Chap 16 – Greedy Algorithms

16.1 An activity-selection problem

16.2 Elements of the greedy strategy

16.3 Huffman codes

16.4 Matroids and greedy methods

16.5 A task-scheduling problem as a matroid

Greedy Algorithms

- Greedy Algorithms
  - Similar to dynamic programming
  - Used for optimization problems
  - ***Idea***
    - When we have a choice to make, make the one that looks best *right now*.
    - Make a *locally optimal choice* in hope of getting a *globally optimal solution*.
  - Greedy algorithms don't always yield an optimal solution. But sometimes they do.
  - Even so, ***greedy heuristics*** are useful for approximating hard problems. (Chap 35)

16.1 An activity-selection problem

- Activity selection problem
  - A set of activities $S = \{a_1, a_2, \ldots, a_n\}$
  - Activity a_i needs resource during period $[s_i, f_i)$, where s_i = start time and f_i = finish time
  - Assume that activities are sorted by finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$
  - Two activities $a_i = [s_i, f_i)$ and $a_j = [s_j, f_j)$ are ***compatible*** if they don't overlap, i.e. $f_i \leq s_j$ or $f_j \leq s_i$.
  - ***Goal***

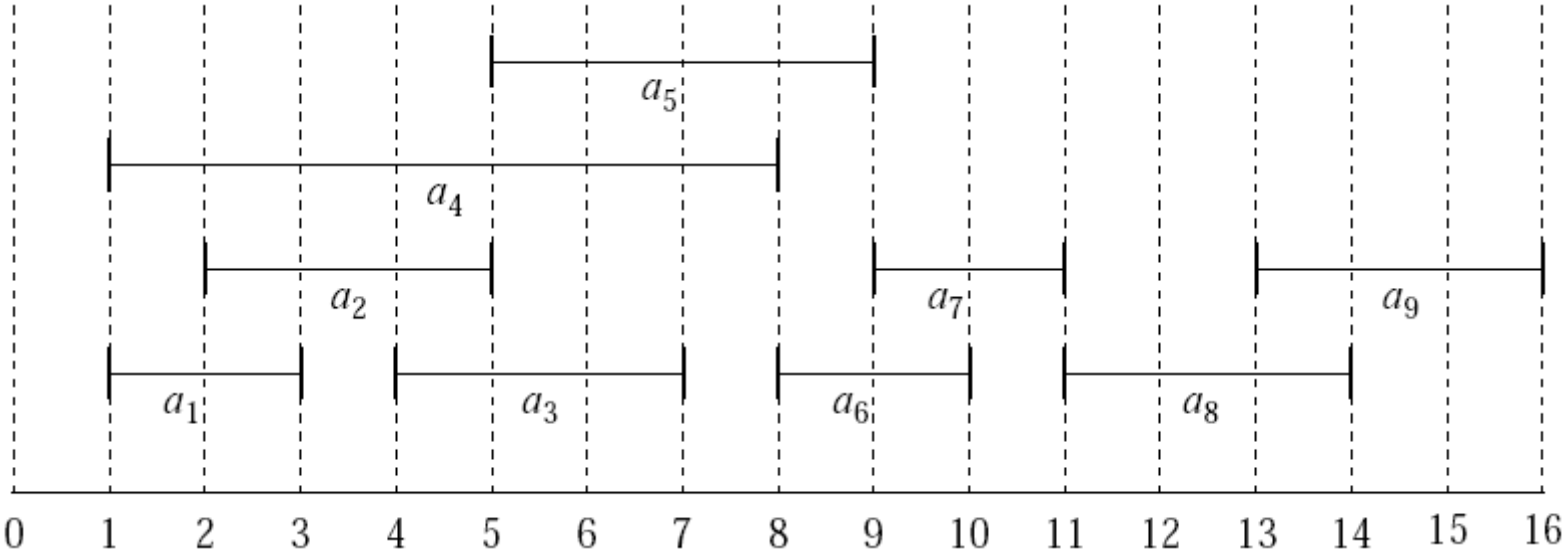
    Select the largest possible set of nonoverlapping (*mutually compatible*) activities

16.1 An activity-selection problem

- Activity selection problem

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

S sorted by finish time



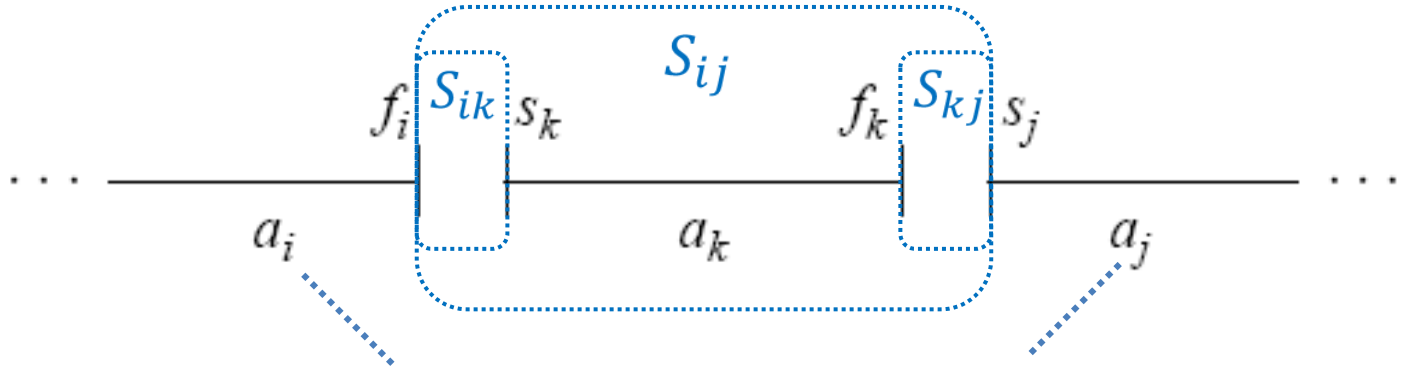
Maximum-size mutually compatible sets

$\{a_1, a_3, a_6, a_8\}, \{a_2, a_5, a_7, a_9\}, \{a_2, a_5, a_7, a_8\}$, etc

16.1 An activity-selection problem

- Optimal substructure

  Let $S_{ij} = \{a_k \in S: f_i \le s_k < f_k \le s_j\}$



  compatible with activities in S_{ij}

  Let A_{ij} be an optimal solution to S_{ij}

  Let $a_k \in A_{ij}$. Then we have two subproblems:

  - Find mutually compatible activities in S_{ik}
  - Find mutually compatible activities in S_{kj}

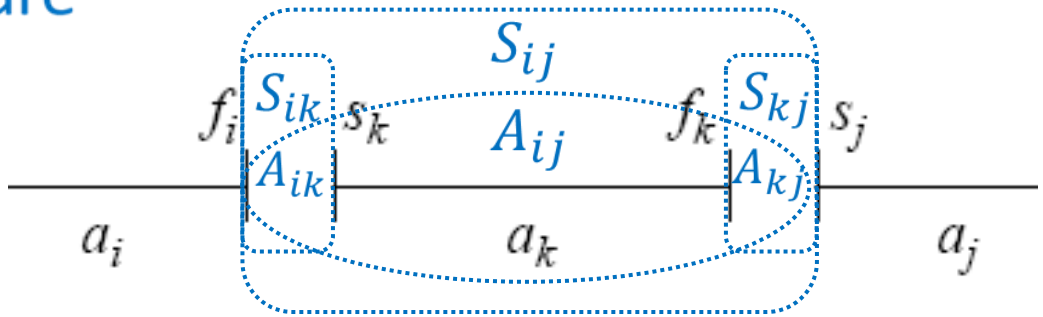
16.1 An activity-selection problem

- Optimal substructure

THEOREM

Let $A_{ik} = A_{ij} \cap S_{ik}$

$A_{kj} = A_{ij} \cap S_{kj}$



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

Proof: Cut-and-paste

If the subproblem S_{ik} has a better solution B_{ik}, i.e. $|B_{ik}| > |A_{ik}|$

Then, $B_{ik} \cup \{a_k\} \cup A_{kj}$ is a solution to the problem S_{ij} that is better than A_{ij}.

Similarity for the subproblem S_{kj}

16.1 An activity-selection problem

- Optimal substructure

  Let $c[i,j] = |A_{ij}|$, then

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

  This gives rise to a dynamic programming solution. (Ex 16.1-1)

  Have to fill in the $O(n^2)$ entities of a table

  Each entity takes $O(n)$ time

  Time in total: $O(n^2) \times O(n) = O(n^3)$

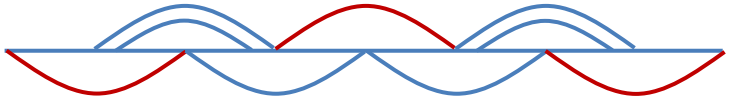
16.1 An activity-selection problem

- Greedy-choice property

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

  Greedy heuristics
  - Choose the first activity to finish. (OK, see below)
  - Choose the last activity to start. (OK, Ex 16.1-2)
  - Choose the activity of least duration (NO, Ex 16.1-3)

  - Choose the activity that overlaps the fewest other activities (NO, Ex 16.1-3)



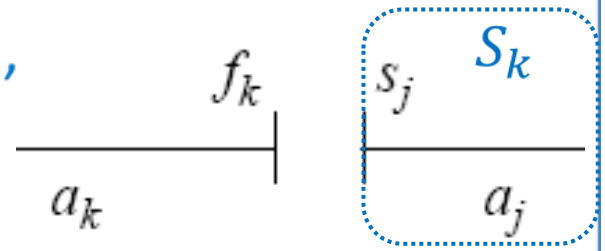
16.1 An activity-selection problem

- Greedy-choice property

  Let a_0 be a fictitious activity with $f_0 = 0$,

  Let $S_k = \{a_j \in S: f_k \leq s_j\}$

  Note that $S_0 = S$



  By the "1st activity to finish" greedy heuristics, a_1 is chosen from S_0.

  Then, S_1 is the only subproblem to solve.

  Comparison

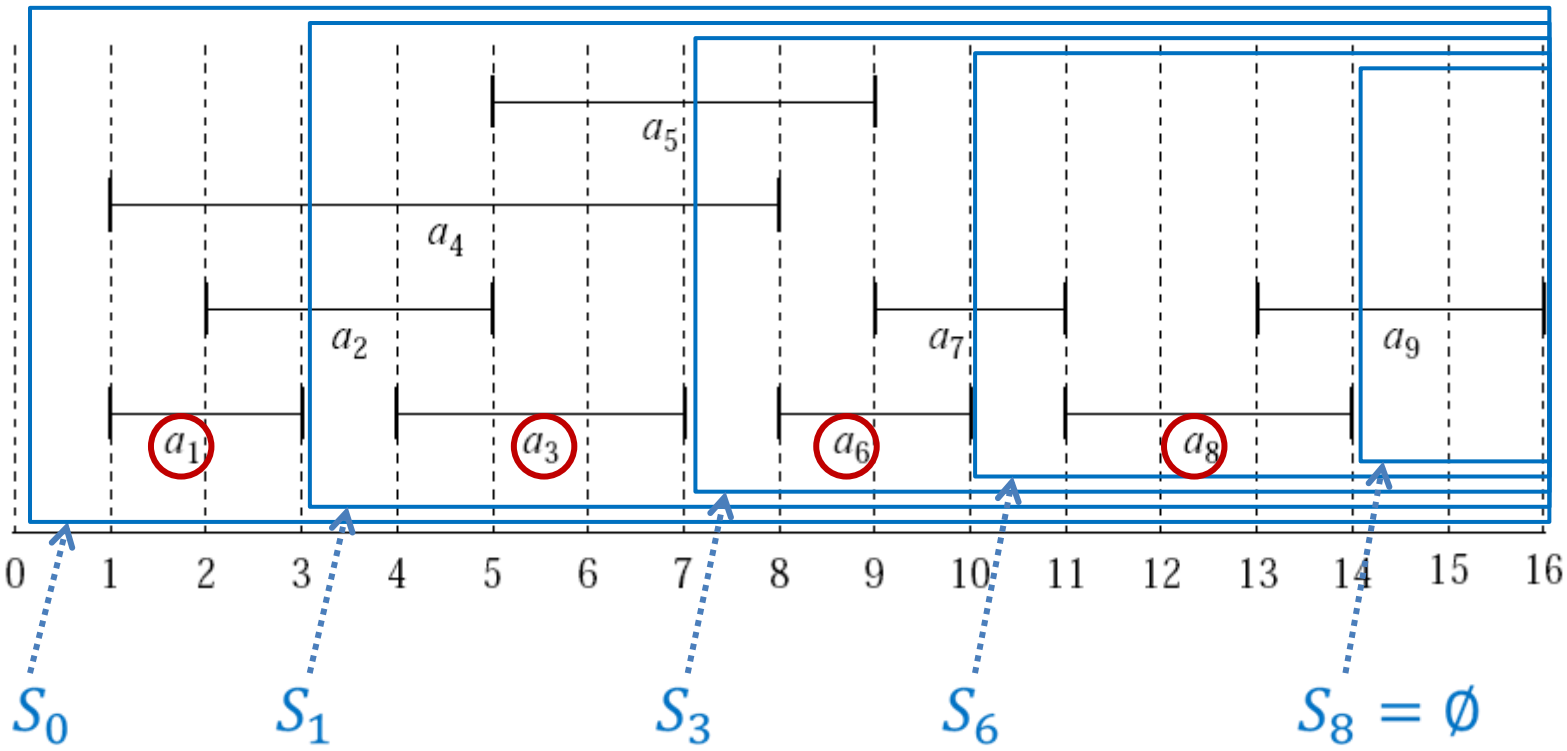
  Greedy: One choice and one subproblem

  Dynamic programming: Several choices and two subproblems.

16.1 An activity-selection problem

- Greedy-choice property

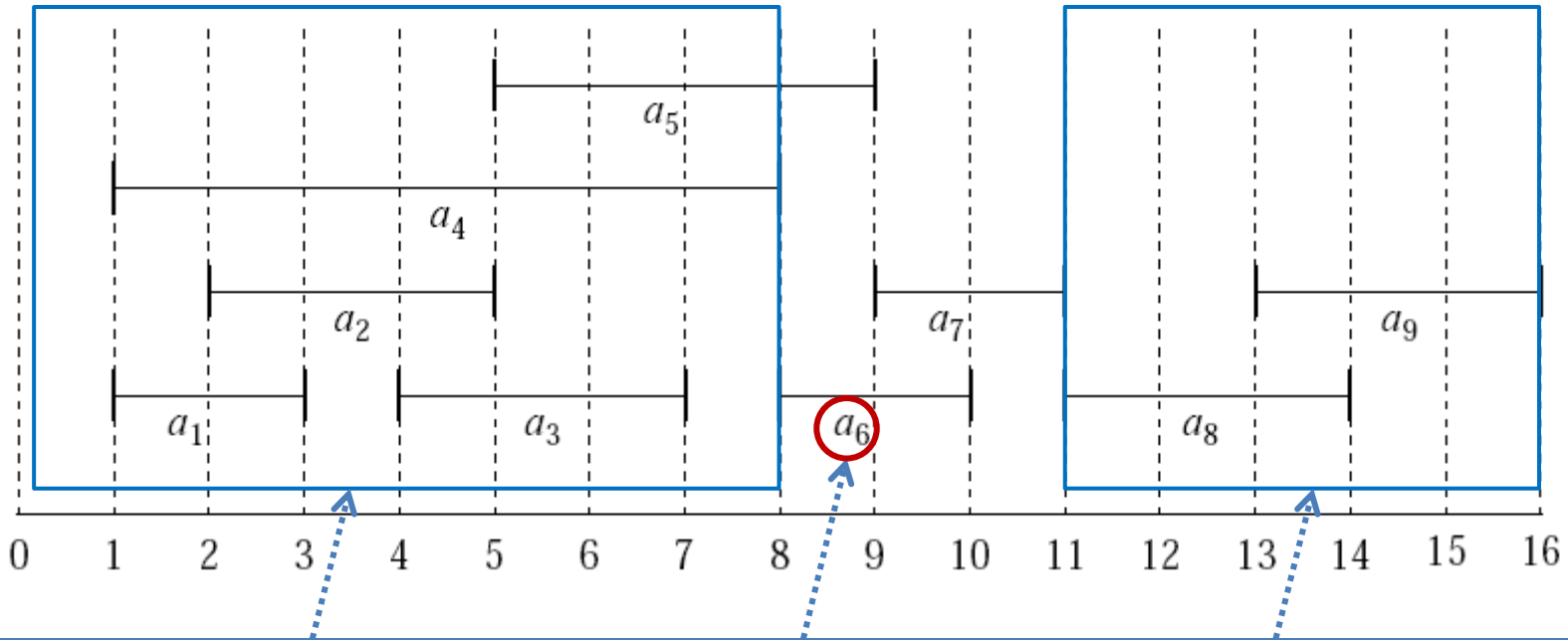
  Example



16.1 An activity-selection problem

- Greedy-choice property

  Example (Cont'd, dynamic programming)



S_{06} $a_6 \in S_{09}$ S_{69}

Have to consider all choices $a_i \in S_{09}$ $\forall i$

16.1 An activity-selection problem

- Greedy-choice property

  Key proof method: ***Transformation***

  If an optimal solution includes the greedy choice, we are done

  Otherwise, transform the optimal solution to another optimal solution that includes the greedy choice.

  THEOREM

  If $S_k \neq \emptyset$ and a_m has the earliest finish time in S_k, then a_m is included in some optimal solution.

  Proof

  Let A_k be an optimal solution to S_k

  Let $a_j \in A_k$ have the earliest finish time of any activity in A_k.

16.1 An activity-selection problem

- Greedy-choice property

  THEOREM (Cont'd)

  If $a_j = a_m$, we are done.

  Otherwise, let $A'_k = A_k - \{a_j\} \cup \{a_m\}$

  CLAIM Activities in A'_k are disjoint.

  ∵ Activities in A_k are disjoint, a_j is the 1st activity in A_k to finish, and $f_m \leq f_j$. ■ (CLAIM)

  Since $|A'_k| = |A_k|$, A'_k is an optimal solution to S_k that includes a_m, as desired.

  E.g. $\{a_2, a_5, a_7, a_9\} \xrightarrow{\text{transforms}} \{a_1, a_5, a_7, a_9\}$ (see p.9)

16.1 An activity-selection problem

- A recursive greedy algorithm

REC-ACTIVITY-SELECTOR(s, f, k, n)

$m = k + 1$ // find 1st activity in S_k to finish

while $m \leq n$ and $s[m] < f[k]$

$m = m + 1$

if $m \leq n$ **then return** $\{a_m\}$ ∪ REC-ACTIVITY-SELECTOR(s, f, m, n)

else return $\emptyset$

Initial call: REC-ACTIVITY-SELECTOR$(s, f, 0, n)$

Time: $\Theta(n)$

Each activity is examined exactly once, assuming that activities are already sorted by finish times.

16.1 An activity-selection problem

- An iterative greedy algorithm

GREEDY-ACTIVITY-SELECTOR(s, f)

$n = s.length$

$A = \{a_1\}$

$k = 1$

for $m = 2$ **to** n

  if $s[m] \geq f[k]$

    $A = A \cup \{a_m\}$

    $k = m$

return A

Time: $\Theta(n)$, if activities are already sorted by finish times.

16.2 Elements of the greedy strategy

- Greedy vs dynamic programming

  Dynamic programming

  - Key ingredient: Optimal substructure
  - Make a choice at each step
  - Choice depends on the optimal solutions to subproblems

    Solve subproblems first
  - Solve bottom-up

  Greedy

  - Key ingredients: Optimal substructure, Greedy-choice property
  - Make a choice at each step

16.2 Elements of the greedy strategy

- Greedy vs dynamic programming

  Greedy (Cont'd)

  - Make the choice before solving the subproblems
  - Solve top-down

- 0/1 and fractional knapsack problem

  The knapsack problem is a good example of the difference.

  0-1 knapsack problem

  - n items

    Item i is worth v_i dollars and weighs w_i pounds.

  - Knapsack capacity: W pounds

    N.B. v_i, w_i and W are all integers

16.2 Elements of the greedy strategy

- 0/1 and fractional knapsack problem

  0-1 knapsack problem (Cont'd)

  - Find a most valuable subset of items with total weight $\leq W$
  - Have to take an item or not take it – can't take part of it
  - Formally,

    Maximize $\sum_{i=1}^{n} v_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \leq W$

    where $x_i = 0$ or 1

  fractional knapsack problem

  - Like 0-1 knapsack, but can take a fraction of an item.

    i.e. $0 \leq x_i \leq 1$

16.2 Elements of the greedy strategy

- 0-1 and fractional knapsack problem

  Both problems have optimal substructure.

  0-1 knapsack

  Let C be an optimal solution for n objects with capacity W

  Let $x_i = 1$, then $C' = C - \{x_i\}$ must be an optimal solution for the remaining $n - 1$ objects with capacity $W' = W - w_i$

  fractional knapsack

  Let C be an optimal solution for n objects with capacity W

  Let $x_i > 0$, then $C' = C - \{x_i\} \cup \{x_i'\}$ where $0 \leq x_i' < x_i$ must be an optimal solution for the remaining $n - 1$ objects plus $1 - x_i'$ portion of item i with capacity $W' = W - w_i x_i'$

16.2 Elements of the greedy strategy

- 0-1 and fractional knapsack problem

  fractional knapsack (Cont'd)

Item i	1	2	3
Value v_i	60	100	120
Weight w_i	10	20	30

$W = 50$

$C = \{x_1 = 1, x_2 = 1, x_3 = 2/3\}$

Item i	1	2	3
Value v_i	60	100	80
Weight w_i	10	20	20

$W' = 40$

$C' = \{x_1 = 1, x_2 = 1, x_3' = 1/3\}$

16.2 Elements of the greedy strategy

- 0-1 and fractional knapsack problem

  Greedy heuristics

  Take as much as possible the item with the maximum value density, i.e. the maximum v_i/w_i.

  This greedy heuristics doesn't work for 0-1 knapsack.

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

$W = 50$

Greedy solution

Take items 1 and 2

Value $= 160$, weight $= 30$

Optimal solution

Take items 2 and 3

Value $= 220$, weight $= 50$

16.2 Elements of the greedy strategy

- 0-1 and fractional knapsack problem

  But, it works for fractional knapsack.

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

Greedy solution

Take items 1, 2

Take 2/3 of item 3

Value = 240, weight = 50

Thus, 0-1 knapsack can be solved by dynamic programming in $\Theta(nW)$ time. (Ex. 16.2-2)

On the other hand, fractional knapsack can be solved by greedy method.

16.2 Elements of the greedy strategy

- Fractional knapsack problem

  Greedy algorithm for the fractional knapsack problem

  FRACTIONAL-KNAPSACK(v, w, W)

  $load = 0$

  $i = 1$

  while $load < W$ and $i \leq n$

  if $w_i \leq W - load$

  take all of item i

  else take $(W - load)/w_i$ of item i

  add what was taken to $load$

  $i = i + 1$

  Time: $O(n \lg n)$ to sort, $O(n)$ thereafter

16.2 Elements of the greedy strategy

- Fractional knapsack problem

  Ex 16.2-1

  Prove that the fractional knapsack problem has the greedy-choice property.

  THEOREM (Greedy-choice property of fractional knapsack)

  Assume that $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$

  There is an optimal solution with $x_1 = \min\{1, W/w_1\}$, i.e. if $w_1 \leq W$ then $x_1 = 1$ else $x_1 = W/w_1$

  Note: Not every optimal solution has $x_1 = \min\{1, W/w_1\}$.

  For example, $W = v_1 = w_1 = v_2 = w_1 = 1$ has infinitely many optimal solutions as long as $x_1 + x_2 = 1$

16.2 Elements of the greedy strategy

- Fractional knapsack problem

  THEOREM (Cont'd)

  Proof

  We shall assume that $w_1 \leq W$ and show that there is an optimal solution with $x_1 = 1$. (The proof for $w_1 > W$ is similar.)

  Let $C = \{x_1, x_2, \ldots, x_n\}$ be an optimal solution

  If $x_1 = 1$, we are done.

  So, assume that $x_1 < 1$

  Then, $\exists x_i > 0, 2 \leq i \leq n$

  Otherwise, $C' = \{1,0, \ldots, 0\}$ is better than $C = \{x_1, 0, \ldots, 0\}$

16.2 Elements of the greedy strategy

- Fractional knapsack problem

  THEOREM (Cont'd)

  Case 1: $w_1(1-x_1) \le w_i x_i$

  Let $x_1' = 1$

  $$x_i' = x_i - w_1(1-x_1)/w_i$$

  $$C' = C - \{x_1, x_i\} \cup \{x_1', x_i'\}$$

  Then, value of C' − value of C

  $$= v_1(1-x_1) - v_i\, w_1(1-x_1)/w_i$$

  $$= (1-x_1)(v_1 - v_i\, w_1/w_i)$$

  $$\ge 0 \qquad \because v_1/w_1 \ge v_i/w_i \Rightarrow v_1 \ge v_i\, w_1/w_i$$

  Since C is optimal, value of $C' =$ value of C and C' optimal with $x_1' = 1$, as desired.

16.2 Elements of the greedy strategy

- Fractional knapsack problem

  THEOREM (Cont'd)

  Case 2: $w_1(1 - x_1) > w_i x_i$

  Let $x_1' = x_1 + w_i x_i / w_1$

  $x_i' = 0$

  $C' = C - \{x_1, x_i\} \cup \{x_1', x_i'\}$

  Then, value of C' − value of C

  $= v_1 w_i x_i / w_1 - v_i x_i$

  $= x_i(v_1 w_i / w_1 - v_i)$

  ≥ 0 $\quad \because v_1/w_1 \geq v_i/w_i \Rightarrow v_1 w_i / w_1 \geq v_i$

  Since C is optimal, value of $C' =$ value of C and C' optimal with $x_1' < 1$. Note that x_1' is nearer to 1 than x_1.

16.2 Elements of the greedy strategy

- Fractional knapsack problem

  THEOREM (Cont'd)

  By applying the transformation a finite number time, we will eventually reach an optimal solution.

  Why?

  If case 1 is met, we are done.

  Otherwise, there are at most $n - 1$ case-2 transformations, since each case-2 transformation sets one x_i to 0.

  So, the transformation process will terminate, and it will terminate with $C' = \{1,0,\dots,0\}$

16.3 Huffman codes

- Prefix code

  Fixed-length code

  2 bits for each character

  Total # of bits:

  $2 \times (1 + 2 + 3 + 4) = 20$

  Variable-length code

  Frequent characters have short codes

  Total # of bits:

  $3 \times 1 + 3 \times 2 + 2 \times 3 + 1 \times 4 = 19$

  Prefix codes

  No code is a prefix of some other code.

Character	Frequency
a	1
b	2
c	3
d	4

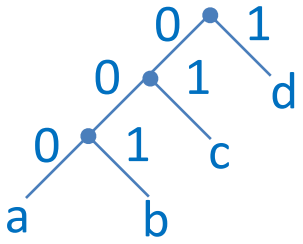
Character	Code
a	000
b	001
c	01
d	1

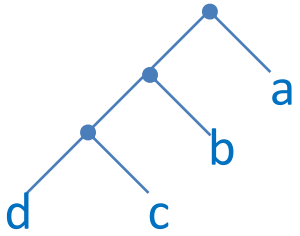
16.3 Huffman codes

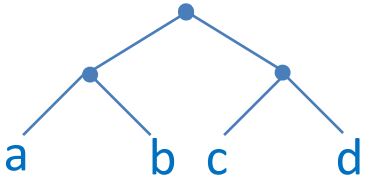
- Prefix code

  Prefix code tree – By assigning left and right branches 0 and 1, respectively, a prefix code tree defines a prefix code.

Optimal







1 Prefix codes are not unique – looking for an optimal code

2 Even optimal prefix codes are not unique – a branch may be assigned either 0 or 1.

3 The decoding process is based on the prefix code tree.
e.g. 000 001 01 1 ⇒ a b c d

16.3 Huffman codes

- Huffman code

  Huffman's greedy algorithm constructs an optimal prefix code

  HUFFMAN(C)

  $n = |C|;\ Q = C$ // Q is a min-priority queue; time $O(n)$

  for $i = 1$ **to** $n - 1$

  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)

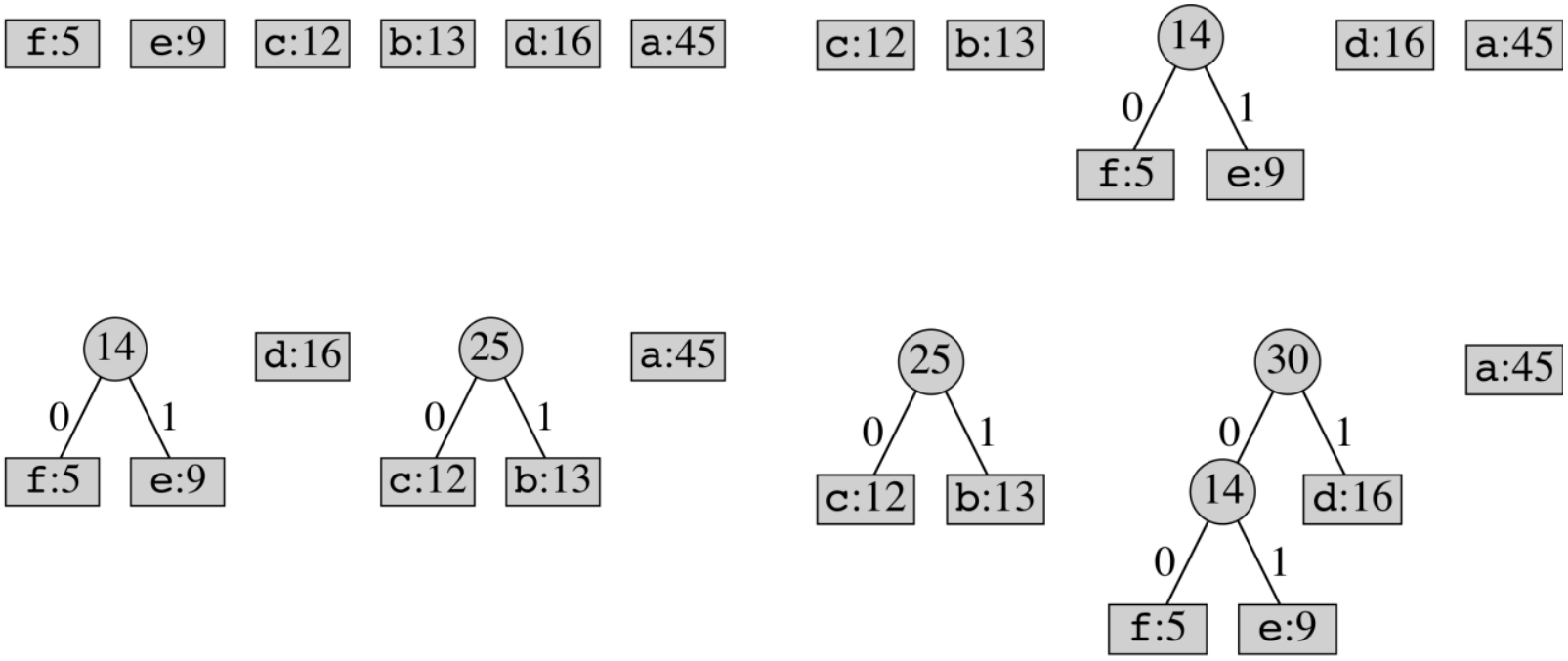
  return EXTRACT-MIN(Q) // return the root of the tree

  Running time: $O(n) + (n-1)O(\lg n) = O(n \lg n)$

16.3 Huffman codes

- Huffman code

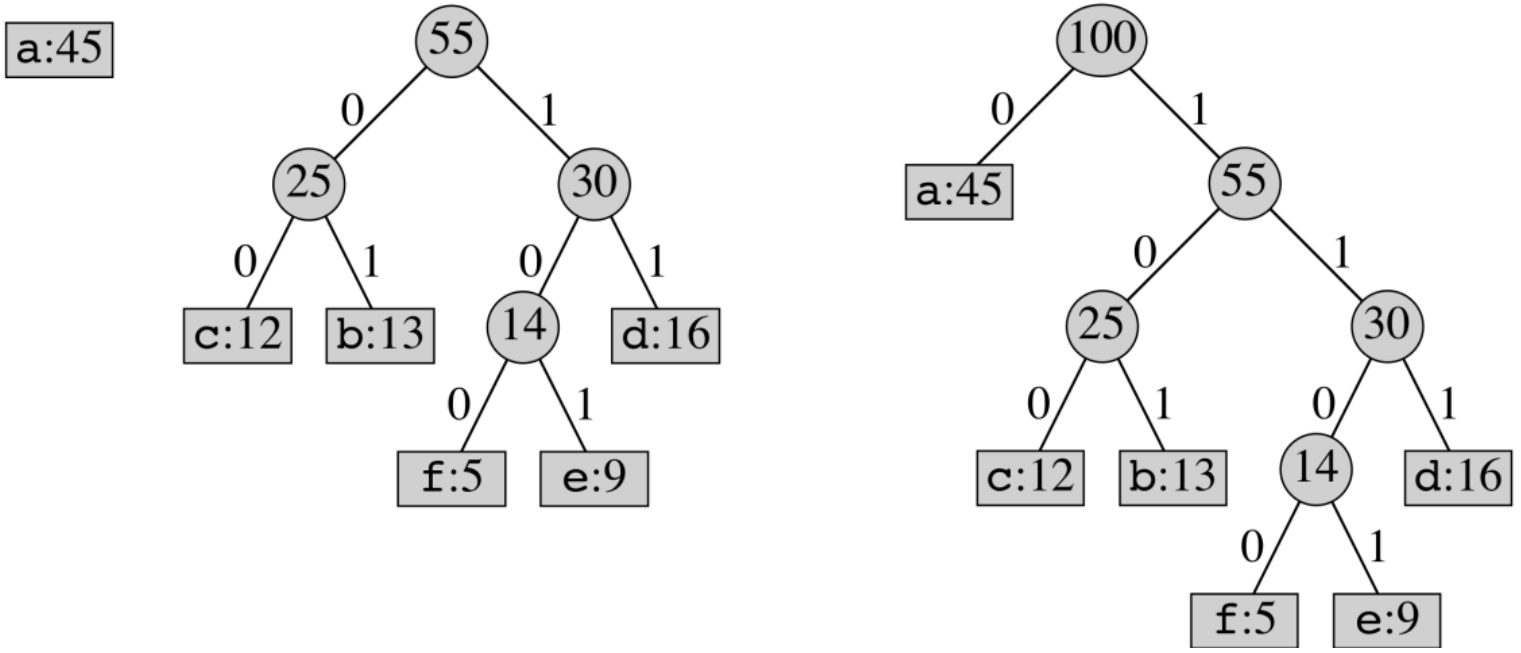
  Example



16.3 Huffman codes

- Huffman code

  Example (Cont'd)



16.3 Huffman codes

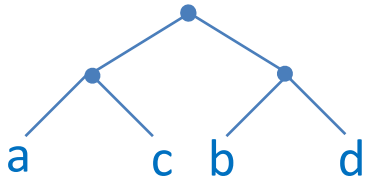
- Huffman code

  Comment

  - The Huffman tree actually built depends on how the algorithm is implemented: how to break ties and which of the two EXTRACT-MIN(Q)'s goes to $left[z]$.
  - Huffman trees are optimal prefix code trees.

    But, optimal prefix code trees needn't be Huffman trees.

    For example, given a/2, b/2, c/3, d/3



    This prefix code tree is optimal, but it isn't a Huffman tree.

16.3 Huffman codes

- Greedy-choice property

  Let T be a prefix code tree

  Let $d_T(c)$ = the depth of character c in T

  Define

  $$B(T) = \sum_{c \in C} f(c) d_T(c) = \#\text{ of bits required for the encoding}$$

  LEMMA An optimal prefix code tree is a full binary tree (i.e. every internal node has two children.)

  Proof

  Suppose not.

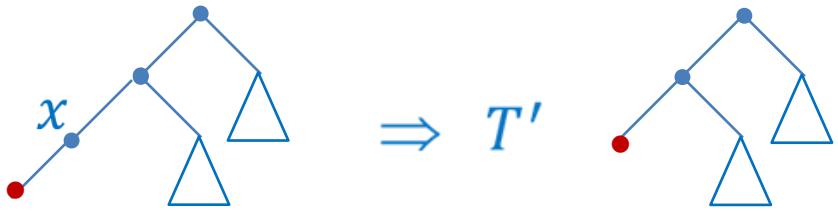
  Let x be an internal node having one child

16.3 Huffman codes

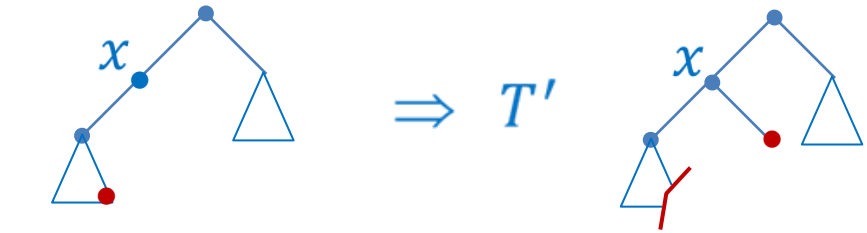
- Greedy-choice property

  LEMMA (Cont'd)

  Case 1: The child is a leaf. Remove x as follows:



  Case 2: The child isn't a leaf. Do this:



  In either case, $B(T) > B(T')$, because the depth of the red-colored node decreases. A contradiction.

16.3 Huffman codes

- Greedy-choice property

  LEMMA (Greedy-choice property)

  Let x and y be two characters having the lowest frequencies. Then, there is an optimal prefix code tree in which x and y are siblings.

  Proof

  Let T be an optimal prefix code tree

  If x and y are siblings in T, we are done.

  Otherwise, let a and b be two characters that are siblings of the maximum depth in T. (a and b exist, by the preceding lemma)

16.3 Huffman codes

- Greedy-choice property

  LEMMA (Cont'd)

  Without loss of generality, assume $f(x) \le f(y), f(a) \le f(b)$

  We have

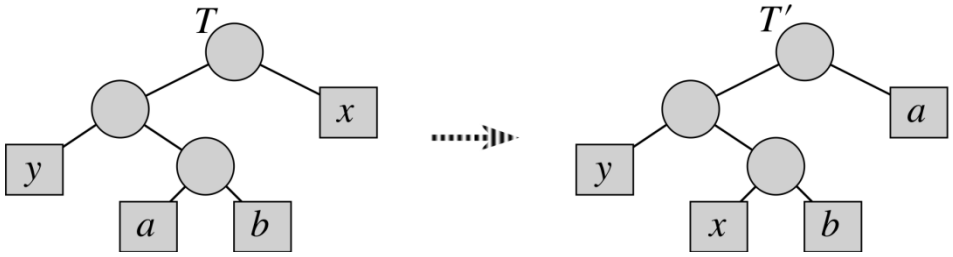
  1 $f(x) \le f(a), f(y) \le f(b)$

  $\because$ x and y have the lowest frequencies

  2 $d_T(x) \le d_T(a), d_T(y) \le d_T(b)$

  $\because$ a and b are of maximum depth

  Transform T to T' by swapping x and a



16.3 Huffman codes

- Greedy-choice property

  LEMMA (Cont'd)

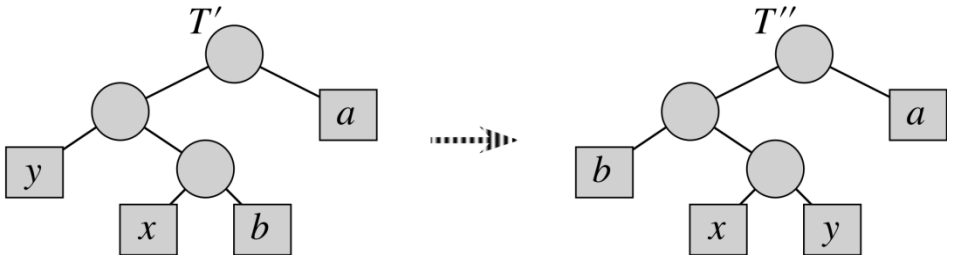
  $B(T) - B(T')$

  $= f(x)(d_T(x) - d_T(a)) + f(a)(d_T(a) - d_T(x))$

  $= (f(a) - f(x))(d_T(a) - d_T(x)) \geq 0$

  Since T is optimal, $B(T) = B(T')$ and T' is optimal, too.

  Next, transform T' to T'' by swapping y and b



  Silimarly, $B(T') = B(T'')$ and T'' is the desired optimal prefix code tree.

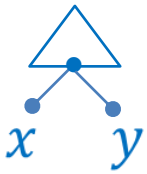
16.3 Huffman codes

- Optimal substructure

LEMMA

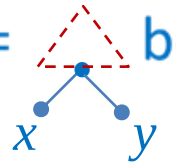
Let $x, y \in C$ be two characters with minimum frequencies

Let $C' = C - \{x, y\} \cup \{z\}$, where z is a new character with $f(z) = f(x) + f(y)$

Let T' =  be optimal for C', then T =  is optimal for C

Proof

Suppose T isn't optimal for C

Let T_1 =  be optimal for C (N.B.  is different from  .)

(By the greedy-choice property, T_1 exists.)

16.3 Huffman codes

- Optimal substructure

  LEMMA (Cont'd)

  Let $T_1' =$ 

  Then,

  $B(T) - B(T') = B(T_1) - B(T_1') = f(z)$

  ∵ In both trees, (depth of x and y − depth of z) $= 1$, and

  $f(x) + f(y) = f(z)$

  Thus,

  $B(T) - B(T_1) = B(T') - B(T_1') > 0$

  contradicting that T' is optimal for C'.