Greedy Strategy And Dynamic Programming

A short list of categories so far covered

- Simple recursive algorithms
- Divide and conquer algorithms
- Greedy algorithms
- Dynamic Programming

2

Contents

- Knapsack problem
- Huffman code generation algorithm
- Job Sequencing with Deadlines

3

Optimization problems

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A “greedy algorithm” sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

4

Introduction : Greedy Method

- A greedy algorithm for an optimization problem always makes the choice that looks best at the moment and adds it to the current subsolution.

- Final output is an optimal solution.

- Greedy algorithms don't always yield optimal solutions but, when they do, they're usually the simplest and most efficient algorithms available.

Introduction : Greedy Method

- It is used to solve problems that have ‘n’ inputs and require us to obtain a subset that **satisfies some constraints**.
- Any subset that satisfied the constraints is called as a **feasible** solution.
- We need to find the optimum feasible solution i.e. the feasible solution that optimizes the given **objective functions**.
- It works in stages. At each stage, At each stage a decision is made regarding weather or particular input is in the optimum solution.

6

Introduction : Greedy Method

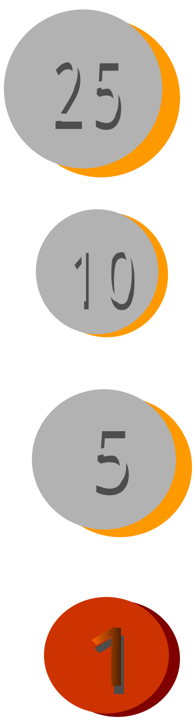
- The greedy method suggests that one can divide the algorithm that works in stages. Considering one input at a time.
- At each stage a decision is made regarding weather or particular input is in the optimum solution.
- For this purpose all the inputs must be arranged **in a particular order by using a selection process**.
- If the inclusion of next input in to the partially constructed solution will result in an infeasible solution then the input will not be considered and will not be added to partially constructed set otherwise it is added.
- CHANGE-MAKING PROBLEM (coin changing)

Coin Changing

An optimal solution to the coin changing problem is the minimum number of coins whose total value equals a specified amount. For example what is the minimum number of coins (current U.S. mint) needed to total 83 cents.

Objective function: Minimize number of coins returned.

Greedy solution: Always return the largest coin you can




```
83 - 25 = 58
58 - 25 = 33
33 - 25 =  8
 8 -  5 =  3
 3 -  1 =  2
 2 -  1 =  1
 1 -  1 =  0
```

A Greedy Algorithm for Coin Changing

1. Set remval=initial_value

2. Choose largest coin that is less than remval.

3. Add coin to set of coins and set remval:=revamal-coin_value

4. repeat Steps 2 and 3 until remval = 0;

Greedy Method

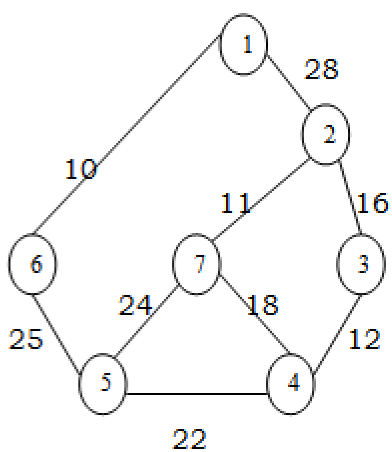
- **Feasible Solution:-**

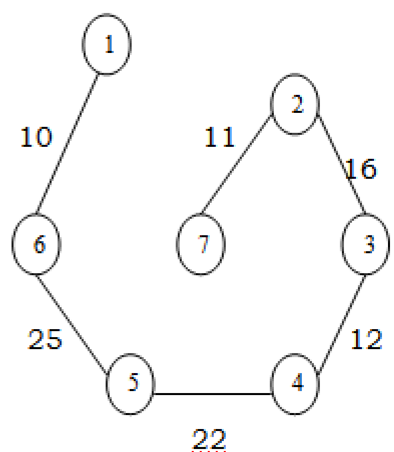
  Any subset of the solutions that satisfies the constraints of the problem is known as a feasible solution.

- **Optimal Solution:-**

  The feasible solution that maximizes or minimizes the given objective function is an optimal solution.

Example : MST





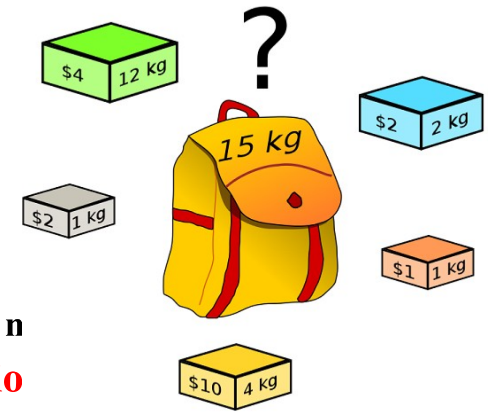
Control Abstraction : Greedy Method

```
Algorithm Greedy (a,n)
 //a[1:n] contains the 'n' inputs
{
    Solution: = 0; //initialize the solution
    for i: = 1 to n do
    {
                x: = select (a);
                if Feasible (solution, x) then
                solution: = Union (solution, x);
    }
      return solution;
}
```

- **Select** is a function which is used to select an input from the set.
- **Feasible** is a function which verifies the constraints and determines weather the resultant solution is feasible or not.
- **Union** is a function which is used to add elements to the partially constructed set.

Knapsack Problem

- **Given :**
  - A knapsack with capacity ‘**m**’
  - A set of ‘**n**’ objects with each item i having
  - $\mathbf{p_i}$ - a positive benefit
  - $\mathbf{w_i}$ - a positive weight
  - **Goal:** Choose items with maximum total benefit but with weight at most **n**



- If we are allowed to take fractional amounts, then this is the **fractio**
  - Let $\mathbf{x_i}$ denote the amount we take of item i , $\mathbf{0} \leq x_i \leq \mathbf{1}$
  - **Objective:** **maximize** $\sum_{1 \leq i \leq n} \mathbf{p_i x_i}$
  - **Constraint:** $\sum_{1 \leq i \leq n} \mathbf{w_i x_i \leq m}$

    and $0 \leq \mathbf{x_i} \leq 1$, $1 \leq \mathbf{i} \leq n$

Knapsack Problem : Example

Ex: - Consider following instance of Knapsack Problem with 3 objects whose profits and weights are defined as

$(P_1, P_2, P_3) = (25, 24, 15)$ $(W_1, W_2, W_3) = (18, 15, 10)$

n=3 Knapsack Capacity **m**=20

Determine the optimum strategy for placing the objects in to the knapsack.

The four feasible solutions are :

(1) Greedy about profit: -

(x_1, x_2, x_3)	Profit Order	Weight Order	$\sum$ xiwi <=20	$\sum$ xipi
(1, 2/15, 0)	(p1,p2,p3) (25,24,15)	(w1,w2,w3) (18, 15, 10)	18 x 1+(2/15) x 15 = 20	25 x 1+ (2/15) x 24 = 28.2

(2) Greedy about weight: -

(x_1, x_2, x_3)	Profit Order	Weight Order	$\sum$ xiwi <=20	$\sum$ xipi
(0, 2/3, 1)	(p3,p2,p1) (15, 24,25)	(w3,w2,w1) (10,15,18)	18 x 0+(2/3) x 15+10 = 20	25 x0+ (2/3) x 24 +25= 31

(3) Greedy about profit / unit weight: -

(x_1, x_2, x_3)	$\sum$ xiwi <=20	$\sum$ xipi
(0, 1, ½)	18 x 0+1 x 15+(1/2)x10 = 20	25 x 0+ 1 x 24+(1/2)x15 = 31.5

(4) If an additional constraint of including each and every object is placed then the greedy strategy could be

(½, ⅓, ¼) **$\sum$ xiwi =** ½ x 18+⅓ x15+ ¼ x10 = 16. 5

$\sum$ xipi =½ x 25+⅓ x24+ ¼ x15 = 12.5+8+3.75 = 24.25

(x_1, x_2, x_3)	$\sum$ xiwi <=20	$\sum$ xipi
(½, ⅓, ¼)	16.5	24.25

Algorithm : Greedy Knapsack

Algorithm Greedy knapsack (m, n)

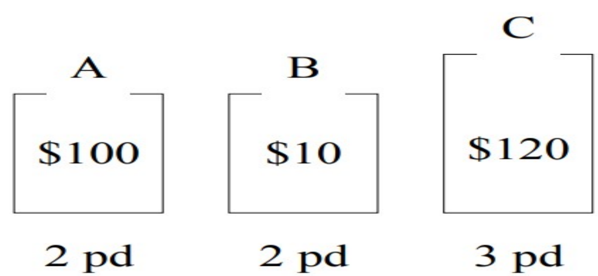
```
//p (1:n) and w(1:n) contain the profits and weights resp. of the n objects
//ordered such that p(i)/W(i) ≥ p(i+1)/w (i+1). M is the knapsack size and x (1:n)
// is the solution vector
{
    for  i: = 1 to n do x(i) = 0.0i// initialize x                O(n)
   u : = m;
   for  i: = 1 to n do                                             O(n)
   {
            if (w(i) > u) then break;
            x(i): = 1.0;
            u: = u-w(i);
   }
    if (i≤n) then x(i): = u/w(i);
}
```

- **Analysis**: - If. we do not consider the time considered for sorting the inputs then the complexity will be **O(n).**

Knapsack : More examples

1) Greedy approach- consider the following instances of knapsack problem n=5, w=100, W (10, 20, 30, 40, 50) , V(20, 30, 66, 40, 60), find the optimal solution.

2) A thief enters a store and sees the following items:



His Knapsack holds 4 pounds. What should he steal to maximize profit?

3) Let us consider that the capacity of the knapsack ***W* =60** and the list of provided items are shown in the following table −

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24

Huffman Coding

- Huffman codes can be used to compress information
  - JPEGs do use Huffman as part of their compression process

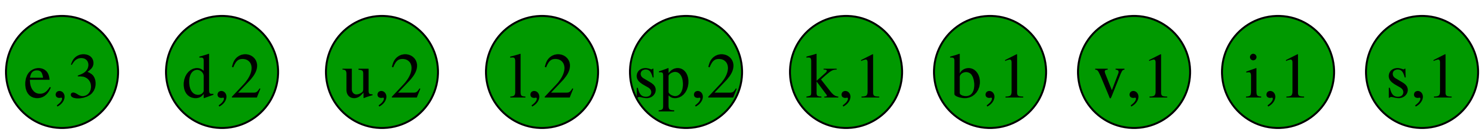
- The basic idea is to store the more frequently occurring characters using fewer bits and less frequently occurring characters using more bits
  - On average this should decrease the filesize (usually ½)

Huffman Coding

- As an example, lets take the string:

  "duke blue devils"

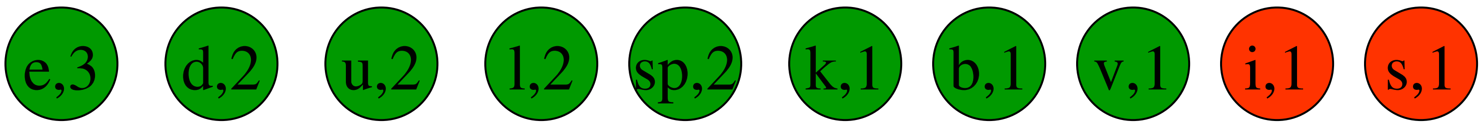
- We first to a frequency count of the characters:
  - e:3, d:2, u:2, l:2, space:2, k:1, b:1, v:1, i:1, s:1
- Next we use a Greedy algorithm to build up a Huffman Tree
  - We start with nodes for each character



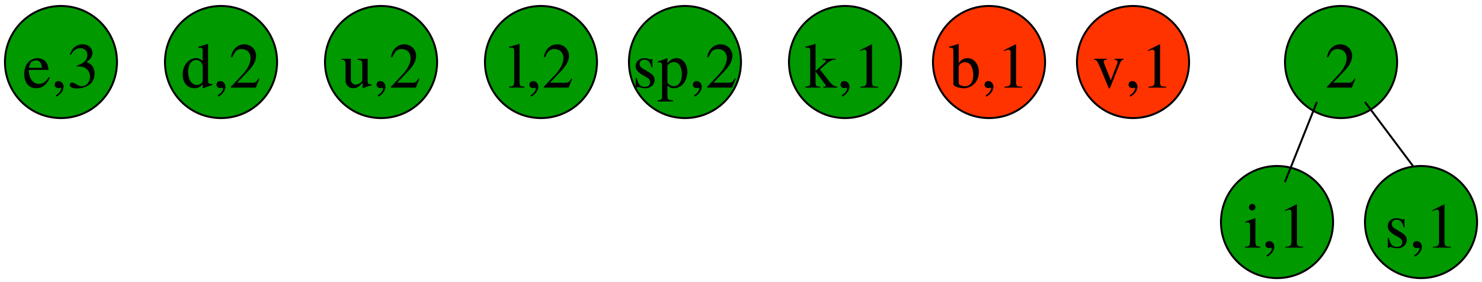
Huffman Coding

- We then pick the nodes with the smallest frequency and combine them together to form a new node
  - **The selection of these nodes is the Greedy part**
- The two selected nodes are removed from the set, but replace by the combined node
- This continues until we have only 1 node left in the set

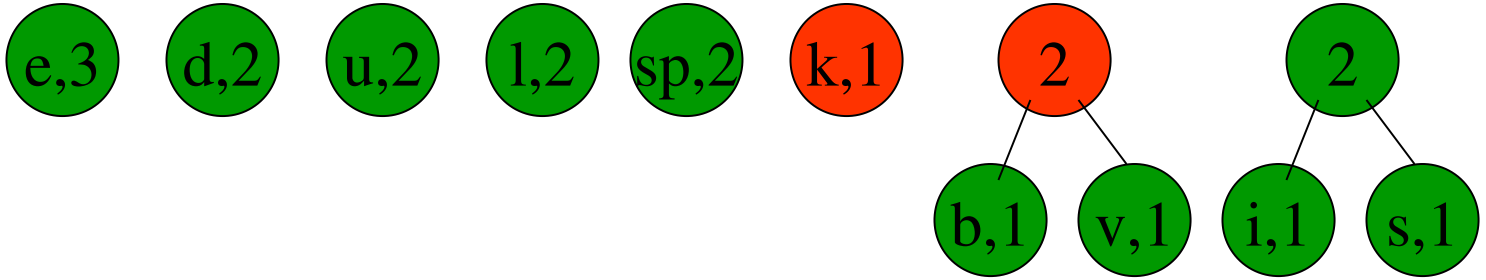
Huffman Coding



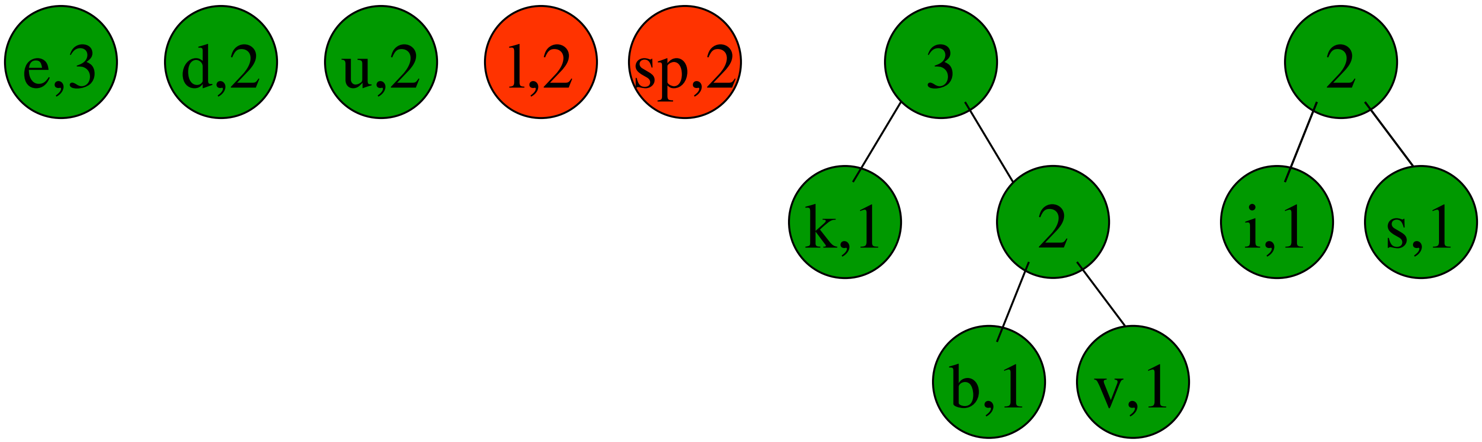
Huffman Coding



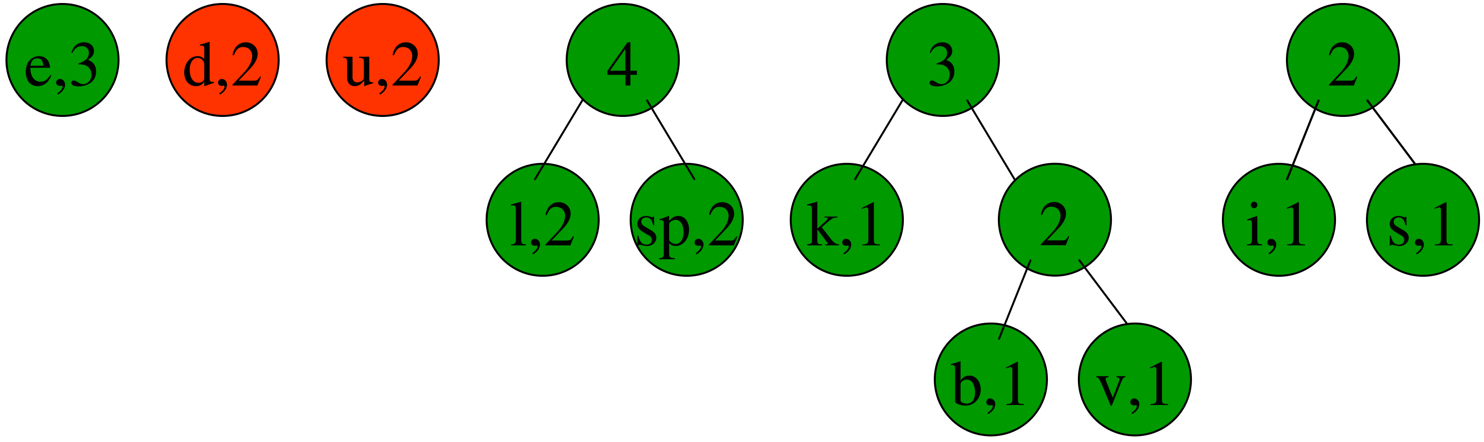
Huffman Coding



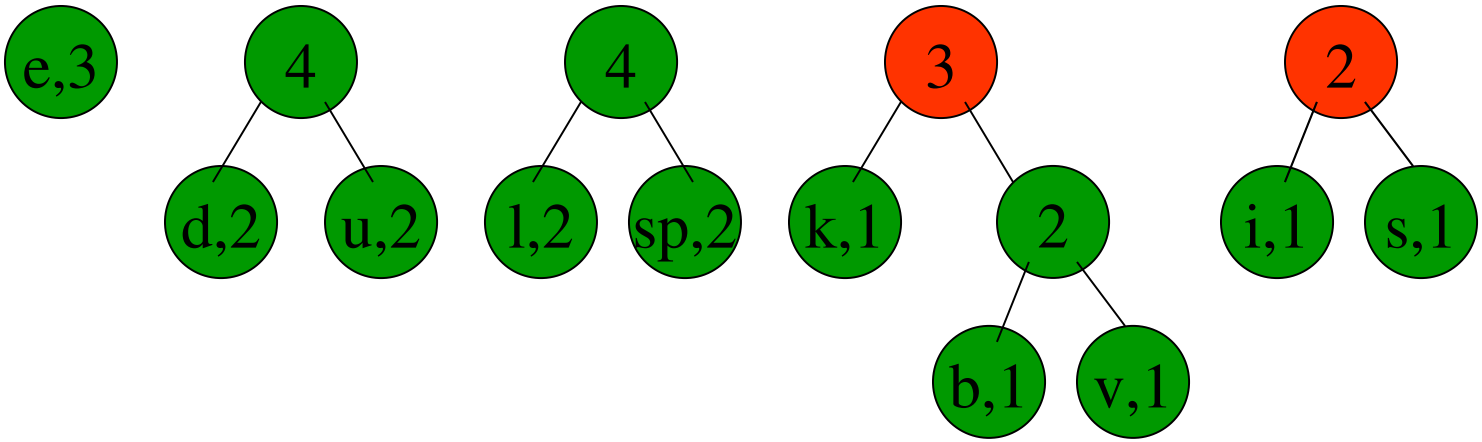
Huffman Coding



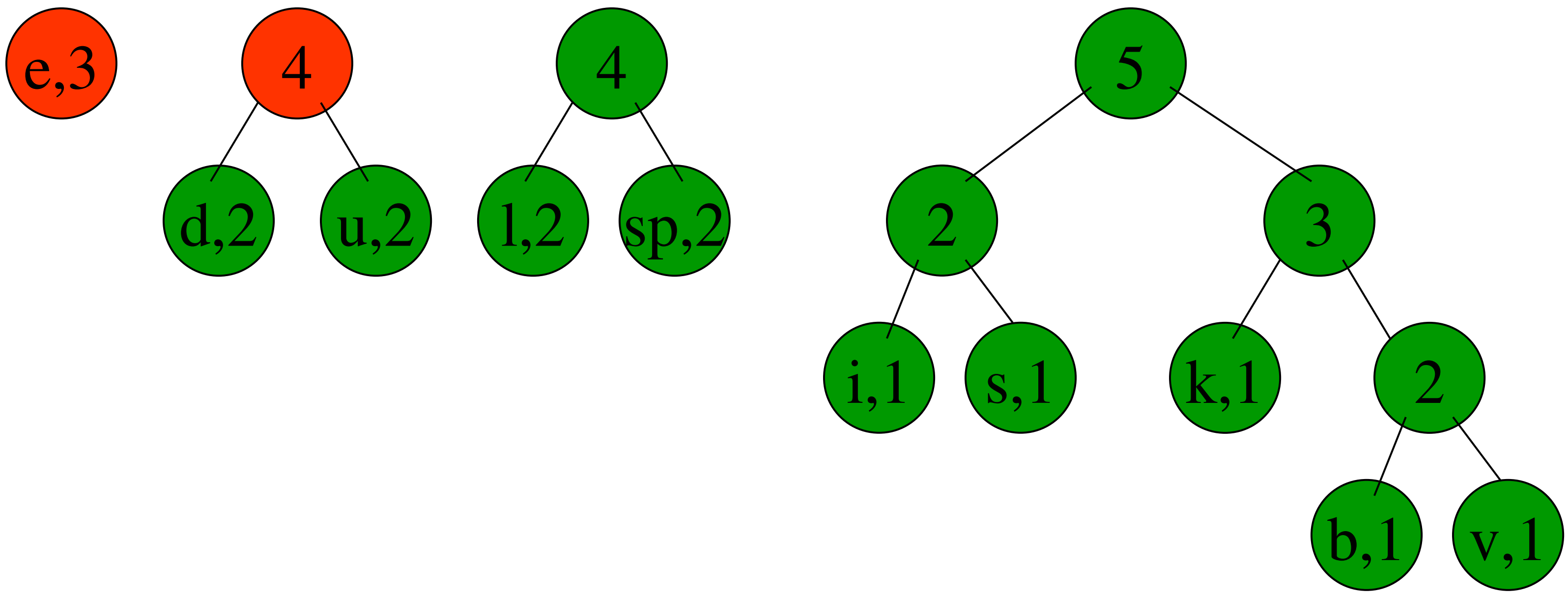
Huffman Coding



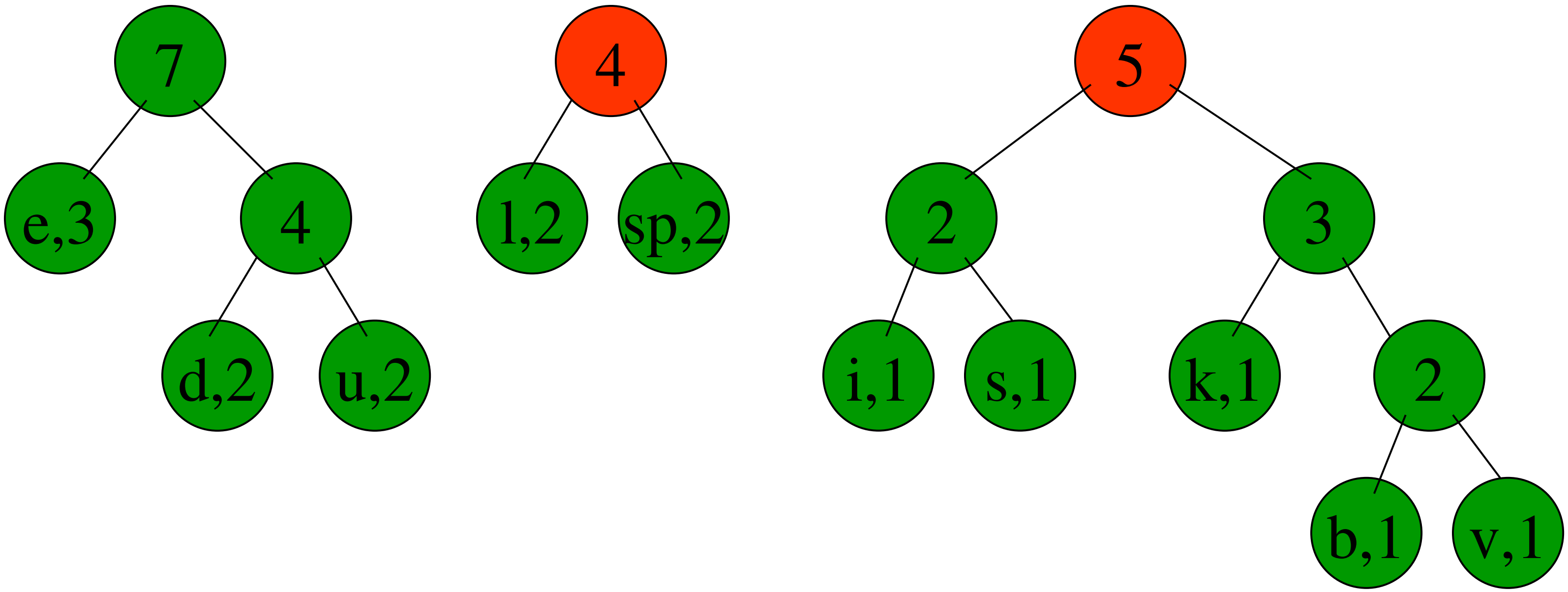
Huffman Coding



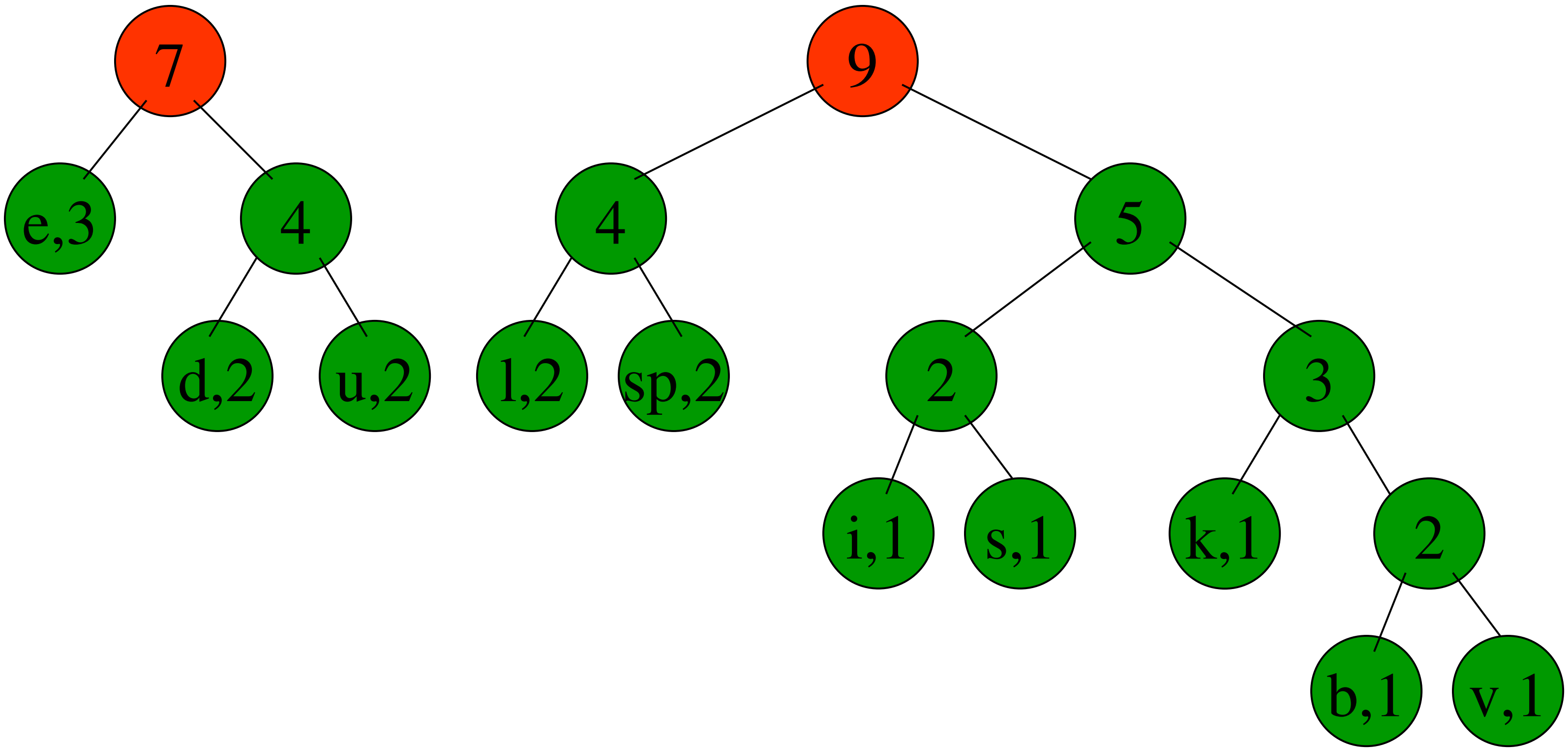
Huffman Coding



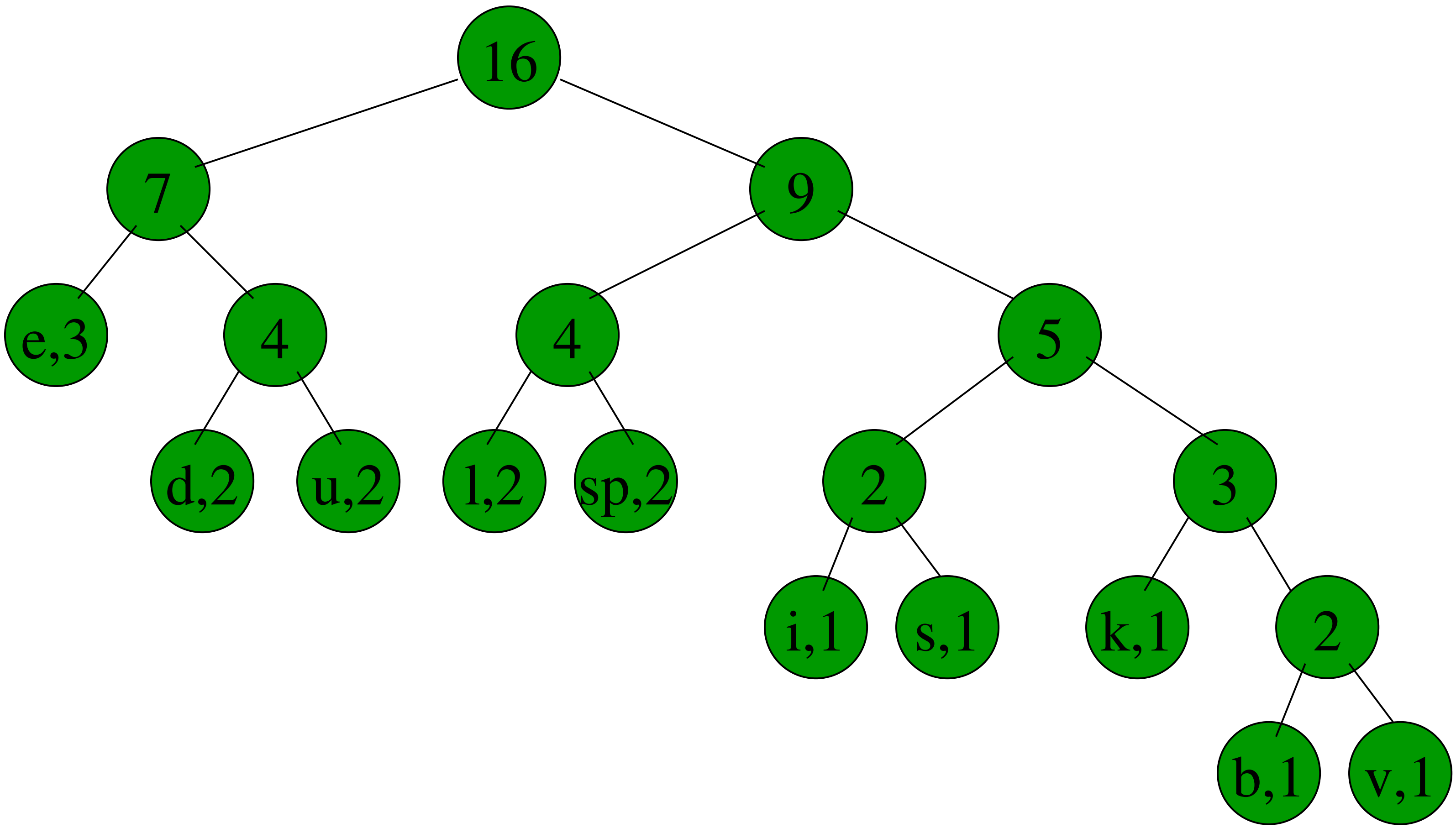
Huffman Coding



Huffman Coding



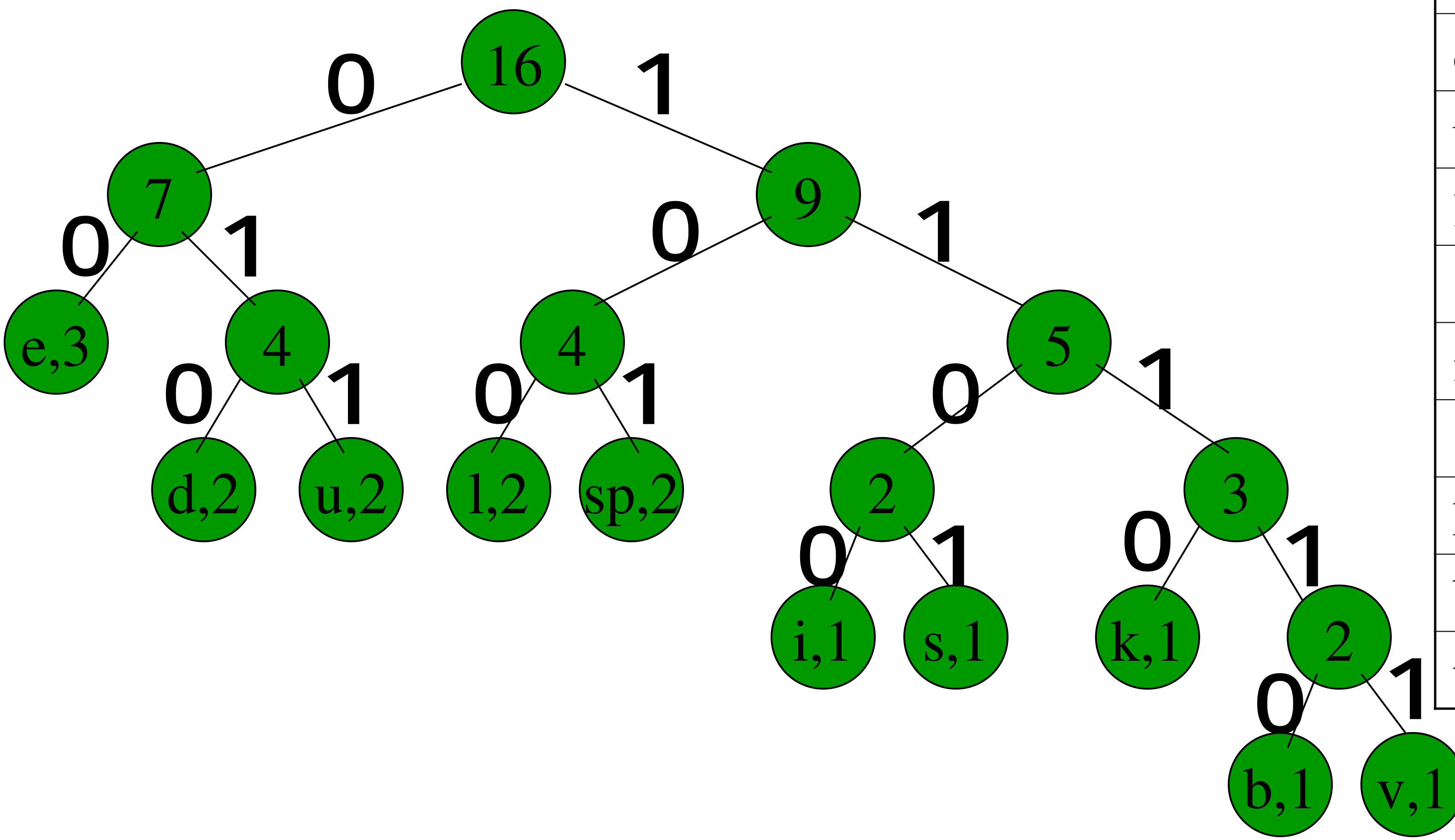
Huffman Coding



Huffman Coding

- Now we assign codes to the tree by placing a 0 on every left branch and a 1 on every right branch
- A traversal of the tree from root to leaf give the Huffman code for that particular leaf character
- Note that no code is the prefix of another code

Huffman Coding



e	00
d	010
u	011
l	100
sp	101
i	1100
s	1101
k	1110
b	11110
v	11111

* Using variable length encoding

Huffman Encoding cormen

HUFFMAN (C)

1. n= |C|

//initializes the min-priority queue Q with the characters in C

2. Q = C O(n)

//loop extracts the two nodes x and y of lowest frequency from the queue

3. **for i = 1 to n – 1** O(n log n)

//replacing them in the queue with a new node z representing their merger

4. allocate a new node z

5. *z.left = x = EXTRACT-MIN(Q)* O(log n)

6. *z.right = y = EXTRACT-MIN(Q)*

7. *z.freq = x:freq + y:freq*

8. INSERT (Q,z)

//After n-1 mergers, line 9 returns the one node left in the queue,which is the root of the code tree

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

Analysis : Must sort n values before making n choices. Therefore, Huffman is O(n log n) + O(n) = O(n log n)

Huffman Coding

- These codes are then used to encode the string

- Thus, “duke blue devils” turns into:

010 011 1110 00 101 11110 100 011 00 101 010 00 11111 1100 100 1101

- When grouped into 8-bit bytes:

01001111 10001011 11101000 11001010 10001111 11100100 1101xxxx

- Thus it takes 7 bytes of space compared to 16 characters * 1 byte/char = 16 bytes uncompressed

Examples : Huffman Coding

- Solve the following using Huffman's code generation algorithm

Symbol	Probability
a	0.12
b	0.04
c	0.45
d	0.16
e	0.23

Huffman Coding Algorithm Example
Construct a Huffman tree by using these nodes

Value	A	B	C	D	E	F
Frequency	5	25	7	15	4	12

Other greedy algorithms

- Dijkstra's algorithm for finding the shortest path in a graph
  - Always takes the *shortest* edge connecting a known node to an unknown node
- Kruskal's algorithm for finding a minimum-cost spanning tree
  - Always tries the *lowest-cost* remaining edge
- Prim's algorithm for finding a minimum-cost spanning tree
  - Always takes the *lowest-cost* edge between nodes in the spanning tree and nodes not yet in the spanning tree

35

JOB SEQUENCING WITH DEADLINES

The problem is stated as below.

- There are n jobs to be processed on a machine.
- Each job i has a deadline $d_i \geq 0$ and profit $p_i \geq 0$.
- Pi is earned iff the job is completed by its deadline.
- The job is completed if it is processed on a machine for unit time.
- Only one machine is available for processing jobs.
- Only one job is processed at a time on the machine.

36

JOB SEQUENCING WITH DEADLINES (contd..)

- A feasible solution is a subset of jobs J such that each job is completed by its deadline.
- An optimal solution is a feasible solution with maximum profit value.
- NO LATER THAN THEIR RESPECTIVE DEADLINES.

Example : Let n = 4, $(p_1,p_2,p_3,p_4) = (100,10,15,27)$, $(d_1,d_2,d_3,d_4) = (2,1,2,1)$

JOB SEQUENCING WITH DEADLINES (contd..)

Example : Let $n = 4$, $(p_1,p_2,p_3,p_4) = (100,10,15,27)$, $(d_1,d_2,d_3,d_4) = (2,1,2,1)$

Sr.No.	Feasible Solution	Processing Sequence	Profit value
(i)	(1,2)	(2,1)	110
(ii)	(1,3)	(1,3) or (3,1)	115
(iii)	(1,4)	(4,1)	127 is the optimal one
(iv)	(2,3)	(2,3)	25
(v)	(3,4)	(4,3)	42
(vi)	(1)	(1)	100
(vii)	(2)	(2)	10
(viii)	(3)	(3)	15
(ix)	(4)	(4)	27

38

Example Explanation

- Consider the jobs in the non increasing order of profits subject to the constraint that the resulting job sequence J is a feasible solution.
- In the example considered before, the non-increasing profit vector is

(100 27 15 10) (2 1 2 1)

p_1 p_4 p_3 p_2 d_1 d_4 d_3 d_2

39

GREEDY ALGORITHM TO OBTAIN AN OPTIMAL SOLUTION (Contd..)

(100 27 15 10)(2 1 2 1)

p_1 p_4 p_3 p_2 d_1 d_4 d_3 d_2

J = { 1} is a feasible one

J = { 1, 4} is a feasible one with processing sequence (4,1)

J = { 1, 3, 4} is not feasible

J = { 1, 2, 4} is not feasible

J = { 1, 4} is optimal

40

GREEDY ALGORITHM TO OBTAIN AN OPTIMAL SOLUTION (Contd..)

J(r+1)□i ; k □k+1

// i is inserted at position r+1 //

// and total jobs in J are increased by one //

repeat

end JS

41

Algorithm for Job sequencing with Deadlines

```
Algorithm JS(d, j, n)
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
// are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
// is the ith job in the optimal solution, 1 ≤ i ≤ k.
// Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
{
    d[0] := J[0] := 0; // Initialize.
    J[1] := 1; // Include job 1.
    k := 1;
    for i := 2 to n do
    {
        // Consider jobs in nonincreasing order of p[i]. Find
        // position for i and check feasibility of insertion.
        r := k;
        while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
        if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
        {
            // Insert i into J[ ].
            for q := k to (r + 1)  step −1 do J[q + 1] := J[q];
            J[r + 1] := i; k := k + 1;
        }
    }
    return k;
}
```

COMPLEXITY ANALYSIS OF JS ALGORITHM

- Let n be the number of jobs and s be the number of jobs included in the solution.
- The loop between lines 4-15 (the for-loop) is iterated (n-1)times.
- Each iteration takes O(k) where k is the number of existing jobs.

$\therefore$ The time needed by the algorithm is 0(sn) $s \leq n$ so the worst case time is $0(n^2)$.

If $d_i = n - i+1 \quad 1 \leq i \leq n$, JS takes $\theta(n^2)$ time

D and J need $\theta(s)$ amount of space.

43

Examples :Job Sequencing Problems:

1) Write a greedy algorithm for sequencing unit time jobs with deadlines and profits. Using this algorithm, find the optimal solutions when n=5,(p1,p2,p3,p4,p5)=(20,15,10,5,1) and (d1,d2,d3,d4,d5)=(2,2,1,3,3).

2) Find the correct sequence for jobs using following instances,

JobID	Deadline	Profit
1	4	20
2	1	10
3	1	40
4	1	30

3)Find the correct sequence for jobs using following instances,

JobID	Deadline	Profit
1	2	100
2	1	19
3	2	27
4	1	25
5	3	15

4)Find the optimum job sequence for the following problem.

n = 7 P_7 J_1 j_2 j_3 j_4 j_5 j_6 j_7

$(P_1, P_2, P_3, P_4, P_5, P_6)$ = (3, 5, 10, 18, 1, 6, 30)

, $d_4, d_5, d_6)$ = (1,3, 4, 3, 2, 1, 2)

Feasible soln.	**Processing Sequence**	**Value**
(1, 2, 3, 4)	(1,2,4,3) or (1,4,2,3)	3+5+10+18= 36

The End

45