Greedy Algorithms

Optimization Problems

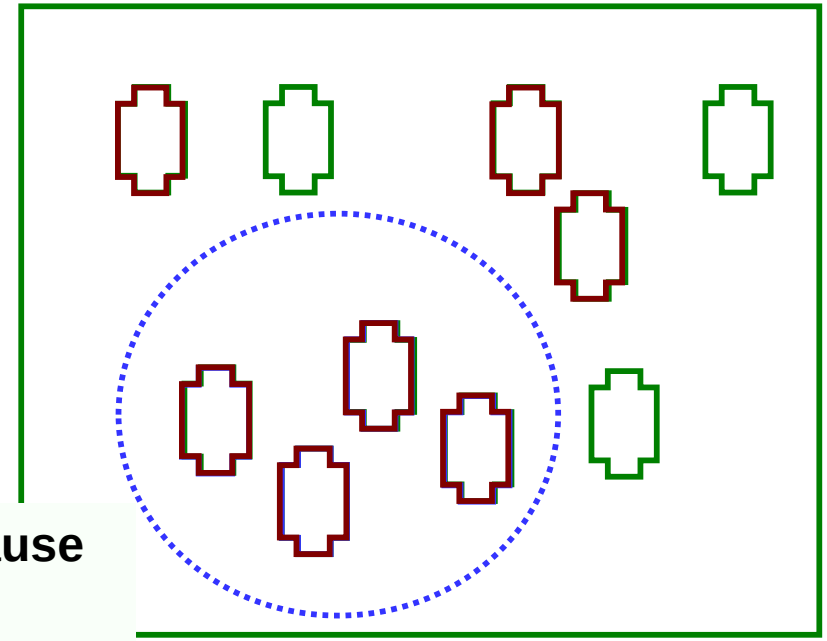
- **Optimization Problem**
  - Problem with an objective function to either:
    - Maximize some profit
    - Minimize some cost

- **Optimization problems appear in so many applications**
  - *Maximize* the number of jobs using a resource **[Activity-Selection Problem**]
  - Encode the data in a file to *minimize* its size [**Huffman Encoding Problem**]
  - Collect the *maximum* value of goods that fit in a given bucket [**knapsack Problem**]
  - Select the *smallest-weight* of edges to connect all nodes in a graph **[Minimum Spanning Tree**]

Solving Optimization Problems

- **Two techniques for solving optimization problems:**
  - Greedy Algorithms ("Greedy Strategy")
  - Dynamic Programming

Space of optimization problems



Greedy algorithms can solve some problems optimally

Dynamic programming can solve more problems optimally (superset)

We still care about Greedy Algorithms because for some problems:

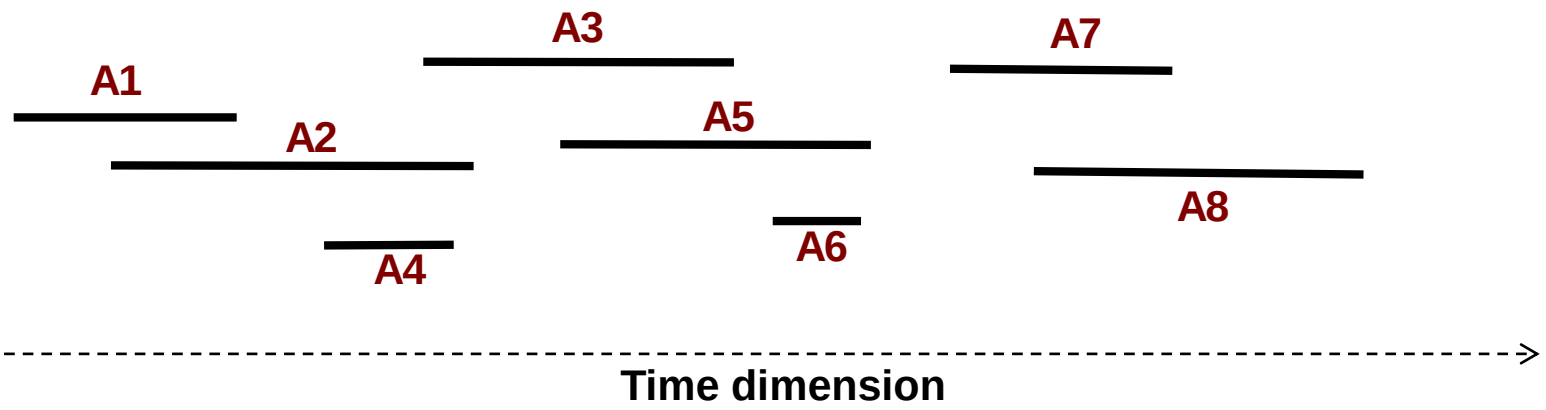
- **Dynamic programming is overkill (slow)**
- **Greedy algorithm is simpler and more efficient**

Greedy Algorithms

- **Main Concept**
  - **Divide the problem into multiple steps (sub-problems)**
  - **For each step take the best choice at the current moment (Local optimal) (Greedy choice)**
  - **A *greedy algorithm* always makes the choice that looks best at the moment**
  - **The hope: A locally optimal choice will lead to a globally optimal solution**
    - For some problems, it works. For others, it does not

Activity-Selection Problem

- **Given a set of activities A_1, A_2, …A_n (E.g., talks or lectures)**
- **Each activity has a start time and end time**
  - **Each A_i has (S_i, E_i)**
- **Activities should use a common resource (E.g., Lecture hall)**
- **Objective: Maximize the number of "compatible" activities that use the resource**
  - **Cannot have two overlapping activities**



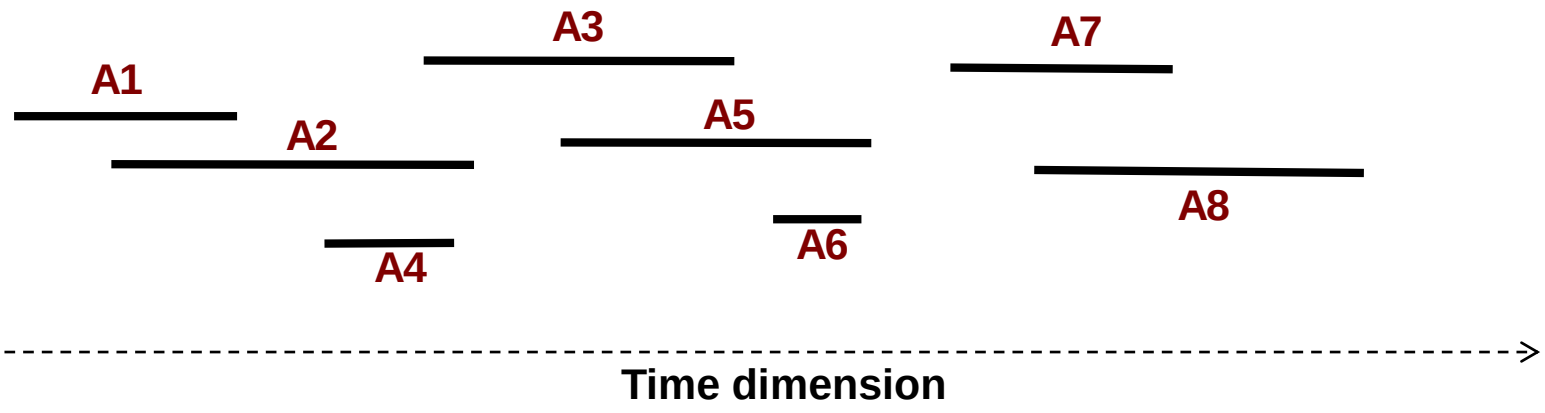
Another Version Of The Same Problem

- **A set of events A_1, A_2, …A_n**
- **Each event has a start time and end time**
- **You want to attend as many events as possible**

Example of Compatible Activities

- Set of activities $A = \{A_1, A_2, \ldots, A_n\}$
- Each $A_i = (S_i, E_i)$

Examples of compatible activities:
{A1, A3, A8}
{A1, A4, A5, A7}
{A2, A5, A8}
….



Greedy Algorithm

- **Select the activity that ends first (smallest end time)**
  - **Intuition: it leaves the largest possible empty space for more activities**
- **Once selected an activity**
  - **Delete all non-compatible activities**
  - **They cannot be selected**
- **Repeat the algorithm for the remaining activities**
  - **Either using iterations or recursion**

Greedy Algorithm

- **Select the activity that ends first (smallest end time)**
  - **Intuition: it leaves the largest possible empty space for more activities**
- **Once selected an activity**
  - **Delete all non-compatible activities**
  - **They cannot be selected**
- **Repeat the algorithm for the remaining activities**
  - **Either using iterations or recursion**

Greedy Choice: Select the next best activity (Local Optimal)

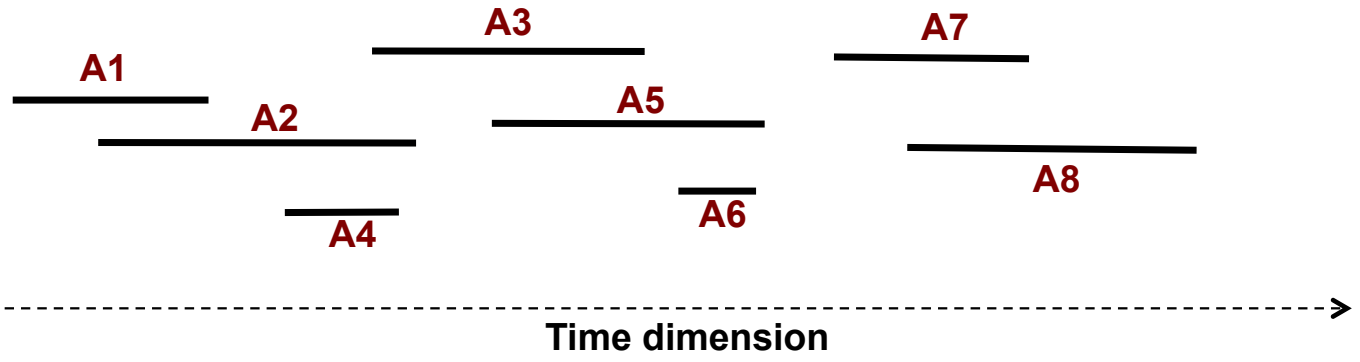
Sub-problem: We created one sub-problem to solve (Find the optimal schedule after the selected activity)

Hopefully when we merge the local optimal + the sub-problem optimal solution ➔ we get a global optimal

DYNAMIC PROGRAMMING VS GREEDY

Dynamic Programming	**Greedy Algorithm**
At each step, the choice is determined based on solutions of sub problems.	At each step, we quickly make a choice that currently looks best. A local optimal (greedy) choice
Sub-problems are solved first.	Greedy choice can be made first before solving further sub- problems.
Bottom-up approach	Top-down approach
Can be slower, more complex	Usually faster, simpler

Example



A: i	1	2	3	4	5	6	7	8
S_i	1	2	7	5	10	16	21	23
E_i	4	9	15	8	18	17	27	30

The greedy algorithm will select:
{A1, A4, A6, A7}

Is that an optimal answer?? Can we find a larger set?

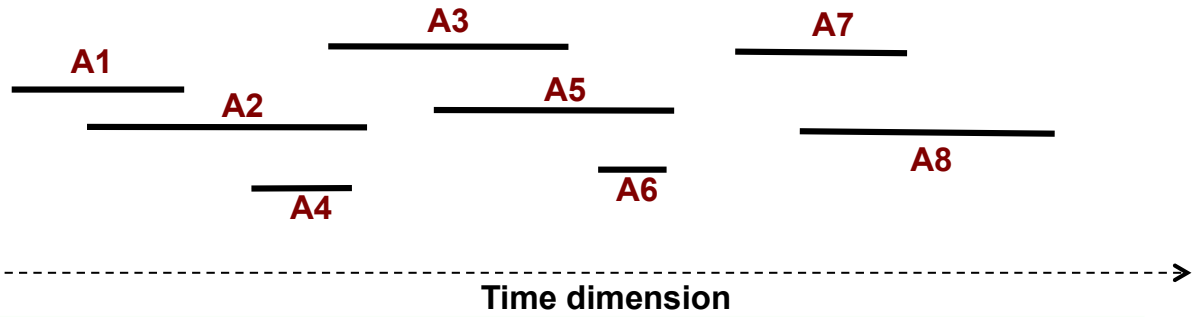
Optimal Solution

- **The Greedy Algorithm leads to optimal solution**

- **How to prove it**
  - **We can convert any other optimal solution (S') to the greedy algorithm solution (S)**

- **Idea:**
  - **Compare the activities in S' and S from left-to-right**
  - **If they match in the selected activity ➔ skip**
  - **If they do not match**
    - **We can replace the activity in S' by that in S because the one in S finishes first**

Example



A:

i	1	2	3	4	5	6	7	8
S_i	1	2	7	5	10	16	21	23
E_i	4	9	15	8	18	17	27	30

- Greedy solution S = {A1, A4, A6, A7}
- Another solution S' = {A1, A4, A5, A8}
- A5 in S' can be replaced by A6 from S (finishes earlier)
- A8 in S' can be replaced by A7 from S (finishes earlier)

By these replacements, we saved time.
So, Greedy must be at least as good as any other optimal solution

Recursive Solution

Two arrays containing the start and end times (Assumption: they are sorted based on end times)

The activity chosen in the last call

The problem size

```
Recursive-Activity-Selection(S, E, k, n)
    m = k +1
    While (m <= n) && ( S[m] < E[k])
            m++;
    If (m <= n)
        return  {A_m}  U Recursive-Activity-Selection(S, E, m, n)
    Else
        return Φ
```

Find the next activity starting after the end of k

Time Complexity: O(n)

(Assuming arrays are already sorted, otherwise we add O(n Log n)

Iterative Solution

Two arrays containing the start and end times (Assumption: they are sorted based on end times)

```
Iterative-Activity-Selection(S, E)
    n = S.Length
    List = {A1}
    lastSelection = 1

    For (i = 2 to n)
        if (S[i] >= E[lastSelection])
            List = List U {Ai}
            lastSelection = i
        End If
    End Loop

    Return List
```

Elements Of Greedy Algorithms

- **Greedy-Choice Property**
  - At each step, we do a greedy (local optimal) choice
- **Top-Down Solution**
  - The greedy choice is usually done independent of the sub-problems
  - Usually done “before” solving the sub-problem
- **Optimal Substructure**
  - The global optimal solution can be composed from the local optimal of the sub-problems

Elements Of Greedy Algorithms

- **Proving a greedy solution is optimal**
  - Remember: Not all problems have optimal greedy solution
  - If it does, you need to prove it
  - Usually the proof includes mapping or converting any other optimal solution to the greedy solution

We mapped S' to S and showed that S is even better

- **Greedy solution S = {A1, A4, A6, A7}**
- **Another solution S' = {A1, A4, A5, A8}**
- **A5 in S' can be replaced by A6 from S (finishes earlier)**
- **A8 in S' can be replaced by A7 from S (finishes earlier)**

Activity-Selection Problem

KNAPSACK PROBLEM

There are two version of knapsack problem:

1. **0-1 knapsack problem:**

   - — Items are indivisible. (either take an item or not)
   - — can be solved with dynamic programming.

2. **Fractional knapsack problem:**

   - — Items are divisible. (can take any fraction of an item)
   - — It can be solved in greedy method

Knapsack Problem: Definition

- Thief has a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value v_i (all w_i , v_i and W are integer values)
- **Problem: How to pack the knapsack to achieve maximum total value of packed items?**

0-1 Knapsack

- **Items cannot be divided**
  - Either take it or leave it

❑ find x_i such that for all $x_i = \{0, 1\}$,

$i = 1, 2, .., n$

$\sum w_i x_i \leq W$ and

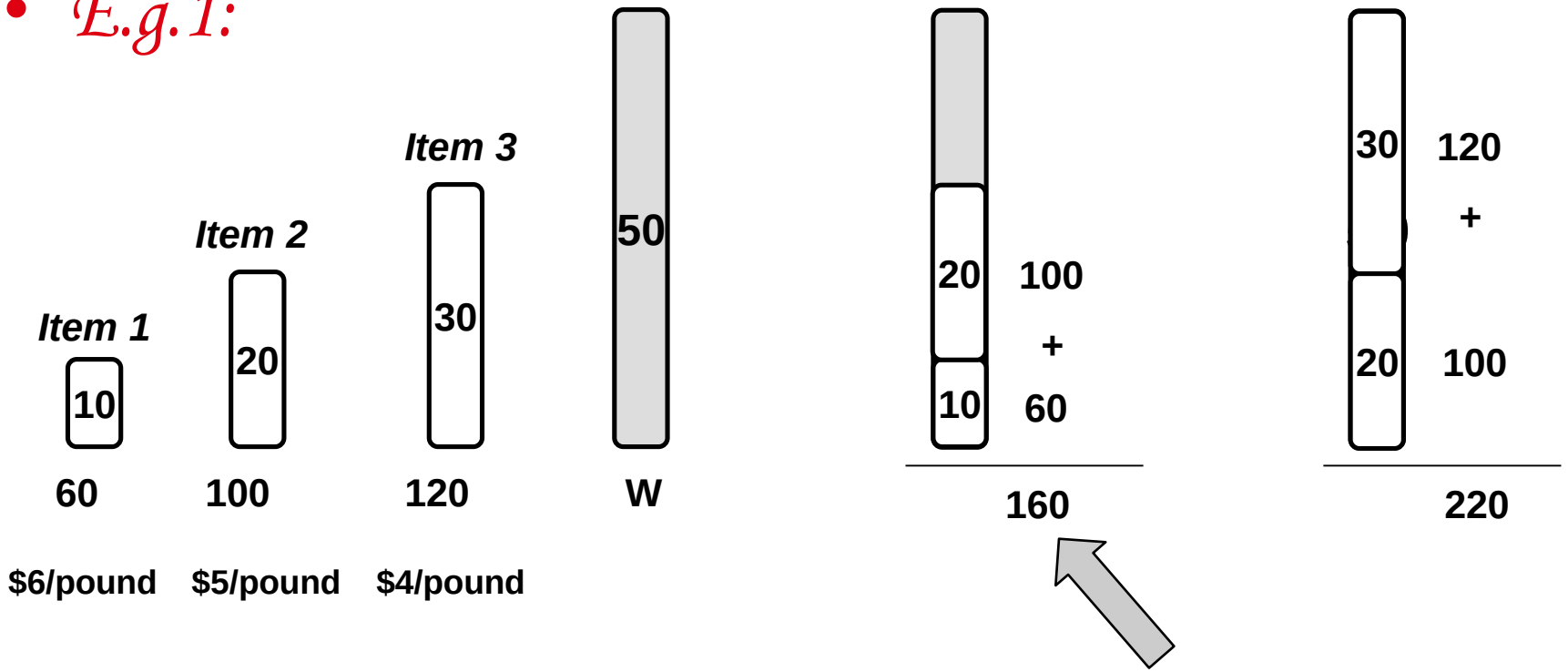
$\sum x_i v_i$ is maximum

If $X_i = 1$, then item i will be taken

If $X_i = 0$, then item i will be skipped

0-1 Knapsack - Greedy Strategy Does Not Work

- *E.g.1:*



- **Greedy choice:**
  - **Compute the benefit per pound**
  - **Sort the items based on these values**

Fractional Knapsack

- **Items can be divided**
  - Can take part of it as needed

❑ find x_i such that for all $0 <= x_i <= 1$,

$i = 1, 2, .., n$

$\sum w_i x_i \leq W$ and

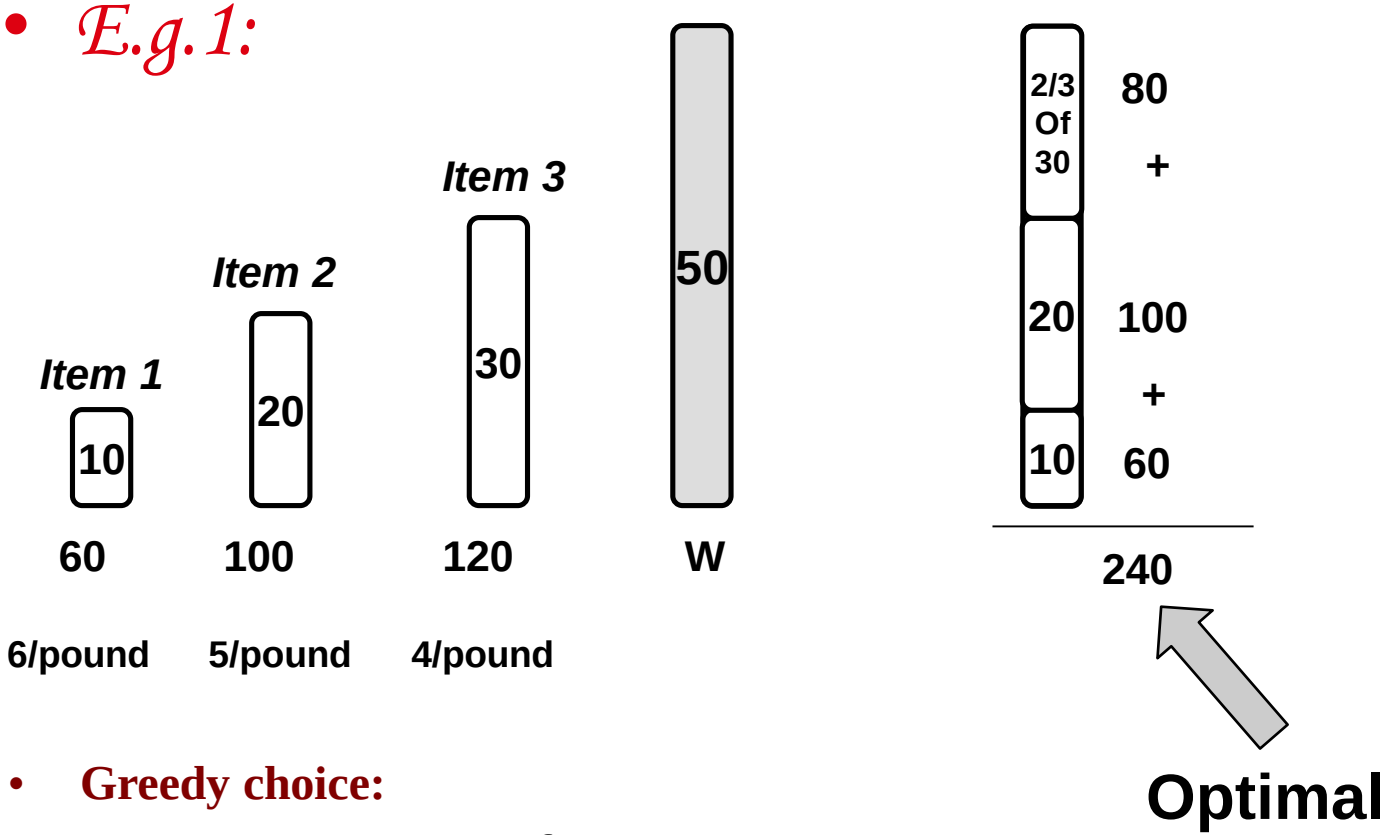
$\sum x_i v_i$ is maximum

If $X_i = 0$, then item i will be skipped

If $X_i > 0$, then X_i fraction of item i will be taken

Fractional Knapsack - Greedy Strategy Works

- *E.g.1:*



- **Greedy choice:**
  - **Compute the benefit per pound**
  - **Sort the items based on these values**
  - **Take as much as you can from the top items in the list**

THE OPTIMAL KNAPSACK ALGORITHM

- **Input:**
  - an integer n
  - positive values wi and vi such that $1 \leq i \leq n$
  - positive value W.
- **Output:**
  - n values of xi such that $0 \leq xi \leq 1$
  - Total profit

THE OPTIMAL KNAPSACK ALGORITHM

Initialization:

- Sort the n objects from large to small based on their ratios vi / wi .

- We assume the arrays $w[1..n]$ and $v[1..n]$ store the respective weights and values after sorting.

- initialize array $x[1..n]$ to zeros.

- weight = 0; i = 1;

THE OPTIMAL KNAPSACK ALGORITHM

while ($i \leq n$ and weight < W) do

  if weight + $w[i] \leq W$ then

  $x[i] = 1$

  else

  $x[i] = (W - \text{weight}) / \text{w}[i]$

  weight = weight + x[i] * w[i]

  i++

KNAPSACK - EXAMPLE

Problem:

$n = 3$

$W = 20$

$(v_1, v_2, v_3) = (25, 24, 15)$ $(w_1, w_2, w_3) = (18, 15, 10)$

KNAPSACK - EXAMPLE

Solution:

- Optimal solution:
  - $x_1 = 0$
  - $x_2 = 1$
  - $x_3 = 1/2$
- Total profit = 24 + 7.5 = 31.5

Examples

- Many algorithms can be viewed as applications of the Greedy algorithms, such as (includes but is not limited to):

1. Minimum Spanning Tree
2. Dijkstra's algorithm for shortest paths from a single source
3. Huffman codes (data-compression codes)

MINIMUM SPANNING TREE

- Let G = (N, A) be a connected, undirected graph where N is the set of nodes and A is the set of edges. Each edge has a given nonnegative length. The problem is to find a subset T of the edges of G such that all the nodes remain connected when only the edges in T are used, and the sum of the lengths of the edges in T is as small as possible possible.
- Since G is connected, atleast one solution must exist.

Finding Spanning Trees

There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms

- Kruskal's algorithm:

Created in 1957 by Joseph Kruskal

- Prim's algorithm

Created by Robert C. Prim

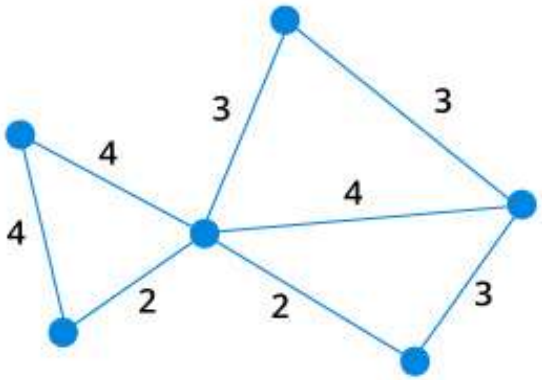
Kruskal's Algorithm

- The steps for implementing **Kruskal's algorithm** are as follows:

- Sort all the edges from low weight to high
- Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
- Keep adding edges until we reach all vertices.

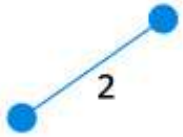
1

Start with a weighted graph



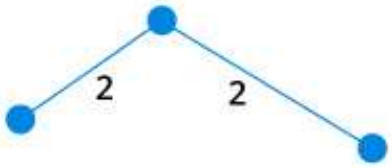
2

Choose the edge with least weight,
if there are more than 1,
choose any one.



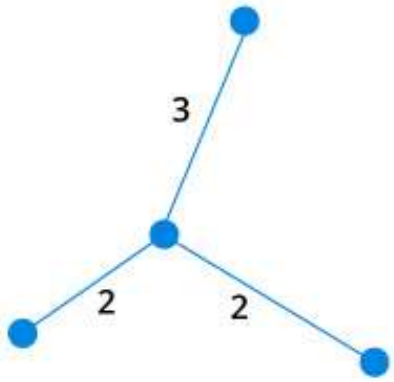
3

Choose the next shortest edge
and add it



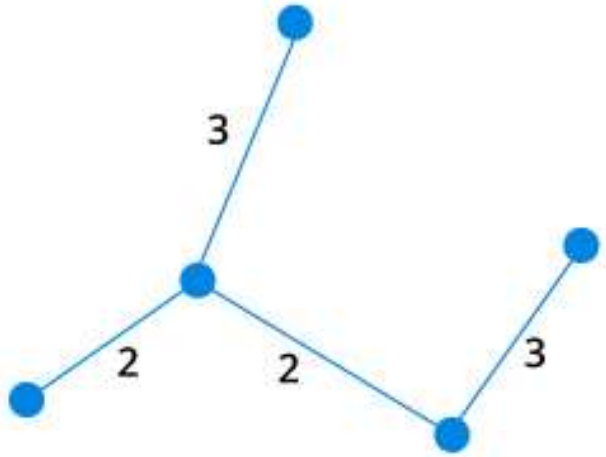
4

Choose the next shortest edge
that doesn't create a cycle and add it



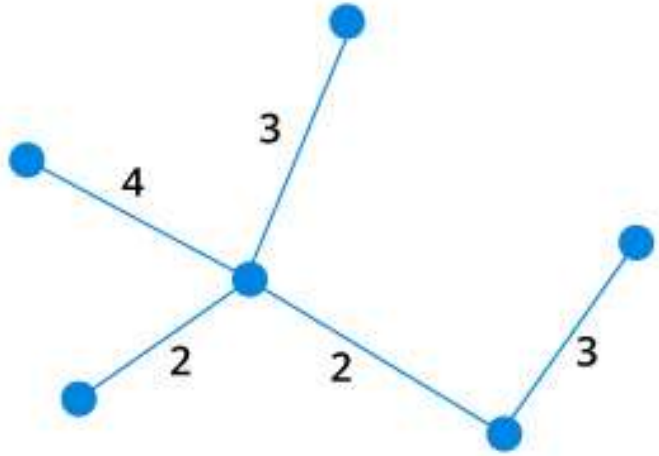
5

Choose the next shortest edge that doesn't create a cycle and add it



6

Repeat until you have a spanning tree



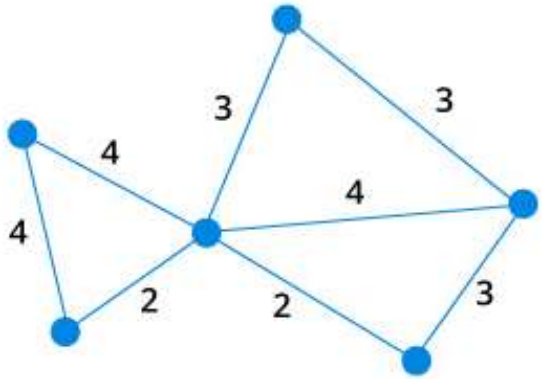
Prim's Algorithm

- The steps for implementing **Prim's algorithm** are as follows:

- Initialize the minimum spanning tree with a vertex chosen at random.
- Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
- Keep repeating step 2 until we get a minimum spanning tree

1

Start with a weighted graph

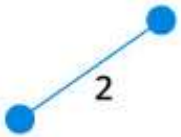


2

Choose a vertex

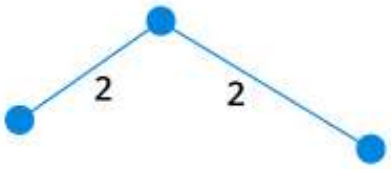
3

Choose the shortest edge from this vertex and add it



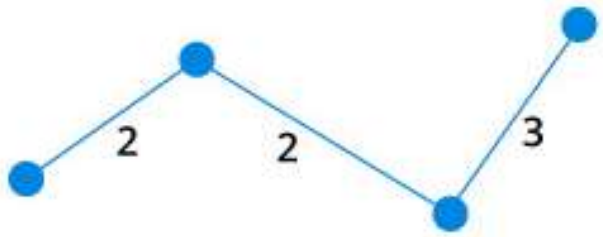
4

Choose the nearest vertex not yet in the solution



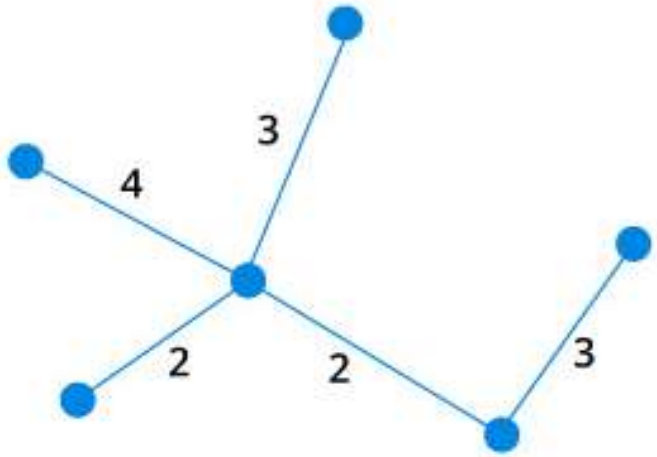
5

Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



6

Repeat until you have a spanning tree



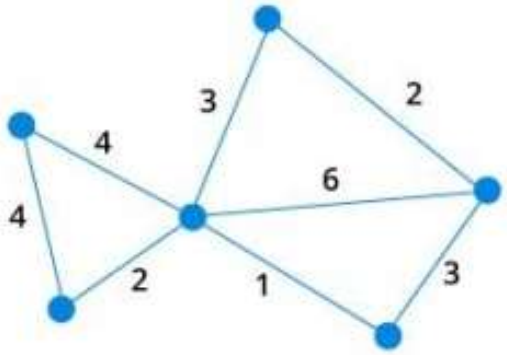
Djikstra's Algorithm Pseudocode

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

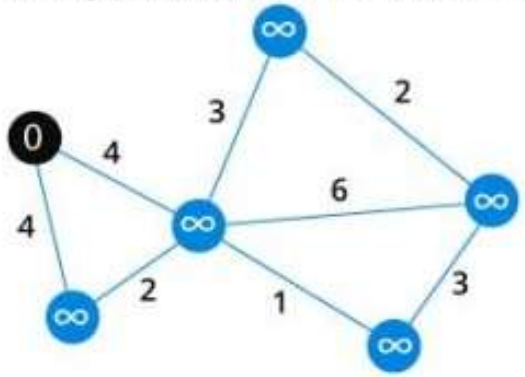
1

Start with a weighted graph



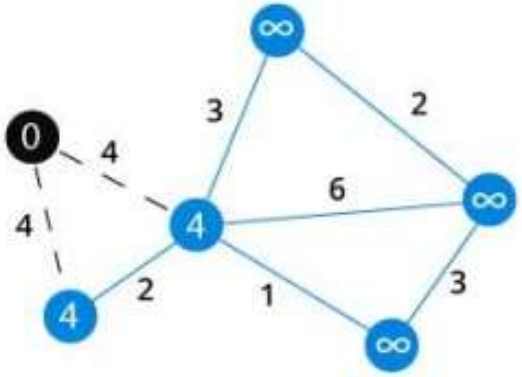
2

Choose a starting vertex and assign infinity path values to all other vertices



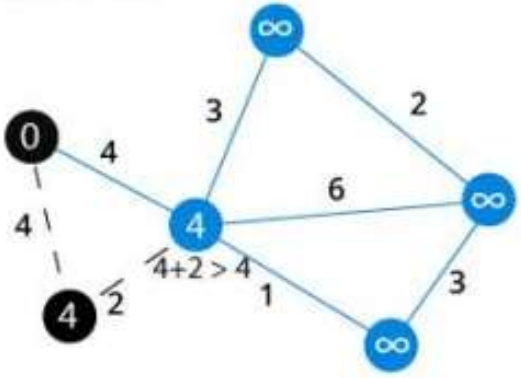
3

Go to each vertex adjacent to this vertex and update its path length



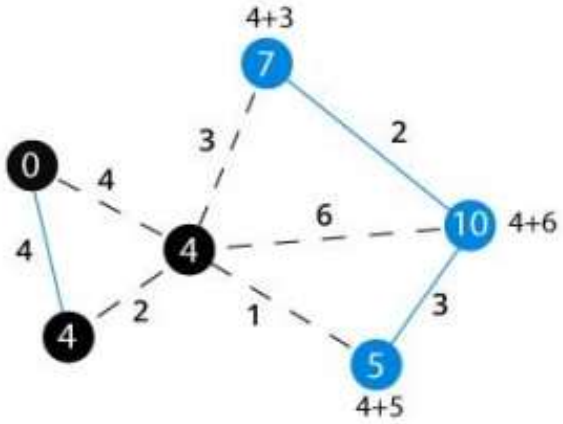
4

If the path length of adjacent vertex is lesser than new path length, don't update it.



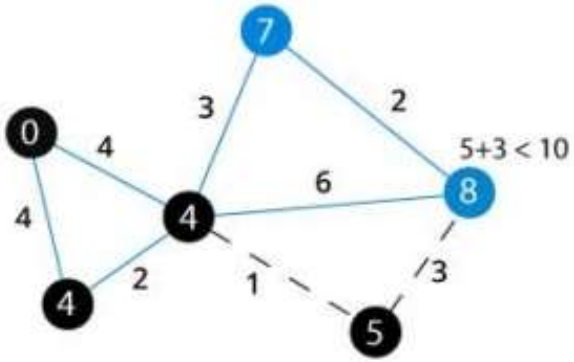
5

Avoid updating path lengths of already visited vertices



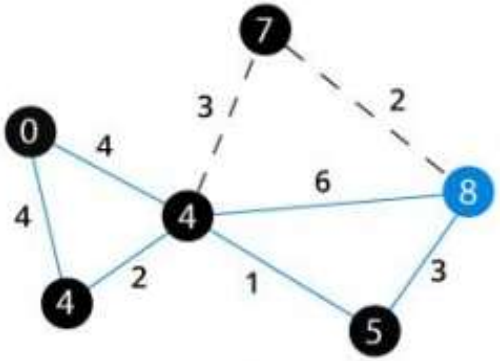
6

After each iteration, we pick the unvisited vertex with least path length. So we chose 5 before 7



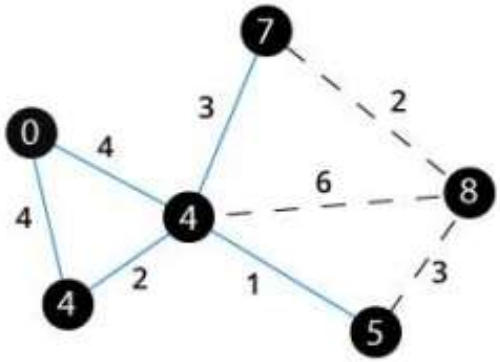
7

Notice how the rightmost vertex has its path length updated twice



8

Repeat until all the vertices have been visited



Huffman Codes

Huffman codes are an effective technique of ‘lossless data compression’ which means no information is lost.

- The algorithm builds a table of the frequencies of each character in a file
  - The table is then used to determine an optimal way of representing each character as a binary string

Constructing a Huffman Code

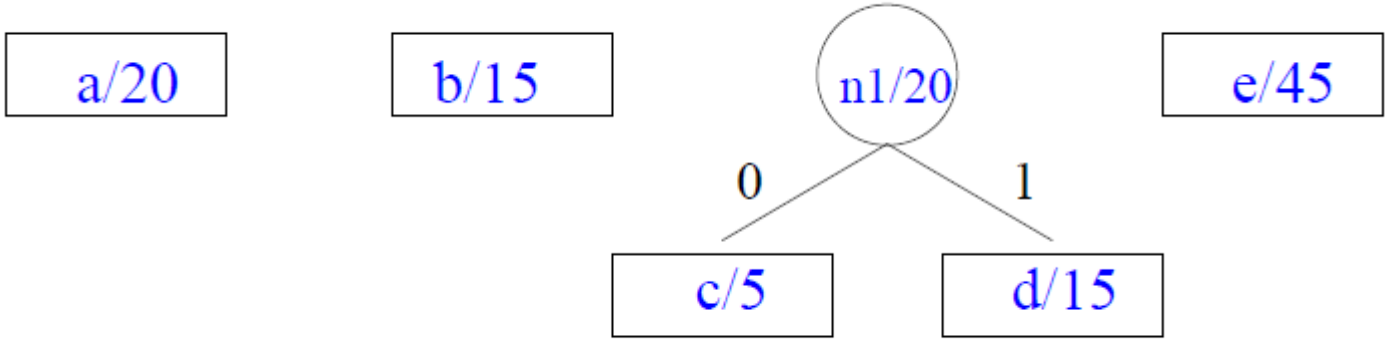
Huffman developed a greedy algorithm for constructing an optimal prefix code

- The algorithm builds the tree in a bottom-up manner
- It begins with the leaves, then performs merging operations to build up the tree
- At each step, it merges the two least frequent members together
- It removes these characters from the set, and replaces them with a “metacharacter” with frequency = sum of the removed characters’ frequencies

Example

Let A = {a / 20, b/ 15, c/ 5, d/ 15, e/ 45} be the alphabet and its frequency distribution.

In the first step Huffman coding merges c and d .

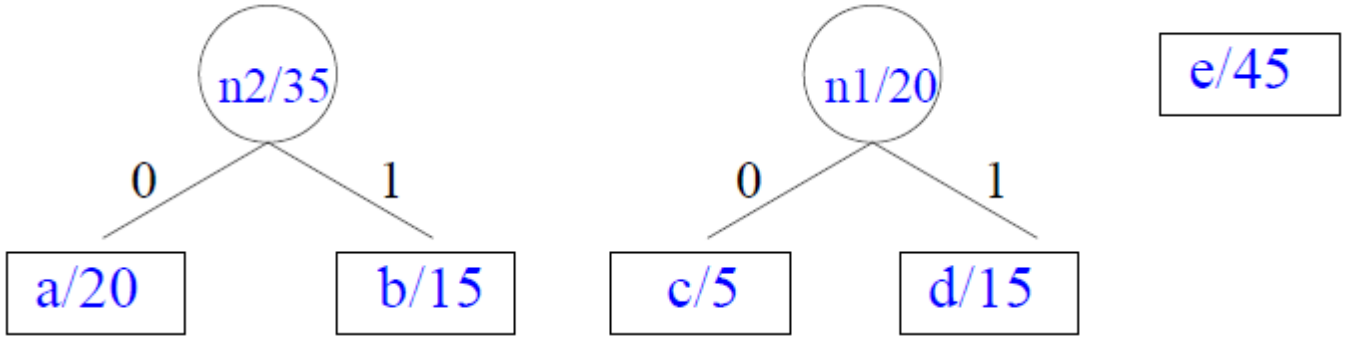


Alphabet is now A 1 = {a / 20 , b/ 15, n1/ 20, e/45}.

Alphabet is now A1 = {a / 20 , b/ 15, n1/ 20 , e/ 45}.

Algorithm merges a and b

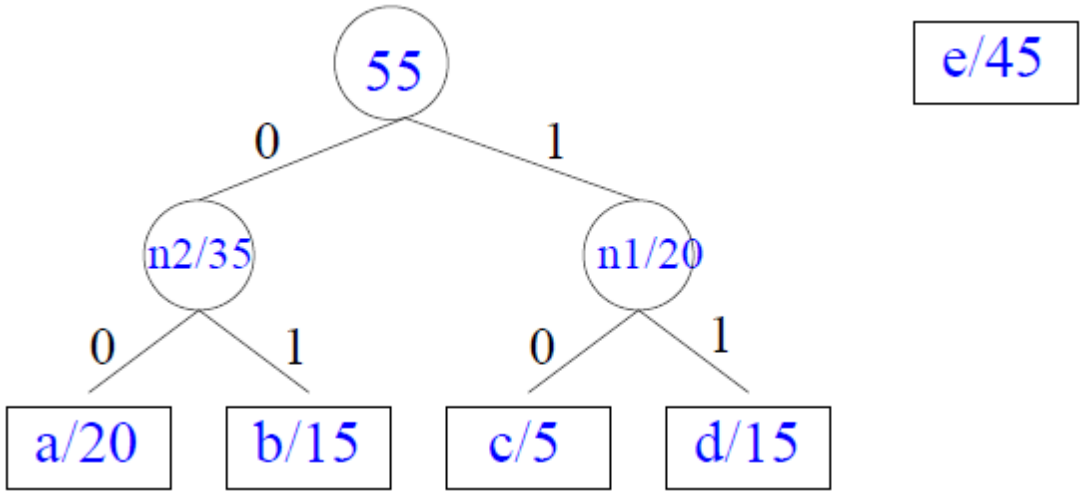
(could also have merged n1 and b).



New alphabet is A2 = {n2/ 35, n1/ 20, e/ 45}.

Alphabet is A2 = {n2/35 , n1/ 20, e/45}.

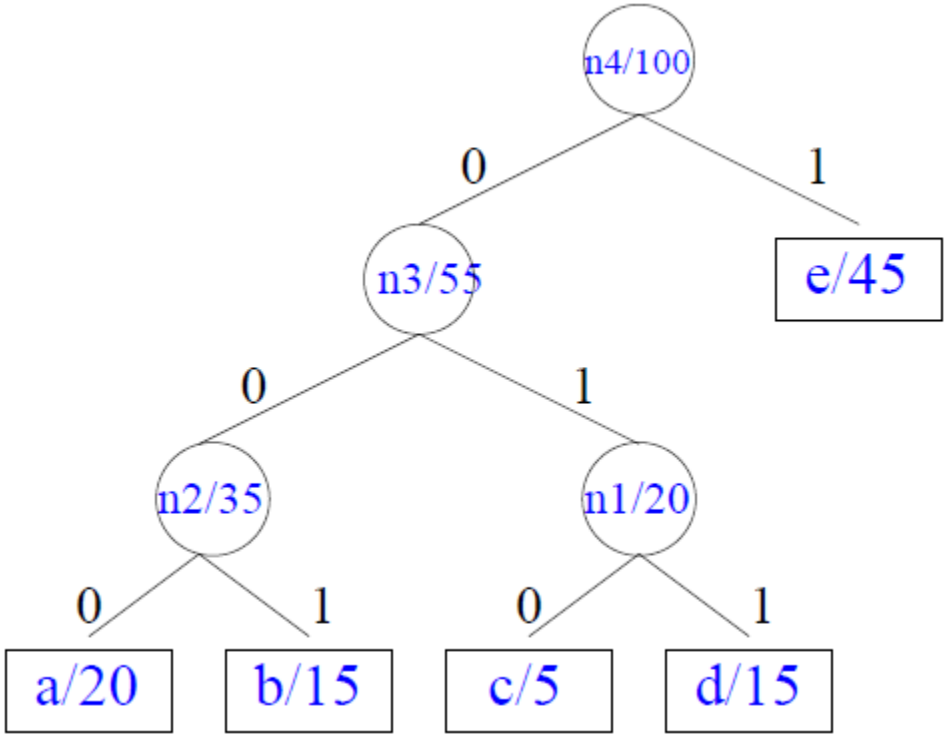
Algorithm merges n1 and n2 .



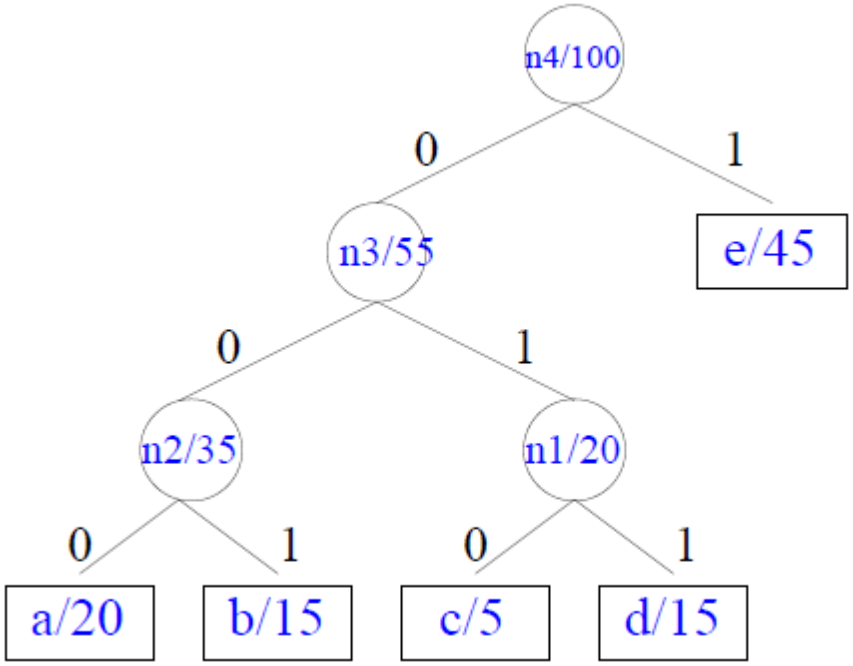
New alphabet is A3 = {n3/ 55, e/ 45}.

Current alphabet is A3= {n3/ 55, e/45}.

Algorithm merges e and n3 and finishes.



Huffman code is obtained from the Huffman tree



Huffman code is
a = 000, b = 001, c = 010, d = 011, e = 1.
This is the optimum (minimum-cost) prefix code for this distribution.