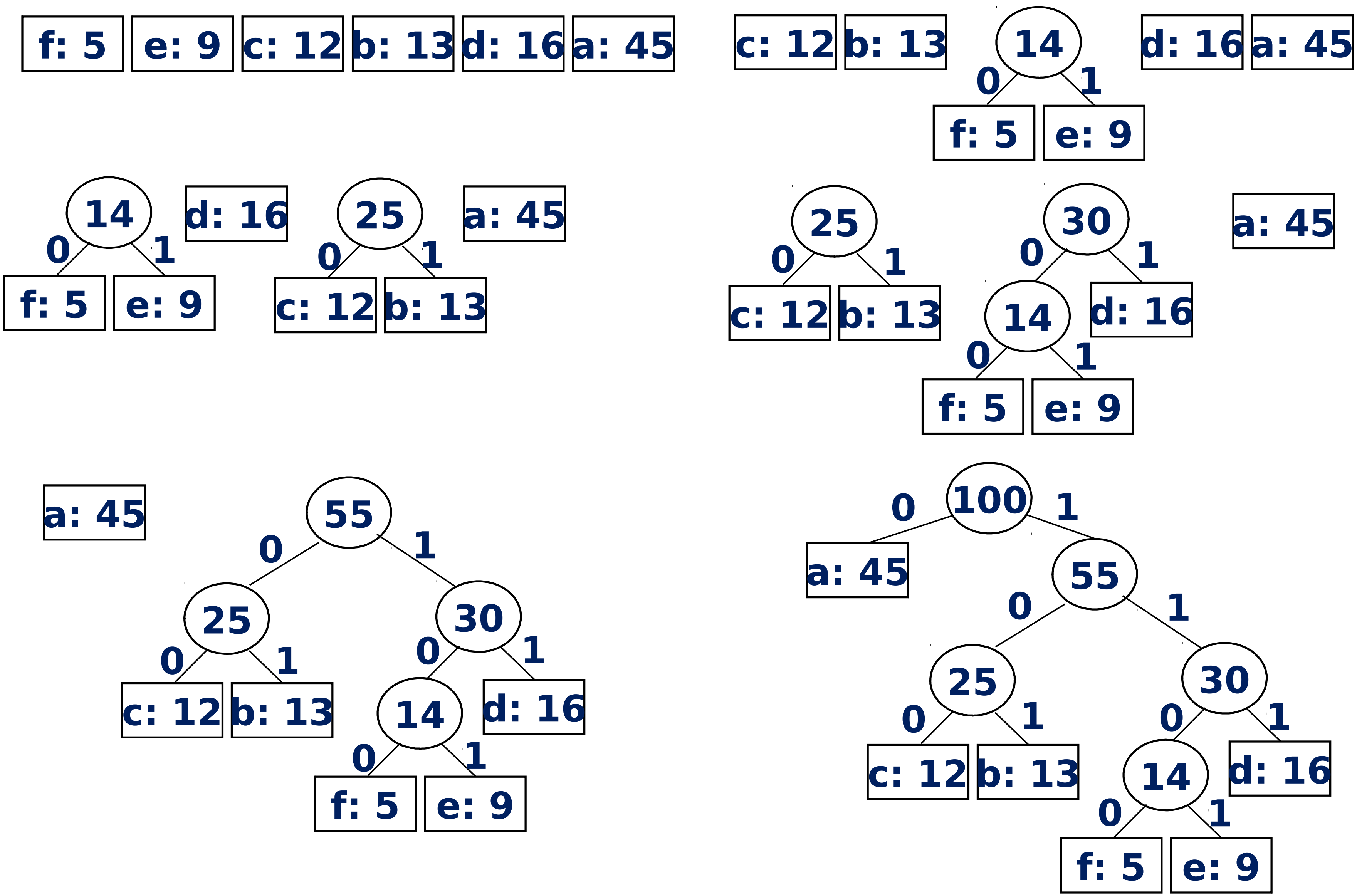
- **Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$.**
- **Not unique: also $\{a_2, a_5, a_7, a_9\}$.**

Huffman Coding

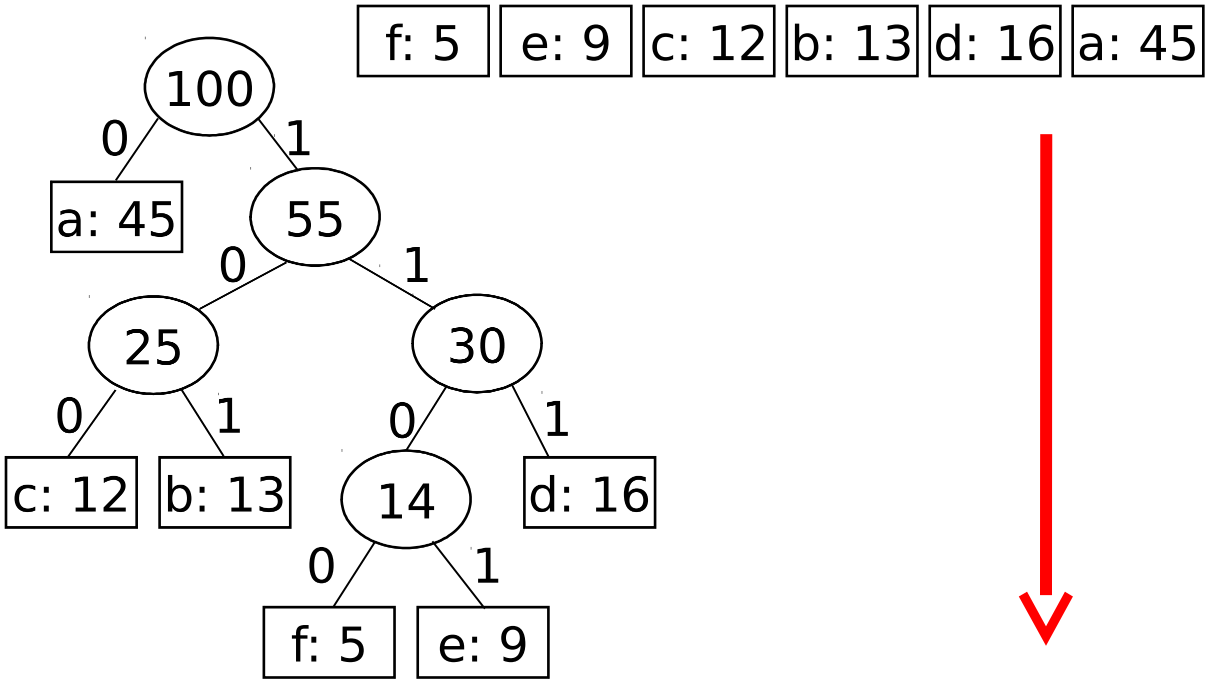
- Binary Search tree whose leaves are the given characters
- Binary codeword
  - the path from the root to the character, where 0 means “go to the left child” and 1 means “go to the right child”
- Length of the codeword
  - Length of the path from root to the character leaf (depth of node)
- E.g.:
  - a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

    0 · 0 · 101 · 1101 = aabe
  - 001011101 =

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



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



a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

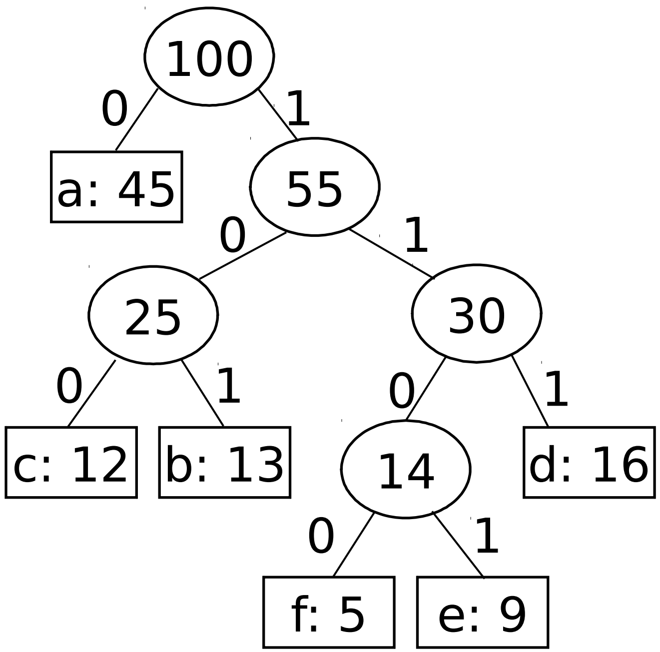
a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

abc = 0 · 101 · 100 = 0101100

0 0 101 11011100 =

00 · 101 · 1101 = aabef

Encoding: Example



Greedy Choice Property of Problems

- **Fractional Knapsack Problem:** Always Pick item with maximum v_i/w_i.
- **Activity Selection Problem :** Always choose an activity with the earliest finish time.
  - Optimal solution=first activity + optimal solution for sub-problem
    =greedy choice +optimal selection from activities
    that do not overlap with greedy choice
    =activity that ends first $\Rightarrow$ greedy choice.
- **Huffman Code:** Always merge the two characters with the lowest frequencies

Optimal Choice Property of Problems

- **Fractional Knapsack Problem :** Consider the most valuable load that weights at most W pounds
  - If we remove a weight w of item i from the optimal load

    $\Rightarrow$ The remaining load must be the most valuable load weighing at most W – w that can be taken from the remaining n – 1 items plus w_i – w pounds of item i

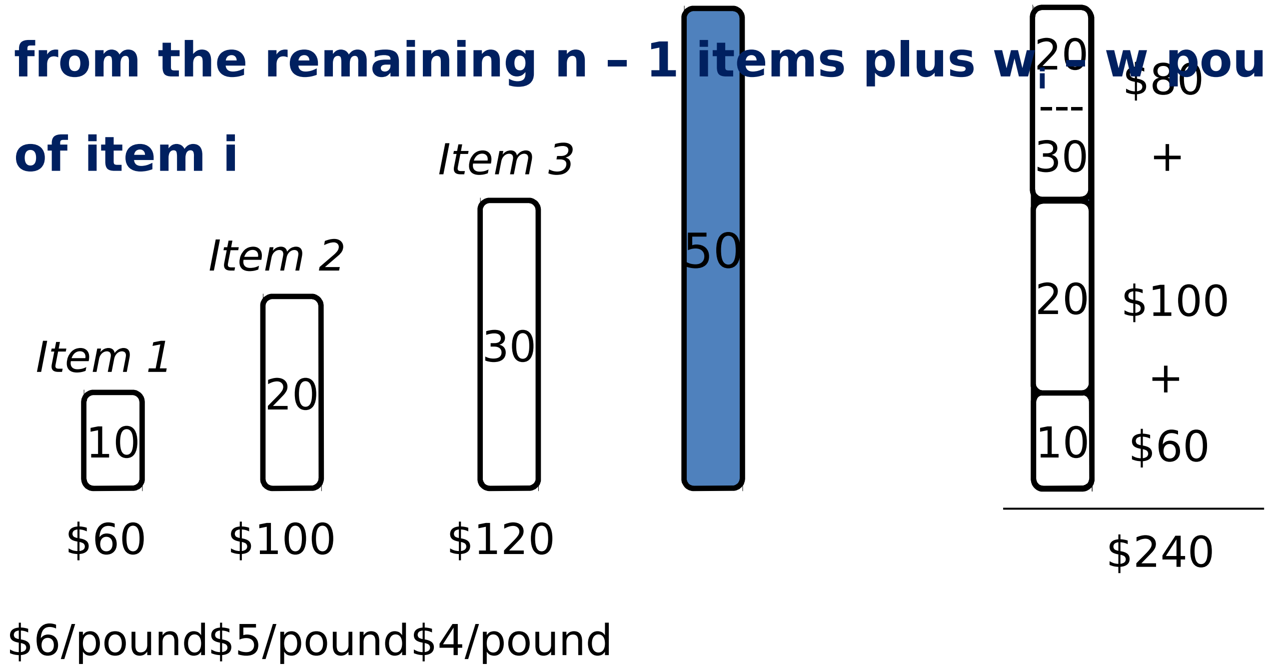
- **Activity Selection :** If an optimal solution to sub-problem S_{ij} includes activity $a_k \Rightarrow$ it must contain optimal solutions to S_{ik} and S_{kj}

  Similarly, a_m + optimal solution to $S_{mj} \Rightarrow$ optimal sol.

- **Huffman Code:** If T is a Binary Tree representing optimal code and T' is a sub-tree of T then T' must be optimal.

If we remove a weight w of item i from the optimal load

$\Rightarrow$ **The remaining load must be the most valuable load weighing at most W – w that can be taken from the remaining n – 1 items plus w_i – w pounds of item i**



- **Choose a weight *w* from item i (part of i, say part of item 1, 5lbs), then remove that part, the remaining load is the most valuable load weighing at most *W*-*w(50-5=45 lbs)* that the thief can take from the *n*-1 original items plus w_i-*w(10-5)* pounds from item i.**

Fractional Knapsack Problem:

1. Sort items by v_i/w_i, renumber.
2. For i = 1 to n

   Add as much of item i as possible

Fractional Knapsack Problem

Alg.: **Fractional-Knapsack (W, v[n], w[n])**

1. While w > 0 and as long as there are items remaining
2. pick item with maximum v_i/w_i
3. $x_i \leftarrow \min(1, w/w_i)$
4. remove item i from list
5. $w \leftarrow w - x_i w_i$

- w – the amount of space remaining in the knapsack (w = W)

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

Pick first item because $v_i/w_i=6$ which is maximum.

$x_i=\min(1, 50/10)=\min(1,5)=1$

$w=w-x_iw_i=50- 1*10=50-10=40$

w>0 and item remains

Pick second item since $v_i/w_i=5$

$x_i=\min(1,40/20)=\min(1,2)=1$

$w=w-x_iw_i=40-1*20=40-20=20$

w>0 and item remains

Pick the third item

$x_i=\min(1,20/30)=\min(1, 0.66)=0.66$

$w=w-x_iw_i=20-0.66*30=20-19.8=0.2$

Remaining weight of third item=30-19.8=10.2

w>0 and item remains

$x_i=\min(1,0.2/10.2)=\min(1,0.019)=0.019$

$w=w-x_iw_i=0.2-0.019*10.2=0.2-0.2=0$

Alg.: Fractional-Knapsack (W, v[n], w[n])

1. **While w > 0 and as long as there are items remaining**
2. **pick item with maximum v_i/w_i**
3. **$x_i \leftarrow \min(1, w/w_i)$**
4. **remove item i from list**
5. **$w \leftarrow w - x_iw_i$**

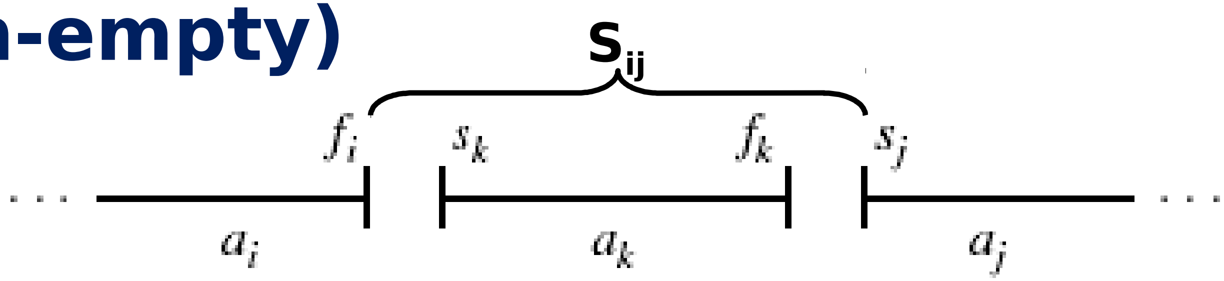
w the amount of space re... = W)

Optimal Load=10+20+(19.8 =49.8 lbs

Optimal cost=10* 6 +20 * 5 +19.8 =$239.2

Optimal Substructure Property :ASP

- **Subproblem:**
  - **Select a maximum size subset of mutually compatible activities from set S_{ij}**
- **Assume that a solution to the above subproblem includes activity a_k (S_{ij} is non-empty)**



$$S_{ij}=\{a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8,a_9\}$$

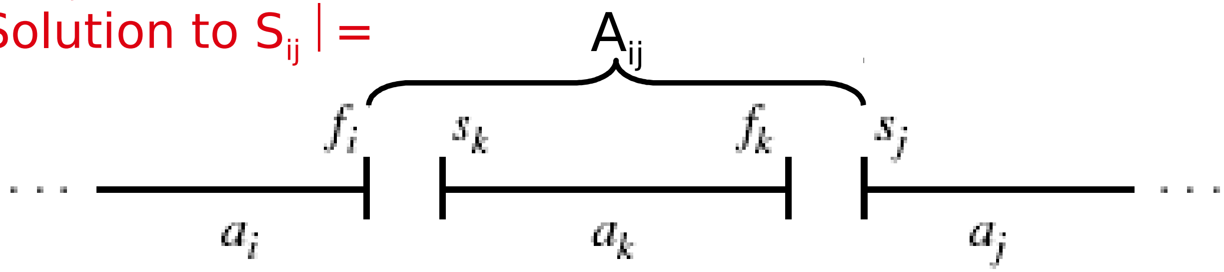
$$\{a_1, a_3, a_6, a_8\}.$$

Optimal Substructure Property :ASP

Solution to S_{ij} = (Solution to S_{ik}) ∪ {a_k} ∪ (Solution to S_{kj})

$|$Solution to $S_{ik}| + 1 + |$Solution to $S_{kj}|$

$|$Solution to $S_{ij}| =$



A_{ij} = Optimal solution to S_{ij}

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} \text{ ,}$$

Optimal Substructure Property :ASP

- **Let a_m be the activity in S_{ij} with the earliest finish time:**

$$f_m = \min \{ f_k : a_k \in S_{ij} \}$$

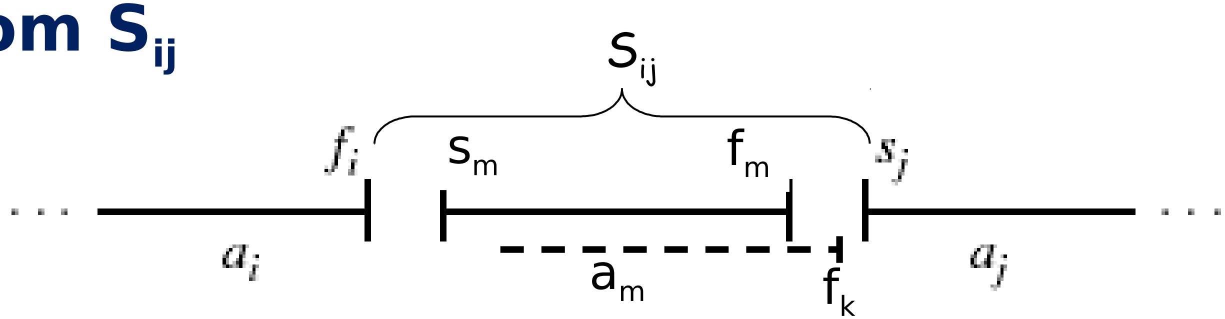
Then

a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}

- **There exists some optimal solution that contains a_m**

Proof: Greedy Choice Property : ASP

- a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}
- A_{ij} = optimal solution for activity selection from S_{ij}
  - finish
  -



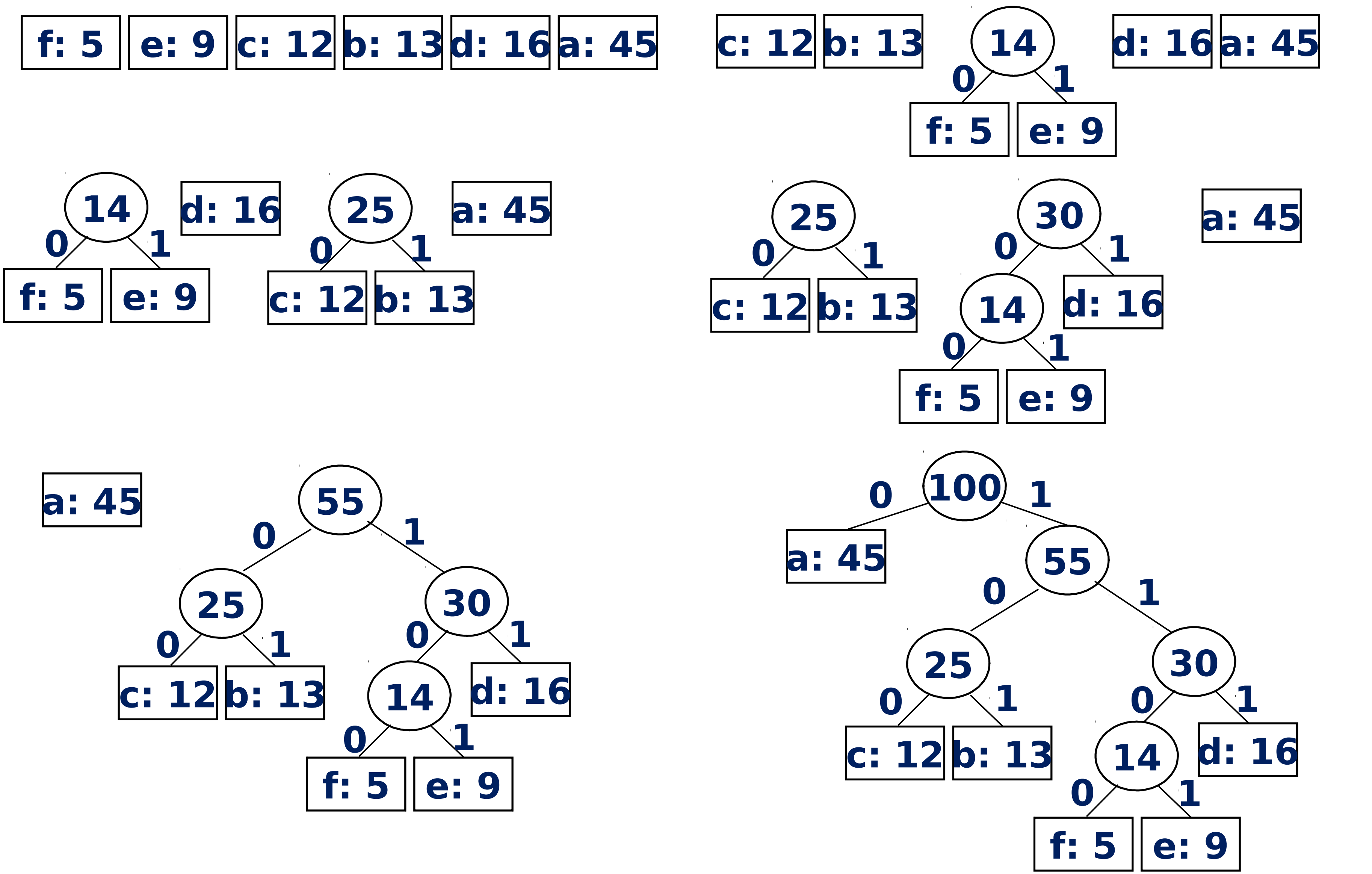
Proof: Greedy Choice Property : ASP

- **If $a_k = a_m$ Done!**
- **Otherwise, replace a_k with a_m (resulting in a set A_{ij}')**
  - **since $f_m \leq f_k$ the activities in A_{ij}' will continue to be compatible**
  - **A_{ij}' will have the same size with $A_{ij} \Rightarrow a_m$ is used in some maximum-size subset**

Greedy Approach : ASP

- **To select a maximum size subset of mutually compatible activities from set S_{ij}:**
  - **Choose $a_m \in S_{ij}$ with earliest finish time (greedy choice)**
  - **Add a_m to the set of activities used in the optimal solution**
  - **Solve the same problem for the set S_{mj}**
- **By choosing a_m we are guaranteed to have used an activity included in an optimal solution**
  - **$\Rightarrow$ We do not need to solve the sub-problem S_{mj} before making the choice!**

Example: Huffman Code



Constructing a Huffman Code

- A greedy algorithm that constructs an optimal prefix code called a Huffman code
- Assume that:
  - C is a set of n characters
  - Each character has a frequency f(c)
  - The tree T is built in a bottom up manner
- Idea:

f: 5	e: 9	c: 12	b: 13	d: 16	a: 45

  - Start with a set of |C| leaves
  - At each step, merge the two least frequent objects: the frequency of the new node = sum of two frequencies
  - Use a min-priority queue Q, keyed on f to identify the two least frequent objects

Building a Huffman Code

Alg.: HUFFMAN(C) **Running time: O(nlgn)**

```
n ← |C|
Q ← C                                   ← O(n)
for i ← 1 to n – 1
      do allocate a new node z
            left[z] ← x ← EXTRACT-MIN(Q)
            right[z] ← y ← EXTRACT-MIN(Q)            } O(nlgn)
            f[z] ← f[x] + f[y]
            INSERT (Q, z)
return EXTRACT-MIN(Q)
```

Designing Greedy Algorithms

1. **Cast the optimization problem as one for which:**
   - **we make a choice and are left with only one sub-problem to solve**
2. **Prove the GREEDY CHOICE**
   - **that there is always an optimal solution to the original problem that makes the greedy choice**
3. **Prove the OPTIMAL SUBSTRUCTURE:**
   - **the greedy choice + an optimal**

Elements of Greedy Algorithms

1. Greedy Choice Property

- A globally optimal solution can be arrived at by making a locally optimal (greedy) choice

2. Optimal Substructure Property

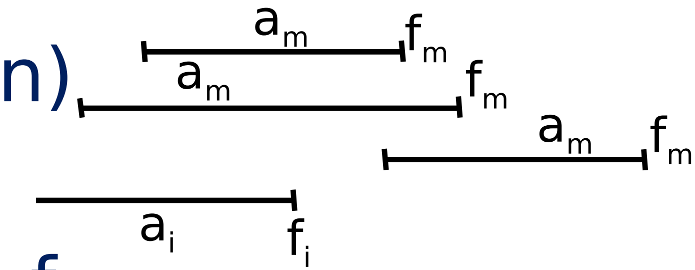
- We know that we have arrived at a subproblem by making a greedy choice
- greedy choice+ Optimal solution to subproblem $\Rightarrow$ optimal solution for the original problem

Activity Selection

- **Greedy Choice Property**
  - **There exists an optimal solution that includes the greedy choice:**
    - **The activity a_m with the earliest finish time in S_{ij}**
- **Optimal Substructure:**

If an optimal solution to sub-problem S_{ij} includes activity a_k $\Rightarrow$ it must contain optimal solutions to S_{ik} and S_{kj}

A Recursive Greedy Algorithm



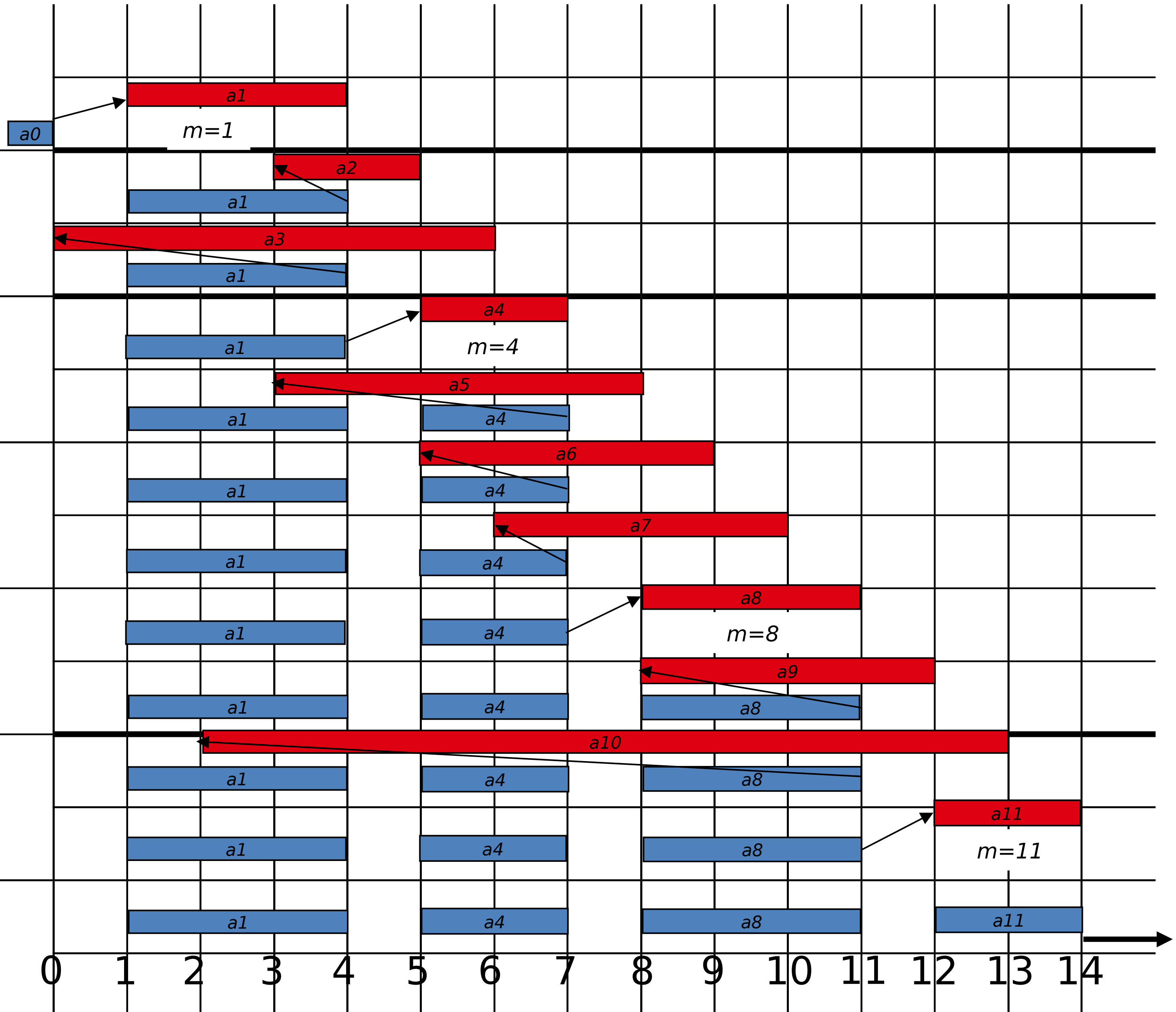
Alg.: REC-ACT-SEL (s, f, i, n)

1. $m \leftarrow i + 1$
2. **while** $m \le n$ **and** $s_m < f_i$ ►Find first activity in S_{ij}
3. **do** $m \leftarrow m + 1$
4. **if** $m \le n$ ► if first compatible activity is found
5. **then return** $\{a_m\} \cup$ REC-ACT-SEL(s, f, m, n)
6. **else return** $\varnothing$

- The while loop examines $a_{i+1}, a_{i+2}, \ldots\ldots a_j$ until it finds first activity a_m which is compatible with a_i, such

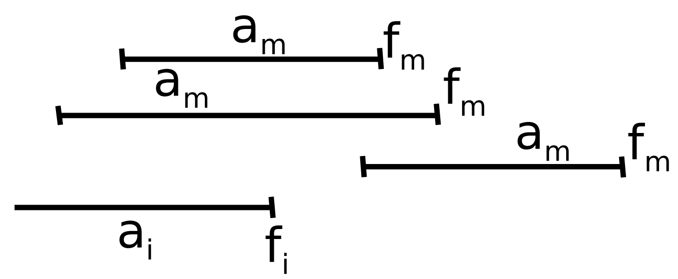
Example

i	s_i	f_i
0	-	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9



An Iterative Greedy Algorithm

Alg.: GREEDY-ACTIVITY-SELECTOR(s, f)



1. $n \leftarrow$ length[s]
2. $A \leftarrow \{a_1\}$
3. $i \leftarrow 1$
4. for $m \leftarrow 2$ to n
5. do if $s_m \geq f_i$ ► activity a_m is compatible with a_i
6. then $A \leftarrow A \cup \{a_m\}$
7. $i \leftarrow m$ ► a_i is most recent addition to A
8.

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16

$A = \{a_1, a_3, a_6, a_8\}$

Dynamic Programming vs. Greedy Algorithms

- **Dynamic programming**
  - **We make a choice at each step**
  - **The choice depends on solutions to sub-problems**
  - **Bottom up solution, from smaller to larger sub-problems**
- **Greedy algorithm**
  - **Make the greedy choice and THEN**
  - **Solve the sub-problem arising after the choice is made**
  - **The choice we make may depend on previous choices, but not on solutions to**

Steps Toward Our Greedy Solution

1. Determine the optimal substructure of the problem
2. Develop a recursive solution
3. Prove that one of the optimal choices is the greedy choice
4. Show that all but one of the subproblem resulted by making the greedy choice are empty.
   - For example if greedy choice is a_k then first schedule that activity and we skip all other activities that are not compatible with the a_k

Designing Greedy Algorithms

- More generally, we design the greedy algorithms according to the following steps:

1. Cast the optimization problem as one for which:

   we make a choice and are left with only one

   subproblem to solve.

2. Prove that there is always an optimal solution to the original problem that makes the greedy choice.
   - Making the greedy choice is always safe

3. Demonstrate that after making the greedy

General Greedy Algorithm

function greedy (C: set) :set

S ← ∅

while not solution (S) and C ≠ ∅ do

x ← an element of C maximizing

select(x)

C ← C - {x}

if feasible (S ∪ {x}) then

S ← S ∪ {x}

ιf solution (S) then return S

else return “no solutions”

The Knapsack Problem

- The 0-1 knapsack problem
  - A thief rubbing a store finds n items: the i^{th} item is worth v_i dollars and weights w_i pounds (v_i, w_i integers)
  - The thief can only carry W pounds in his knapsack
  - Items must be taken entirely or left behind
  - Which items should the thief take to maximize the value of his load?
- The fractional knapsack problem
  - Similar to above
  - The thief can take fractions of items

Questions

- **Which problem exhibits greedy choice property?**

- **Which one exhibits the optimal-substructure property?**

Comparisons

- **Which problem exhibits greedy choice property?**
  - **Think about the value per pound of items.**
    - **The fractional knapsack problem allows the thief to maximize his take, by selecting items in order of decreasing value per pound (objective function) and taking all of the most valuable item available at each time.**
    - **So, this obeys the greedy choice property.**
    - **The 0-1 knapsack problem fails with the greedy strategy, regardless of what objective function is used.**

Comparisons

- **Which problem exhibits greedy choice property?**
  - Think about the value per pound of items.
    - The fractional knapsack problem allows the thief to maximize his take, by selecting items in order of decreasing value per pound (objective function) and taking all of the most valuable item available at each time.
    - So, this obeys the greedy choice property.
    - The 0-1 knapsack problem fails with the greedy strategy, regardless of what objective function is used.
- **Which problem exhibits the optimal-substructure property?**
  - **Both of them do.**
    - For the 0-1 knapsack problem, removing the most valuable item from the knapsack means what remains is the most valuable load in the absence of what was removed.
    - The same argument holds for the fractional problem, except the most valuable fraction of an item is removed each time.

The Knapsack Problem

- ***0-1 knapsack problem***
- ***Fractional knapsack problem***
- **Both have optimal substructure property.**
  - **0-1 : choose the most valuable load j that weighs $w_j \le W$, remove j, choose the most valuable load i that weighs $w_i \le W-w_j$**
  - **fractional: choose a weight w from item j (part of j), then remove the part, the remaining load is the most valuable load weighing at most $W-w$ that the thief can take from the $n-1$ original items plus w_j-w pounds from item j.**

The Knapsack Problem

- **But the fractional problem has the greedy-choice property, and the 0-1 problem does not.**

The Knapsack Problem

- **0-1 knapsack problem has not the greedy-choice property**
- ***W* = 50.**
- **Greedy solution:**
  - **take items 1 and 2**
  - **value = 160, weight = 30**

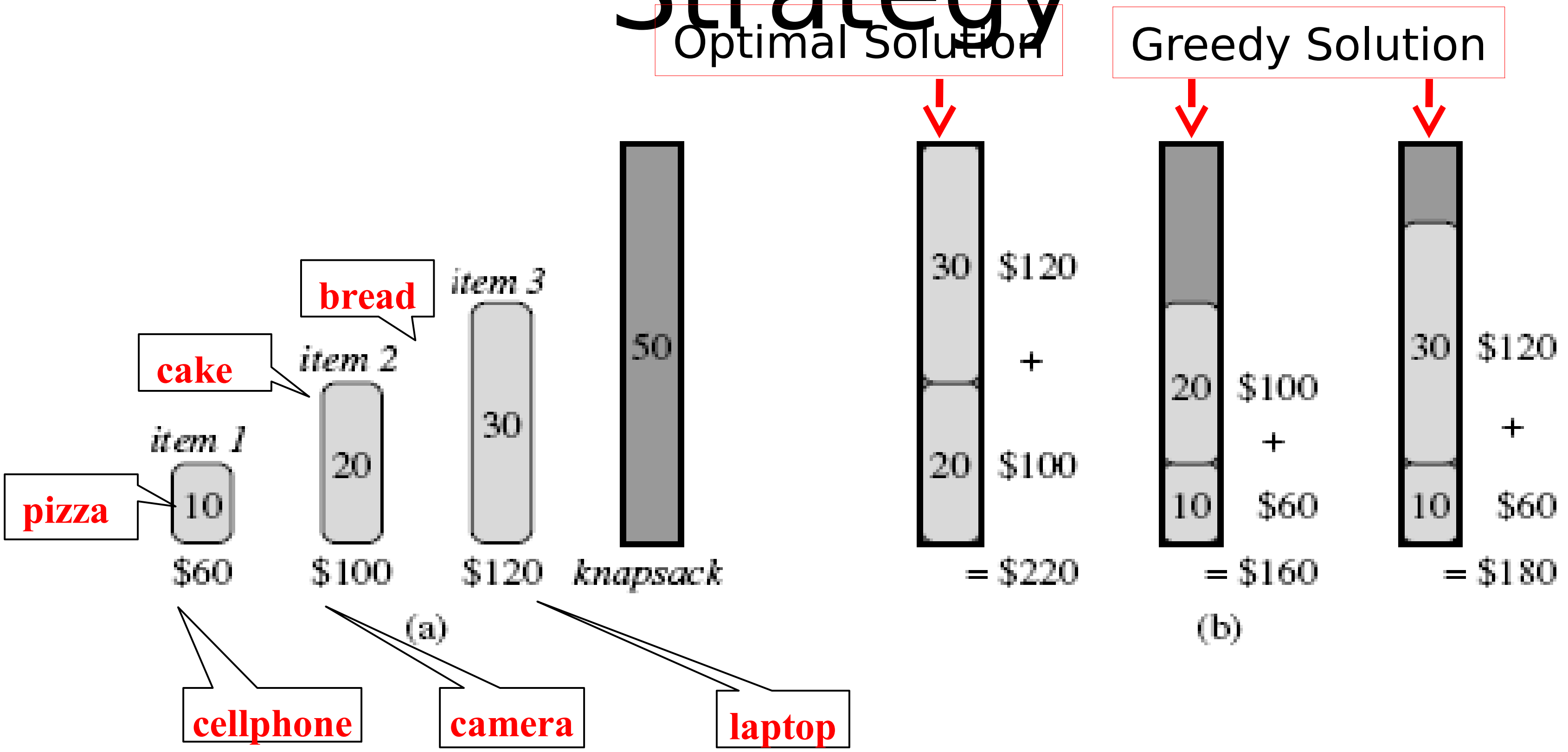
  20 pounds of capacity leftover.

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i / w_i	6	5	4

- **Optimal solution:**
  - **Take items 2 and 3**
  - **value=220, weight=50**

  No leftover capacity.

0-1 Knapsack - Greedy Strategy



$6/pound $5/pound $4/pound

- **None of the solutions involving the greedy choice (item 1) leads to an optimal solution**
  - **The greedy choice property does not hold**

0-1 Knapsack - Greedy Strategy

ow that greedy strategy does work for 0-1 Knapsack.

- The value per pound of item 1 is $6 per pound, which is greater than the value per pound of eithe item 2 or either 3.

- The greedy strategy, therefore, would take item 1 first.

- But the optimal solution takes items 2 and 3(16(b)), leaving item 1 behind.

- The two possible solutions that involve item 1 are both suboptimal.

0-1 Knapsack - Greedy Strategy(Skip)

- **In the 0-1 problem, when we consider an item for**
  inclusion in the knapsack, we must compare the solution to the subproblem in which the item is excluded before we can make the choice.

- **The problem formulated in this way gives rise to many overlapping subproblems - a hallmark of dynamic programming and hence dynamic programming can be used to solve the 0-1 problem.**

Fractional Knapsack Problem

- **Knapsack capacity:** W
- **There are n items:** the i^{th} item has value v_i and weight w_i
- **Goal:**
  - find x_i such that for all $0 \leq x_i \leq 1$, $i = 1, 2, .., n$

    $\sum w_i x_i \leq W$ and

    $\sum x_i v_i$ is maximum

Fractional Knapsack Problem

- **Greedy strategy 1:**
  - **Pick the item with the maximum value**
- **E.g.:**
  - **$W = 1$**
  - **$w_1 = 100$, $v_1 = 2$**
  - **$w_2 = 1$, $v_2 = 1$**
  - **Taking from the item with the maximum value:**

    Total value taken = $v_1/w_1 = 2/100=0.5$
  - **Smaller than what the thief can take if choosing the other item**

    Total value (choose item 2) = $v_2/w_2 = 1$

Fractional Knapsack Problem

Greedy strategy 2:

- **Pick the item with the maximum value per pound v_i/w_i**
- **If the supply of that element is exhausted and the thief can carry more: take as much as possible from the item with the next greatest value per pound**
- **It is good to order items based on their value per pound**

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$$

Fractional Knapsack Problem

Alg.: **Fractional-Knapsack (W, v[n], w[n])**

1. **While w > 0 and as long as there are items remaining**
2. **pick item with maximum v_i/w_i**
3. **$x_i \leftarrow \min(1, w/w_i)$**
4. **remove item i from list**
5. **$w \leftarrow w - x_i w_i$**

- **w – the amount of space remaining in the knapsack (w = W)**

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

Pick first item because $v_i/w_i=6$ which is maximum.

$x_i=\min(1, 50/10)=\min(1,5)=1$

$w=w-x_iw_i=50-1*10=50-10=40$

w>0 and item remains

Pick second item since $v_i/w_i=5$

$x_i=\min(1,40/20)=\min(1,2)=1$

$w=w-x_iw_i=40-1*20=40-20=20$

w>0 and item remains

Pick the third item

$x_i=\min(1,20/30)=\min(1, 0.66)=0.66$

$w=w-x_iw_i=20-0.66*30=20-19.8=0.2$

w>0 and item remains

$x_i=\min(1,0.2/0.2)=\min(1,1)=1$

$w=w-x_iw_i=0.2-1*0.2=0.2-0.2=0$

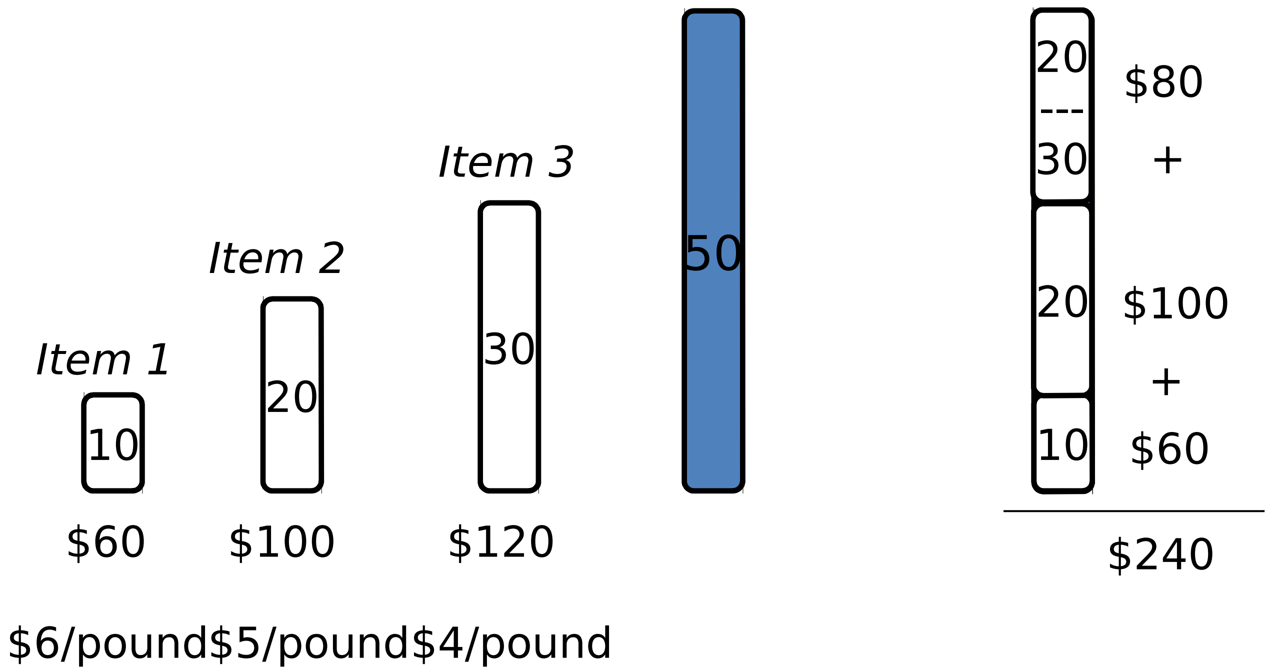
Alg.: Fractional-Knapsack (W, v[n], w[n])

1. While w > 0 and as long as there are items remaining
2. pick item with maximum v_i/w_i
3. $x_i \leftarrow \min(1, w/w_i)$
4. remove item i from list
5. $w \leftarrow w - x_iw_i$

w – the amount of space remaining in the knapsack (w = W)

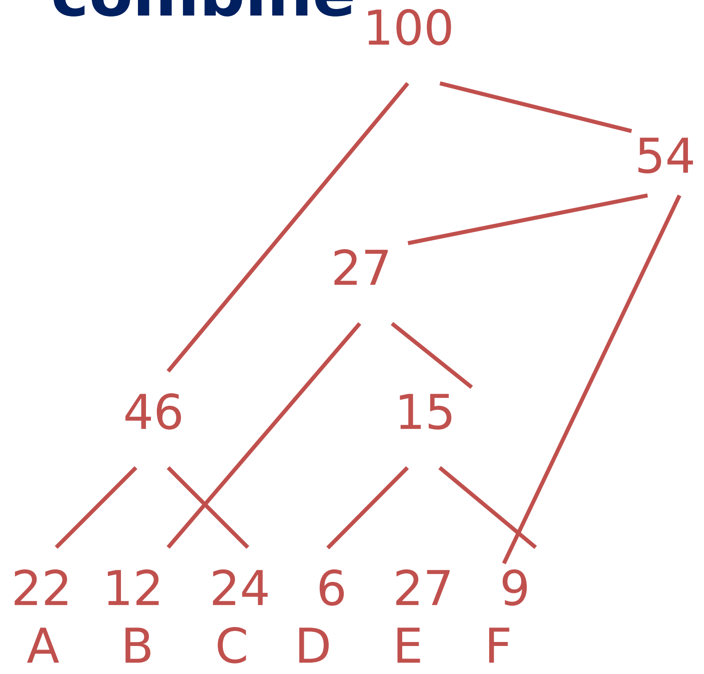
Fractional Knapsack - Example

- E.g.:



Huffman encoding

- **The Huffman encoding algorithm is a greedy algorithm**
- **You always pick the two smallest numbers to combine**

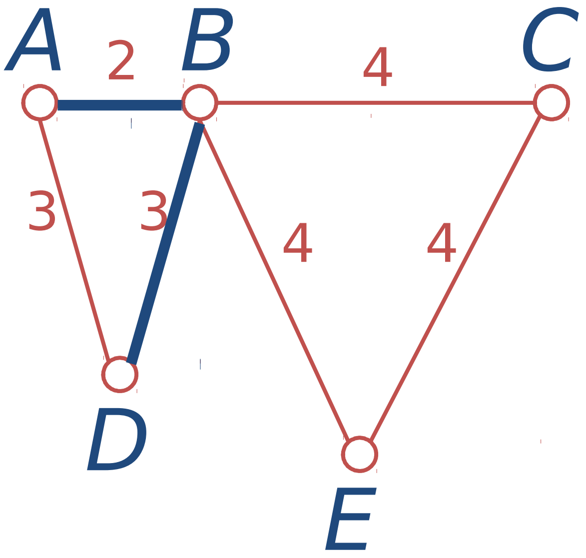


A=00
B=100
C=01
D=1010
E=11
F=1011

- **Average bits/char:**
  0.22*2 +
  0.12*3 +
  0.24*2 +
  0.06*4 +
  0.27*2 +
  0.09*4
  = 2.42
- **The Huffman algorithm finds an optimal**

Traveling salesman

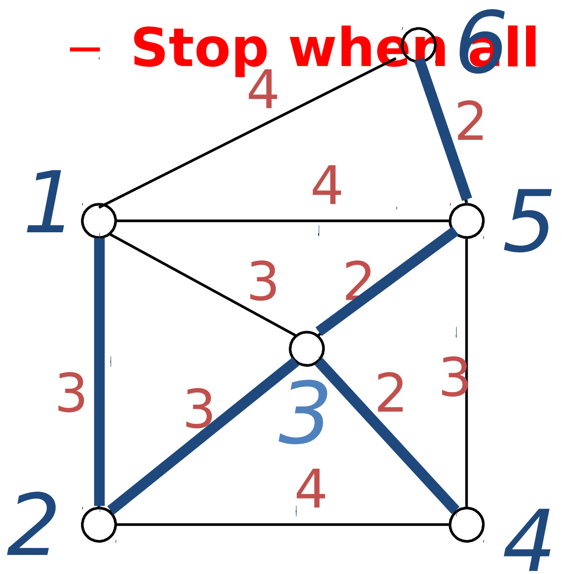
- **A salesman must visit every city (starting from city *A*), and wants to cover the least possible distance**
  - **He can revisit a city (and reuse a road) if necessary**
- **He does this by using a greedy algorithm: He goes to the next nearest city from wherever he is**



- From ***A*** he goes to ***B***
- From ***B*** he goes to ***D***
- This is *not* going to result in a shortest path!
- The best result he can get now will be ***ABDBCE***, at a cost of 16
- An actual least-cost path from ***A*** is ***ADBCE***, at a cost of 14

Minimum spanning tree

- **A minimum spanning tree is a least-cost subset of the edges of a graph that connects all the nodes**
  - **Start by picking any node and adding it to the tree**
  - **Repeatedly: Pick any *least-cost* edge from a node in the tree to a node not in the tree, and add the edge and new node to the tree**
  - **Stop when all nodes have been added to the tree**



- The result is a least-cost **(3+3+2+2+2=12)** spanning tree
- If you think some other edge should be in the spanning tree:
  - **Try adding that edge**
  - **Note that the edge is part of a cycle**
  - **To break the cycle, you must remove the edge with the greatest cost**