Algorithms – I (CS29003/203)

Autumn 2022, IIT Kharagpur

Greedy Algorithms

Greedy Algorithms

- Greedy algorithm repeatedly makes locally best choice/decision ignoring what its effect will be in future
- They are often intuitive, easy to understand and easy to implement
- However the problem is that in many situations we can not solve a problem using a greedy approach
- Greedy solution is not necessarily best
- Sometimes greedy may also be good enough
    - When you can prove it

Optimization Problems

- A class of problems in which we are asked to
  - Find a set (or a sequence) of "items"
  - That satisfy some constraints and simultaneously optimize (i.e., maximize or minimize) some objective function
- Formally

$$\min/\max f(\boldsymbol{x})$$
$$s.t.\ \boldsymbol{g}(\boldsymbol{x}) \gtreqless 0$$

- A sequence of tasks with deadline, maximize reward while finishing before deadline
  - Items: tasks, constraints: finish before deadline, optimize: total reward
- A set of products with weights and values, put into a bag of weight limit x and maximize value
  - *Items*: products, *constraints*: weight limit, *optimize*: total value
- A file in computer, encode/compress it to minimize the length
  - *Items*: codewords for each character, *constraints*: original file recoverable, *optimize*: code length

Knapsack Problem

- Given n items of known weights $w_1, w_2, \dots, w_n$ and values $v_1, v_2, \dots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack
- Two variations of Knapsack problem is there
  - (Fractional Knapsack): You are allowed to take fractions of items
  - (0-1 Knapsack): You have to take an item either whole or none
- Objective: Total value in the knapsack
- Constraints: Sum of weights of items in Knapsack can't exceed the knapsack capacity. (In 0-1 version) Only whole item or none

Image source: www.indiamart.com

Knapsack Problem

- Which variation of the Knapsack problem should be easy to solve?
- Any simple approach to solve the fractional Knapsack problem?
- Get the per unit value of the items and fill out the knapsack starting with the item highest per unit value and go on in a descending order
- The total value is coming to be Rs. 240
- The strategy here is literally "greedy"
- It is also optimal in this case

Total value = 60+100+4*20 = Rs. 240

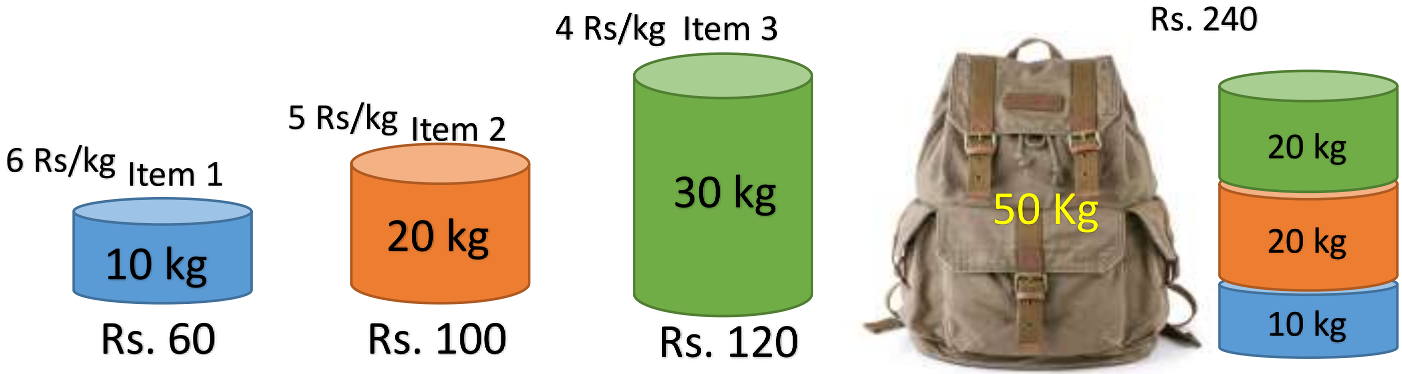


Image source: www.indiamart.com

Knapsack Problem

- What may come challenging sometimes is proving a greedy technique indeed gives optimal solution
- For fractional knapsack its relatively easy
- Intuitively: For the first 10 kg space of the knapsack we are putting the best possible option (item 1).
- For the next 20 kg space we going for the best possible option available and so on
- Formal proof will come later

Total value =
60+100+4*20 =
Rs. 240

Image source: www.indiamart.com

Knapsack Problem

- Lets see how this strategy works for 0-1 Knapsack problem
- But this is not the optimal solution. We can do better
- What is the optimal solution?

Image source: www.indiamart.com

Knapsack Problem

- Lets see how this strategy works for 0-1 Knapsack problem
- But this is not the optimal solution. We can do better
- What is the optimal solution?
- Full of item 3 and item 2
- It was easy for this example as the number of items are only 3
- In general, for n items, you need to check 2^n combinations

Total value = 120+100 = Rs. 220

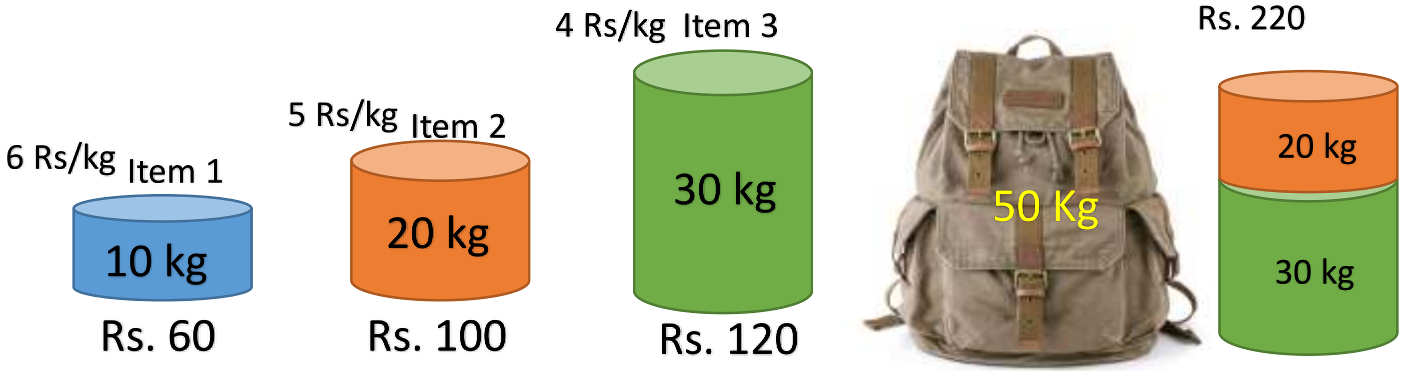


Image source: www.indiamart.com

Knapsack Problem

- The greedy strategy that worked for the fractional Knapsack problem, did not work for the 0-1 Knapsack problem
- Lets try another greedy strategy
- Starting with the item highest **total** value and go on in a descending order
- For this instance, it gives the optimal solution
- Any instance/example where this strategy fails?

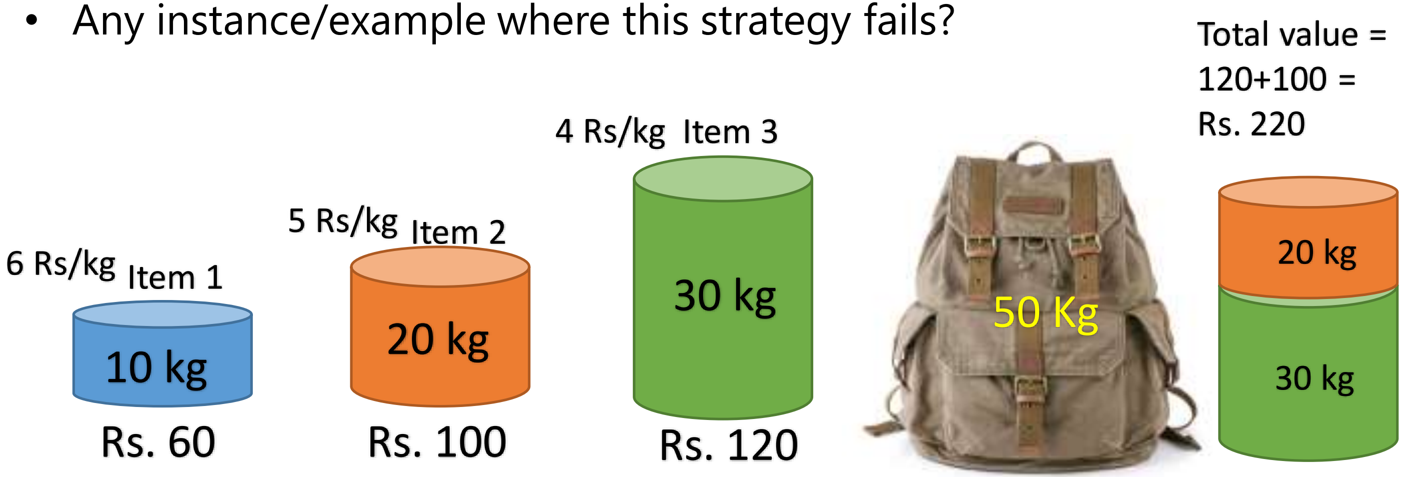


Image source: www.indiamart.com

Knapsack Problem

- Add a 4th item with more weight but with slightly high value
- Say a 4th item weighs 40 kg with value Rs. 130
- The greedy strategy gives the solution as 40 kg of item 4 and 10 kg of item 1. The total value is Rs. 190
- However, the optimal solution is 30 kg of item 1 and 20 kg of item 2. The total value is Rs. 220

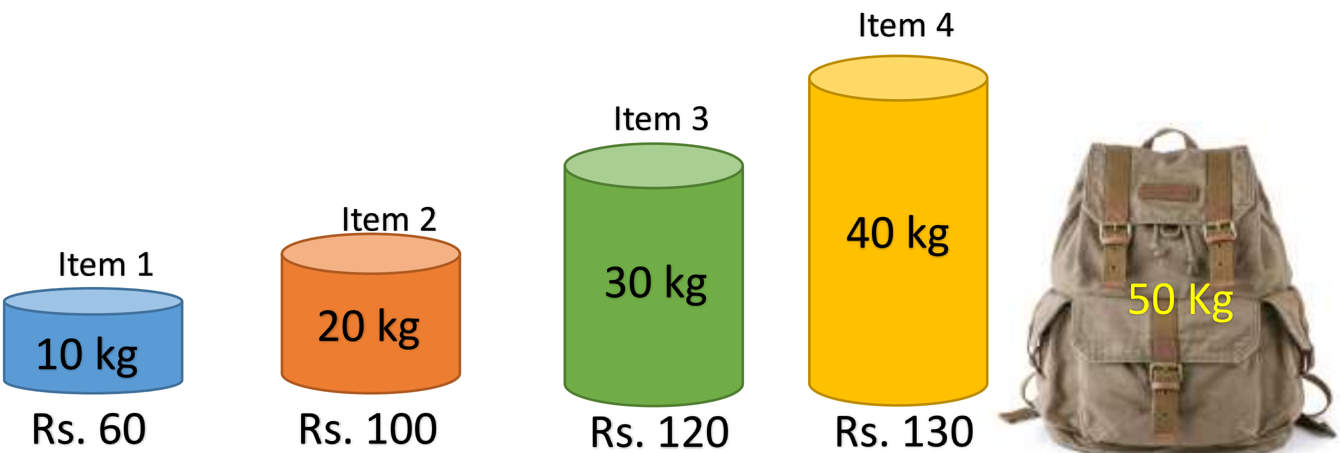


Image source: www.indiamart.com

Knapsack Problem

- We have tried two different greedy strategies to solve 0-1 Knapsack problem. But none of them worked
- Do we have any greedy strategy that works for 0-1 Knapsack problem?
- To the best of our knowledge there is no known greedy strategy that works for 0-1 Knapsack problem
- Later we will see other approaches to solve a 0-1 Knapsack problem

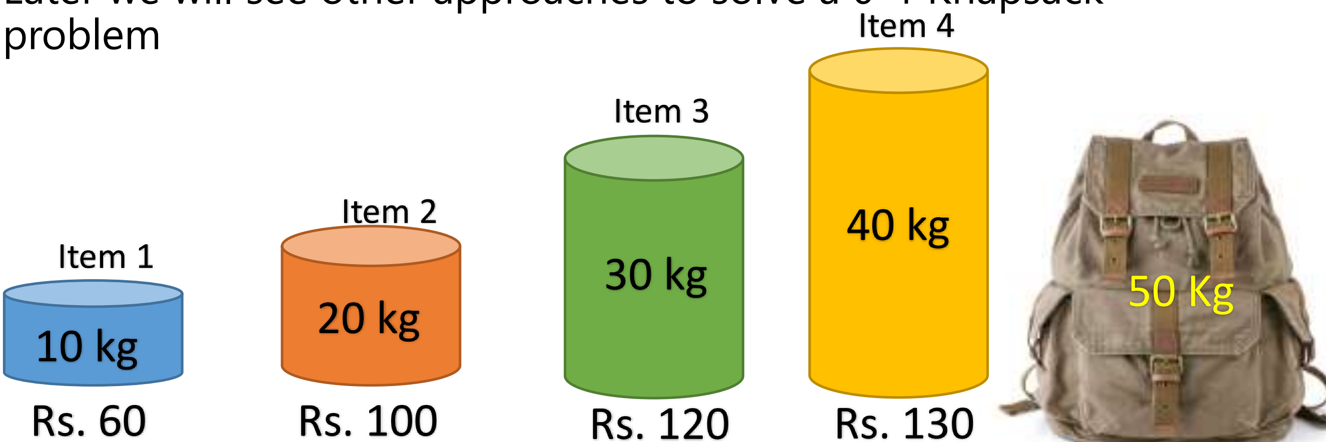


Image source: www.indiamart.com

Fractional Knapsack Problem - Pseudocode

```
FractionalKnapsack(w, V, C, n)
  Find V[i]/w[i] for all item i
  Sort the items in both V[i] and w[i] by V[i]/w[i] in
  descending order
  load = 0
  for i=1 to n
    if w[i] < C - load
      Take whole of item i
      load += w[i]
    else
      Take (C-load) amount of item i
      Load = C
      break
```

Fractional Knapsack Problem - Analysis

```
FractionalKnapsack(w, V, C, n)
  Find V[i]/w[i] for all item i
  Sort the items in both V[i] and w[i] by V[i]/w[i] in
  descending order
  load = 0
  for i=1 to n
    if w[i] ≤ C - load
      Take whole of item i
      load += w[i]
    else
      Take (C-load) amount of item i
      Load = C
      break
```

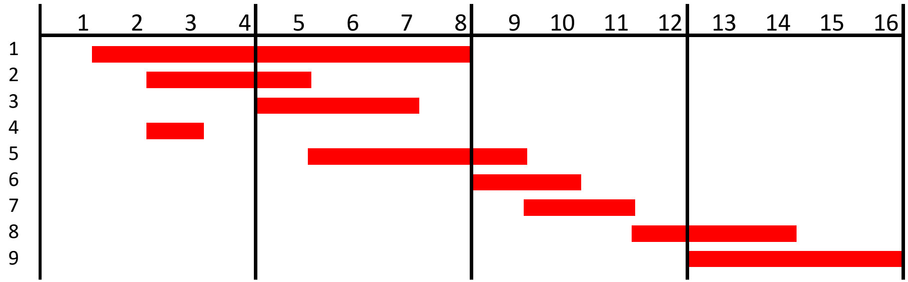
- Find V[i]/w[i] for all item i $\longrightarrow \Theta(n)$
- Sort ... descending order $\longrightarrow \Theta(n \log n)$
- load = 0 $\longrightarrow \Theta(1)$
- for loop $\longrightarrow \Theta(n)$

Overall runtime
$\Theta(n \log n)$

Activity Selection/Interval Scheduling Problem

- Imagine that you are trying to schedule as many classes as possible without any conflicting lectures.
- Given a collection C of intervals, find a subset $S \subseteq C$ so that
  - No two intervals in S overlap
  - $|S|$ is as large as possible

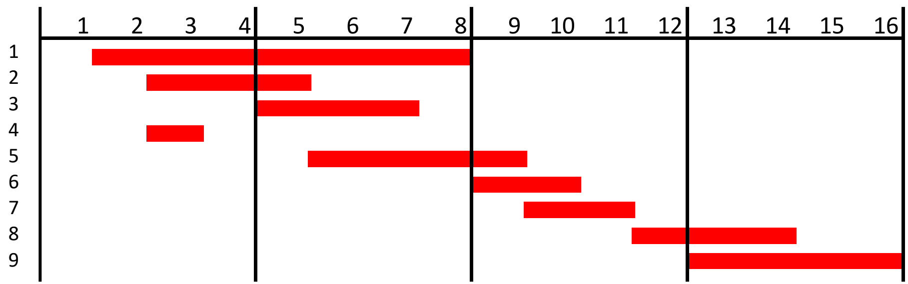
i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	2	5	8	9	11	13
f_i	8	5	7	3	9	10	11	14	16



Activity Selection/Interval Scheduling Problem

- What would be a brute force solution?
- Try all combinations, i.e., find the set of all subsets and check if the elements of the subset are compatible
- Complexity is $\Theta(2^n)$

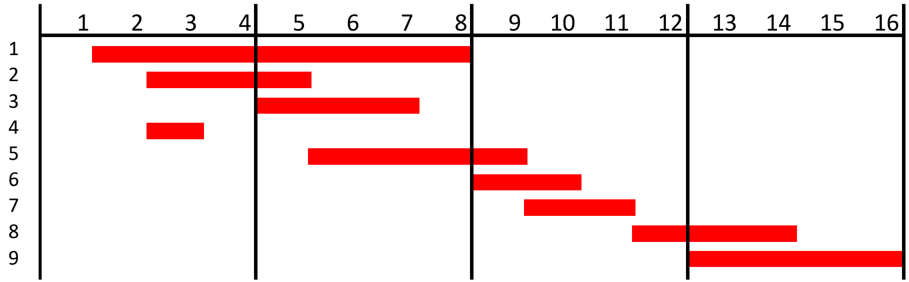
i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	2	5	8	9	11	13
f_i	8	5	7	3	9	10	11	14	16



Activity Selection/Interval Scheduling Problem

- What can be a greedy strategy?
- Remember greedy implies choosing a criterion and according to the criterion, take a decision that seems best at the moment and repeat this
- What about picking the shortest duration first, then the next shortest duration and so on?

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



Shortest Duration

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

✗ ✗ ✓ ✓ ✗ ✓ ✗ ✓ ✗

- So, our greedy solution is the activity subset $\{a_3, a_4, a_6, a_8\}$
- Is it optimal?
- In this case – yes
- We can show that for this problem, we can at max choose 4 actions
- We are deferring the formal proof for later
- However, this greedy strategy may not be optimal for all instances

Earliest Start

- What about choosing by early start?
- Is it optimal?
- No – in general

Earliest Finish

- What about choosing by early finish?
- Is it optimal?
- Yes

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

- So, our greedy solution is the activity subset $\{a_3, a_4, a_6, a_8\}$
- Note: optimal solution is not unique. Another candidate $\{a_2, a_5, a_7, a_9\}$

Alternate Strategy

- Given that 'earliest finish' strategy works, using symmetry what can be an alternate strategy that will also work?
- Hint: Think why 'earliest finish' strategy works

- Choosing earliest finish leaves maximum room for other activities to fill in
- So, by symmetry, 'latest start' [looking from the other end] will also give optimal solution

Tutorial Problems

- Given an infinite array in which the first n elements are integers in sorted order and the rest of the cells are filled with some special symbol (say $). Assume we do not know the n value. Give an algorithm that takes an integer k as input and finds a position in the array containing k, if the integer k exists in the array in $O(\log n)$ time

Tutorial Problems

- Given a sorted array of non-repeated integers A[1..n], check whether there is an index i for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $O(\log n)$

Tutorial Problems

- We are given two sorted arrays of size n. Give an algorithm for finding the median element in the union of the two lists so that the complexity is $\Theta(n)$

Activity Selection Problem - Pseudocode

```
s: Array of start times, f: Array of finish times
R: Set of all requests, A: Set of accepted requests
n: Number of activities, k: Index of last accepted activity
ActivitySel(s, f, A, n)
   Sort the items in s, f and A by f[i] in ascending order
   Remove R[1] from R and add to A
   k = 1; i = 2
   while R is not empty:
       if s[i] > f[k]:
           Remove R[i] from R and add to A
           k = i
       else:
           Remove R[i] from R
       i++
```

- The runtime is $\Theta(n \log n)$ [Sorting dominates]

Optimality of Greedy Activity Selection

- It is not obvious how and whether the greedy activity selection strategy returns an optimal set of intervals i.e., whether or not A is optimal is not clear yet
- However, we can immediately say one thing that A is a compatible set of requests i.e., no two activities in A overlap in time
- We need to show A is optimal i.e., A contains maximum possible non-overlapping activities
- As we don't know yet whether A is optimal, for the purpose of comparison, let us take $\mathcal{O}$ to be an optimal set of activities
- We need to show $|A| = |\mathcal{O}|$
- That is - A contains the same number of intervals as $\mathcal{O}$ and hence is also an optimal solution
- Note: A is what we got using the greedy strategy and $\mathcal{O}$ is an optimal set of activities

Optimality of Greedy Activity Selection

- We are following the book by Kleinberg and Tardos for the proof
- The idea underlying the proof will be to show that the greedy strategy "stays ahead" of the optimal solution $\mathcal{O}$
- It will be very similar to proof by induction
- Let $i_1, \dots, i_k$ be the set of activities in A in the order they are added to A
- Similarly, let $j_1, \dots, j_m$ be the set of activities in $\mathcal{O}$
- Why did we not tell "in the order they are added" for the second case?
- The set $\mathcal{O}$ may have followed some other strategy to get these activities
- However we can always sort the activities $j_1, \dots, j_m$ in increasing finishing time. Lets assume that and this do not cause any loss of generality

Optimality of Greedy Activity Selection

- So $A = \{i_1, \ldots, i_k\}$, $\mathcal{O} = \{j_1, \ldots, j_m\}$; both are sorted in increasing order of the finishing times of the activities
- Note that $|A| = k$ and $|\mathcal{O}| = m$ and our goal is to prove $k = m$
- Lets start by comparing the first activity in both A and $\mathcal{O}$
- i_1 has the least finishing time among all activities. So, $f(i_1) \leq f(j_1)$
- So, if we replace j_1 by i_1 in $\mathcal{O}$, the resulting set still remains optimal as $\{i_1, j_2, \ldots, j_m\}$ still contains the same number of activities (m) which are still compatible
- Now we will prove that for each $r \geq 1$, the r^{th} activity selected by the greedy strategy finishes no later than the r^{th} activity in $\mathcal{O}$
- Thus we will prove that

$$For\ all\ indices\ r \leq k, we\ have\ f(i_r) \leq f(j_r)$$

Optimality of Greedy Activity Selection

- We will prove that $For\ all\ indices\ r \leq k, we\ have\ f(i_r) \leq f(j_r)$
- We have already proved it for $r = 1$
- For $r > 1$, we will assume that the statement is true for $r - 1$ and we will try to prove it for r
- Thus we have $f(i_{r-1}) \leq f(j_{r-1})$ ... (1)
- Since, $\mathcal{O}$ consists of compatible intervals, $f(j_{r-1}) \leq s(j_r)$ ... (2)
- Combining (1) and (2), $f(i_{r-1}) \leq s(j_r)$
- So, activity j_r is one of the possible candidates to be chosen by our greedy strategy. However, the greedy strategy always choses with earliest finish time.
- So, among the available candidates (of which j_r is one), the activity i_r chosen by the greedy strategy has the smallest finish time
- Thus $f(i_r) \leq f(j_r)$. This completes the induction step

Optimality of Greedy Activity Selection

- Thus we have proven that the greedy strategy always "stays ahead" of the optimal solution $\mathcal{O}$
- This is in the sense that – for each r, the r^{th} activity the greedy algorithm selects finishes at least as soon as the r^{th} activity in $\mathcal{O}$
- Now we will see why this implies optimality of the greedy algorithm's set A
- This will be done by contradiction

Optimality of Greedy Activity Selection

- Suppose A is not optimal. Now, as $\mathcal{O}$ is an optimal set, we must have $m > k$
- As $f(i_r) \leq f(j_r)$, $f(i_k) \leq f(j_k)$ [Putting $r = k$] … (3)
- Now, $m > k$. So, there must be at least one activity j_{k+1} in $\mathcal{O}$
- As it starts after j_k is complete, start time of j_{k+1} is after finish time $f(j_k)$ of j_k
- But, by (3), i_k finishes before j_k. So, start time of j_{k+1} is after finish time of i_k
- Thus, after deleting all activities that are not compatible with $i_1, \dots, i_k$, the set of possible activities R, still contains j_{k+1}
- This is a contradiction as we have assumed that the greedy algorithm has stopped at k, and thus R is empty
- This completes the proof that $\textit{the greedy algorithm returns an optimal set of activities}$

Merge Pebbles

- We have piles of pebbles:

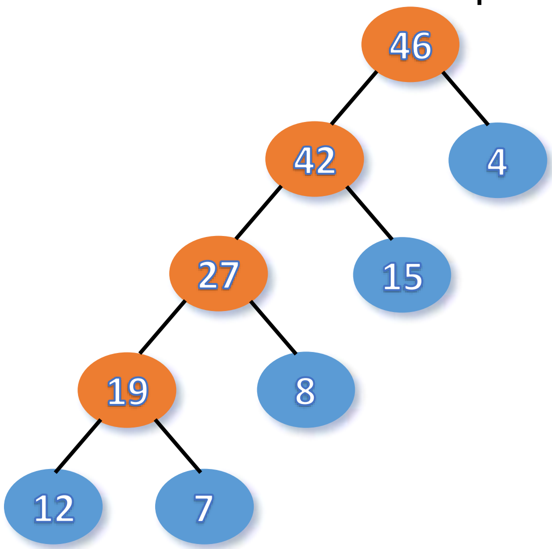
- We want to merge them into one pile, but
  - We can only merge <u>two of them</u> at a time
  - Merging two piles of size a and b costs you $a + b$ units of energy (Let's assume we need to move both piles)
  - It means merging piles of size 12 and 7 results in a new pile of size 19 and costs you 19 units of energy
- How can we merge all of them with **least energy**?

Source: UC Riverside, CS141 course, Fall 2021

Merge Pebbles

- Lets try to merge the piles in the order they are given

- We will use a tree to represent the trace of merging



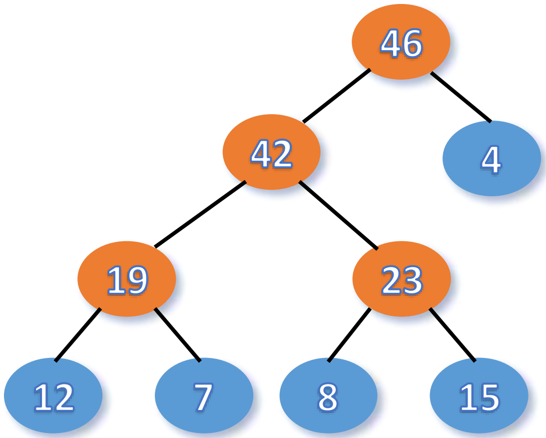
Energy cost: 19 + 27+ 42+ 46 = 134

Source: UC Riverside, CS141 course, Fall 2021

Merge Pebbles – Another Solution

- Lets try to merge the piles two at a time from original piles

- We will use a tree to represent the trace of merging



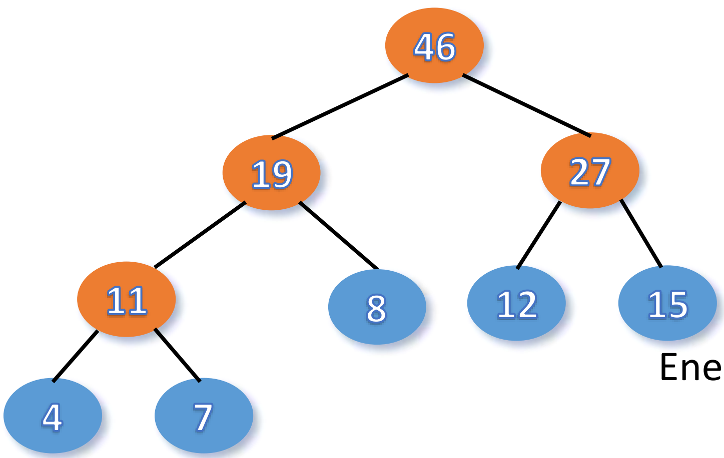
Energy cost: 19 + 23 + 42 + 46 = 130

Source: UC Riverside, CS141 course, Fall 2021

Merge Pebbles – Any Greedy Idea?

- Always merge two smallest piles

- We will use a tree to represent the trace of merging



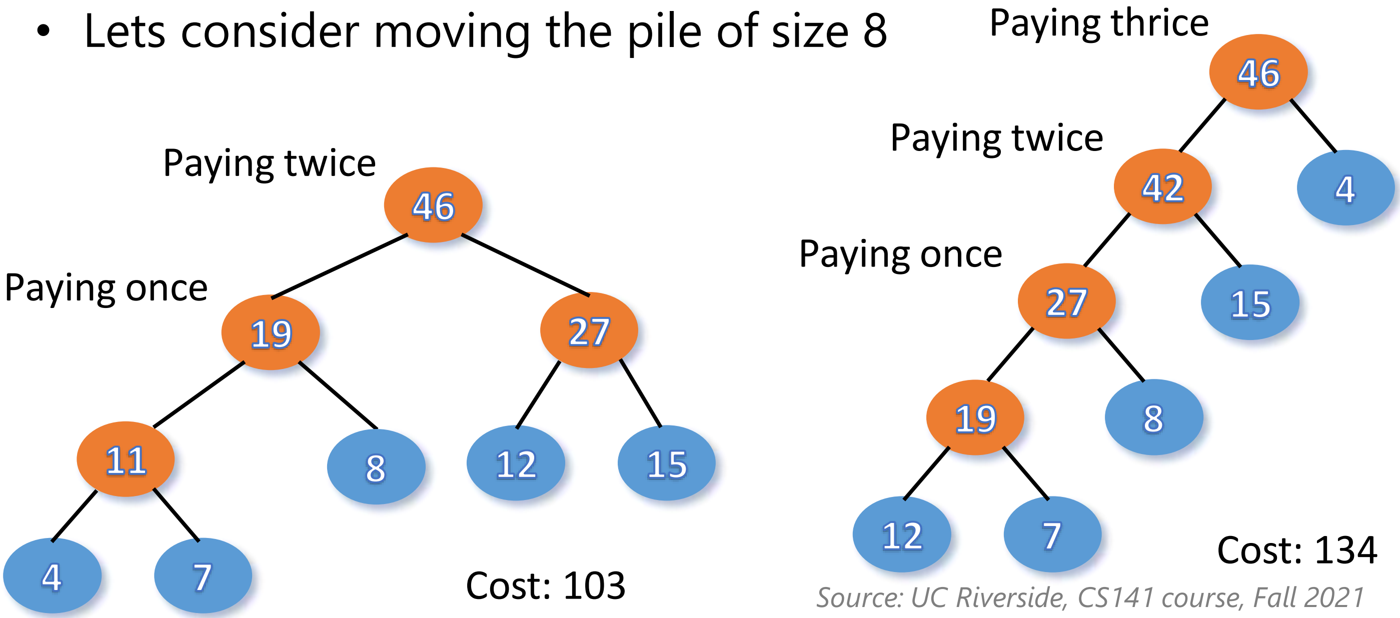
One thing to notice: The total cost sums all the internal nodes

Energy cost: 11 + 19 + 27 + 46 = 103

Source: UC Riverside, CS141 course, Fall 2021

Merge Pebbles – Why Greedy is Good

- You may need to move a pile multiple times (Cost for moving the same pile is incurred multiple times)
- The pile will be charged at all of its ancestors!
- Lets consider moving the pile of size 8

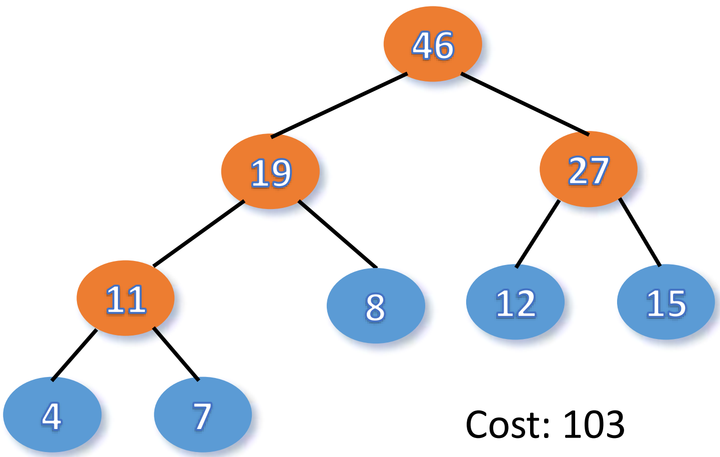


Source: UC Riverside, CS141 course, Fall 2021

Merge Pebbles – Why Greedy is Good

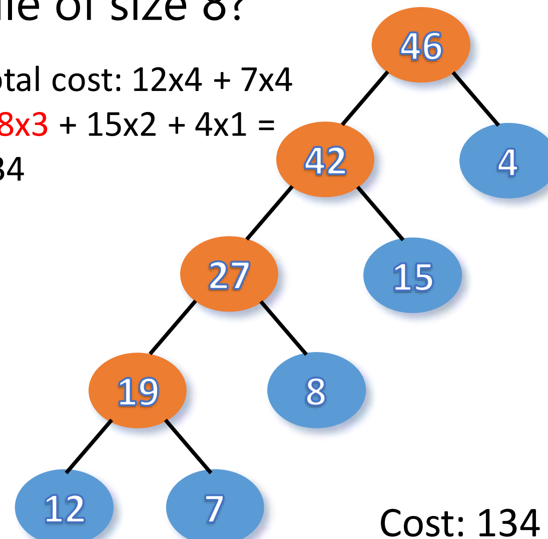
- You may need to move a pile multiple times (Cost for moving the same pile is incurred multiple times)
- The pile will be charged at all of its ancestors!
- How many times do we move the pile of size 8?
  - The height of it (number of ancestors)

Total cost: 4x3 + 7x3 + 8x2 + 12x2 + 15x2 = 103



Cost: 103

Total cost: 12x4 + 7x4 + 8x3 + 15x2 + 4x1 = 134

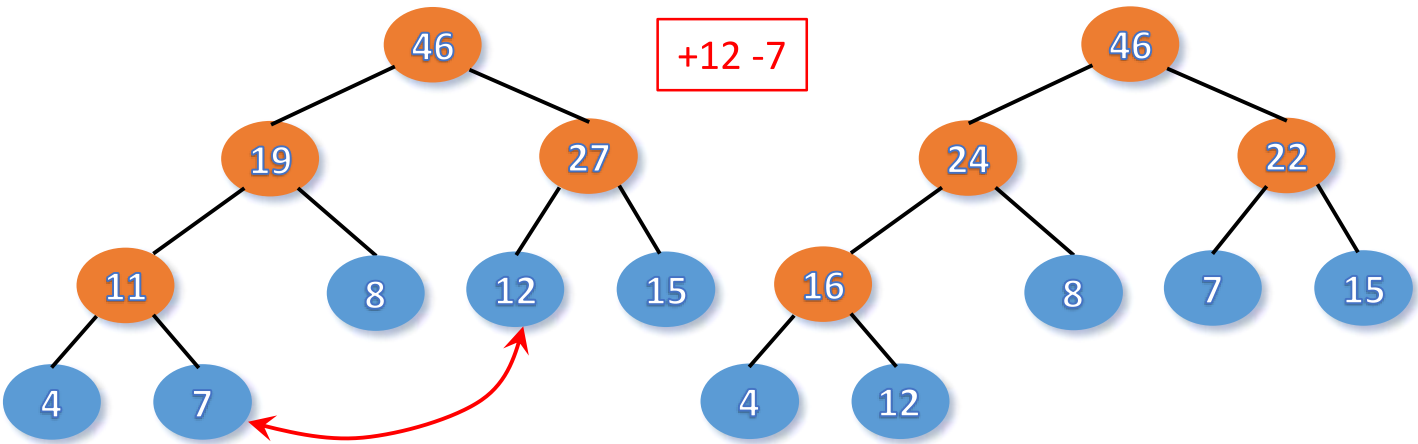


Cost: 134

Source: UC Riverside, CS141 course, Fall 2021

Merge Pebbles – Why Greedy is Good

- $cost = \sum_{i \in L} i \times h(i)$, L is the set of leaf nodes, $h(i)$ is the height of i^{th} leaf node
- We will try to see what if we do not exactly follow the greedy strategy and exchange the order of merging of two piles



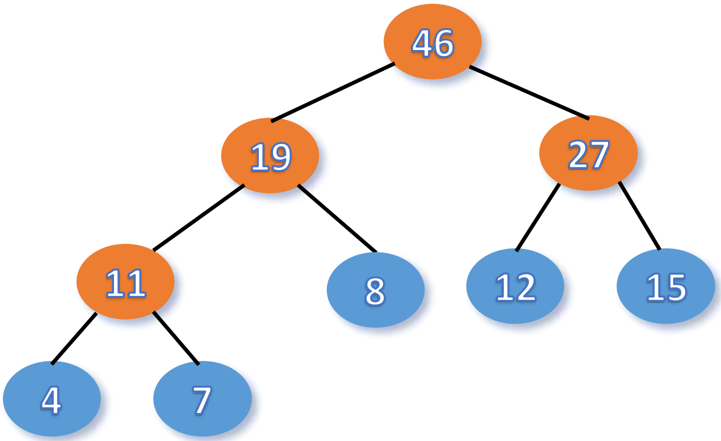
Total cost: 4x3 + 7x2 + 8x2 + 12x3 + 15x2 = 108

Total cost: 4x3 + 7x3 + 8x2 + 12x2 + 15x2 = 103

Source: UC Riverside, CS141 course, Fall 2021

Merge Pebbles – Some Observations

- $cost = \sum_{i \in L} i \times h(i)$, L is the set of leaf nodes, $h(i)$ is the height of i^{th} leaf node



Total cost: 4x3 + 7x3 + 8x2 + 12x2 + 15x2 = 103

- It makes sense to put the smallest piles the deepest
- Since everytime we merge two piles, there are always two leaves in the deepest level
- Once we merge two smallest piles we have the same problem decreased in size by 1 (optimal substructure)
- Why do we care about moving pebble piles?

Source: UC Riverside, CS141 course, Fall 2021

Huffman Codes

- How data is represented in computers?
- Using binary (0’s and 1’s) codes
- Fixed-size codes, e.g., ASCII
  - A: 1000001 (65)
  - B: 1000010 (66)
- Fixed size codes may not necessarily be the best way to store/communicate data
- Any other ways?

Huffman Codes

- Variable size codes like Morse codes
  - A: ● ▬
  - B: ▬ ● ● ●
  - E: ●
  - T: ▬

Image source: https://tinyurl.com/4mncb26k

- Invented in 1800s.
- Not used for storing in computers, rather for communication
- Get more information in www.youtube.com/watch?v=iy8BaMs_JuI
- Note: The length of each code is not same.
- Why? Any advantage?

Fixed Length vs Variable Length Code

- Suppose we have a 100,000-character data file that we wish to store compactly
- The file contains only 6 characters, appearing with the following frequencies

	a	b	c	d	e	f
Frequency in '000s	45	13	12	16	9	5

- We would like to find a binary code that encodes the file using as few bits as possible, i.e., the compression is maximum

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- Fixed length code requires 300,000 bits to encode the file
- The variable length code requires (45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4)*1000 = 224,000 bits
- A savings of approximately 25%

Lets Go Back to Morse Codes

A •–	J •–––	S •••
B –•••	K –•–	T –
C –•–•	L •–••	U ••–
D –••	M ––	V •••–
E •	N –•	W •––
F ••–•	O –––	X –••–
G ––•	P •––•	Y –•––
H ••••	Q ––•–	Z ––••
I ••	R •–•	

"SOS": ••• ––– •••

IJS: (..)(.---)(...)

STZE: (…)(-)(--..)(.)

IAGI: (..)(.-)(--.)(..)

VMS: (...-)(--)(...)

- Anything you can notice?
- All the coded words are same
- Actually 'pause' plays a role in Morse code

Source: UC Riverside, CS141 course, Fall 2021

Prefix Codes

- No code should sit in front of any other code
- Termed as “Prefix code” [Though the book rightly says “Perhaps prefix-free codes would be a better name”]
- Morse code is not “Prefix code” [Or called as Non-prefix code]
- Encoding means simply concatenate all the codes

Character	Prefix code	Non-prefix code
a	0	00
b	101	001
c	100	11
d	111	111
e	1101	01
f	1100	010

- abd -> 0101111

Prefix Codes

Character	Prefix code	Non-prefix code
a	0	00
b	101	001
c	100	11
d	111	111
e	1101	01
f	1100	010

- Decoding is unambiguous
- Example:
  - Message: 'DABA'
  - Encoded message: 11101010
  - Decoding "11101010" – greedily decode it!

Optimum Prefix Codes

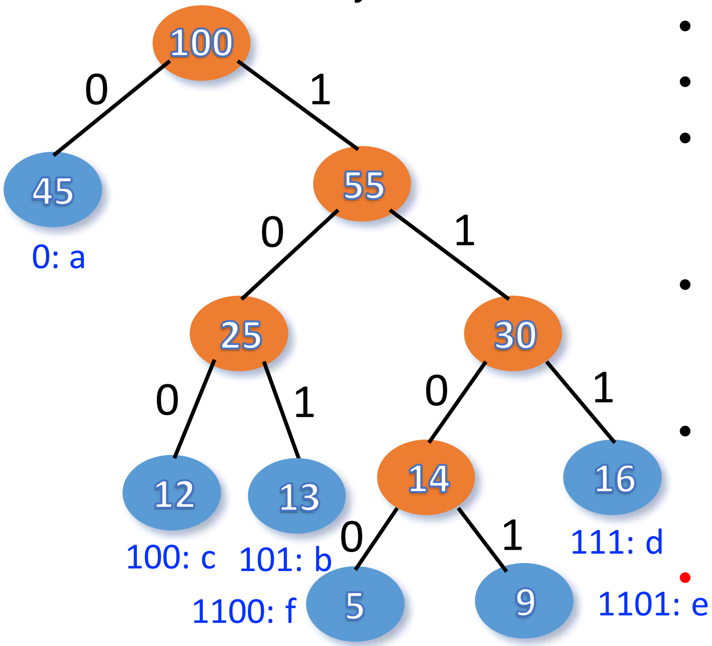
- Given the codewords for different characters, we can easily compute the number of bits required to encode a file
- Given an alphabet $A = \{a_1, \dots, a_n\}$ with frequency distribution $f(a_i)$, codeword $c(a_i)$ and length $L(c(a_i))$, the number of bits required to encode the file is

$$B(T) = \sum_{i=1}^{n} f(a_i)L(c(a_i))$$

- An optimum prefix code is a binary prefix code C for A such that it minimizes the total number of bits $B(T)$
- Huffman developed a nice greedy algorithm for solving this problem and producing a minimum-cost prefix code.
- The code that it produces is called a Huffman code

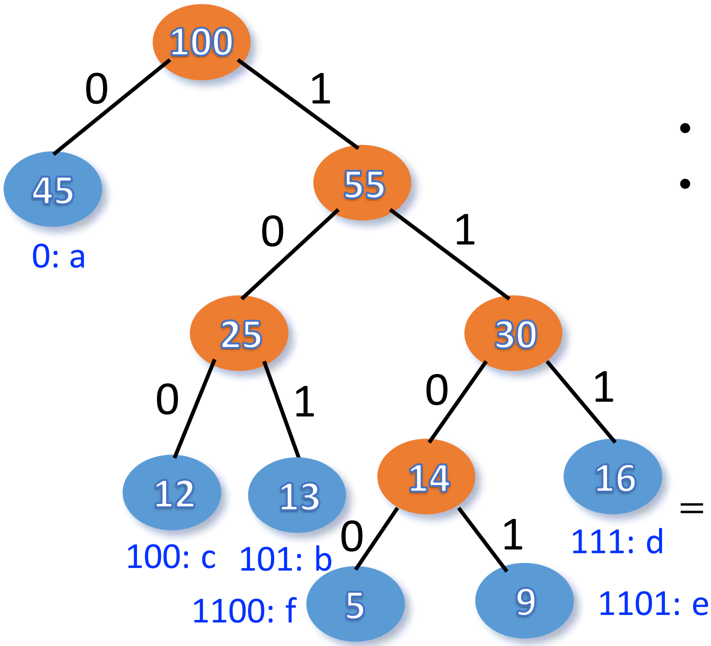
Relation between Prefix Codes and Tree

- Prefix codes can be represented by the leaves of a binary tree
- An optimal code for a file is always represented by a full binary tree, in which every nonleaf node has two children



- A left edge means 0
- A right edge means 1
- Each leaf is a character with its code found by traversing to the leaf from the root
- Each leaf is labeled with the frequency of occurrence of the corresponding character
- Each internal node contains the sum of the frequencies of the leaves of its subtrees
- Any similarity to something we have seen earlier?

Relation between Prefix Codes and Tree

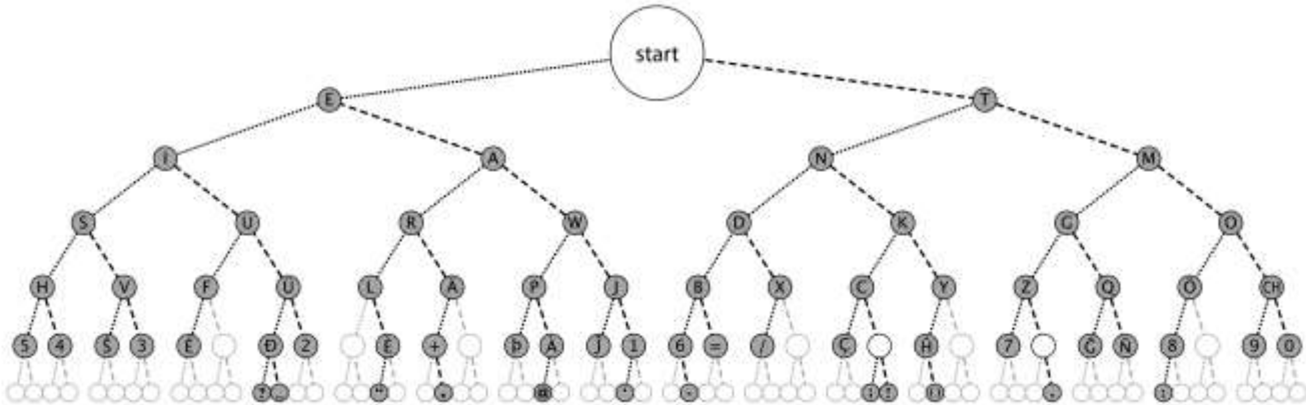


- If C is the alphabet set, then the tree for an optimal prefix code has exactly $|C|$ leaves
- And exactly $|C| - 1$ internal nodes
- Given a tree the number of bits required to encode the file is

$$B(T) = \sum_{i=1}^{n} f(a_i)L(c(a_i))$$

$$= \sum_{i=1}^{n} f(a_i)d(a_i)$$

Morse Code is Non-prefix



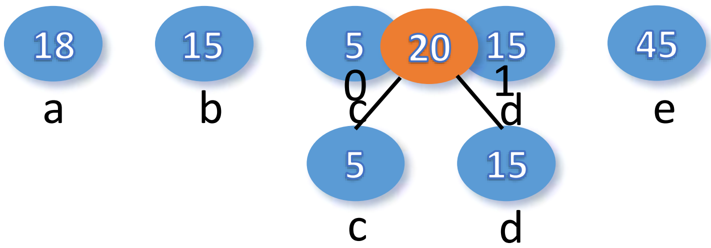
Taken from wikipedia. Attribution to author: By The original uploader was Aris00 at English Wikipedia. - Transferred from en.wikipedia to Commons. Transfer was stated to be made by User:Ddxc., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3177632

Finding Huffman Code

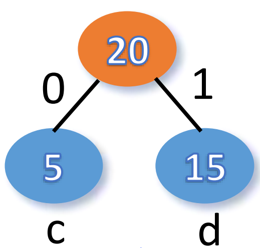
- Step 1: Pick two letters x, y from alphabet A with the smallest frequencies and create a subtree that has these two characters as leaves (greedy idea)
- Label the root of this subtree as z
- Step 2: Set frequency $f(z) = f(x) + f(y)$
- Remove x, y and add z creating new alphabet
- $A' = \{A \cup \{z\}\} \setminus \{x, y\}$
- Note that $|A'| = |A| - 1$
- Repeat this procedure, called merge with new alphabet A' until an alphabet with only one symbol is left
- The resulting tree is the Huffman tree giving Huffman code

Example of Huffman Coding

- Let the alphabet A along with its frequency distribution is
- New alphabet A' is

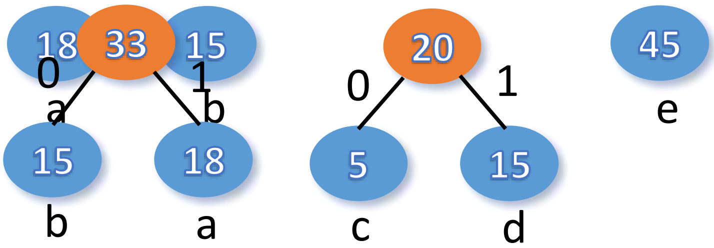


In the first step, merge c and d

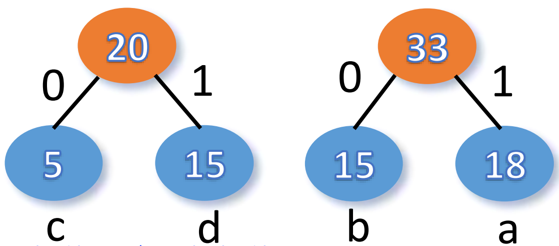


Example of Huffman Coding

- New alphabet A' is

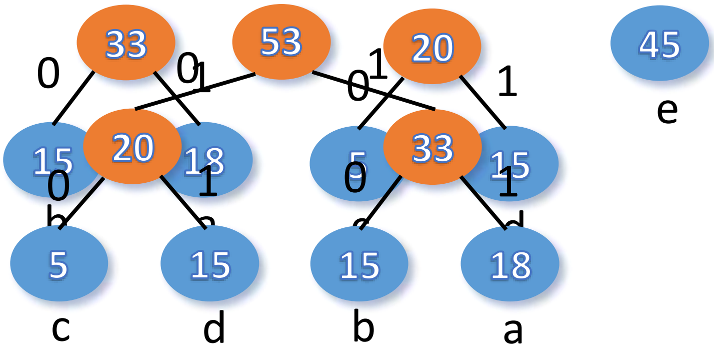


Next, merge a and b

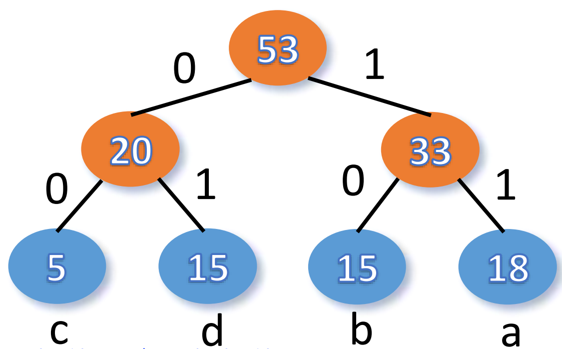


Example of Huffman Coding

- New alphabet A' is

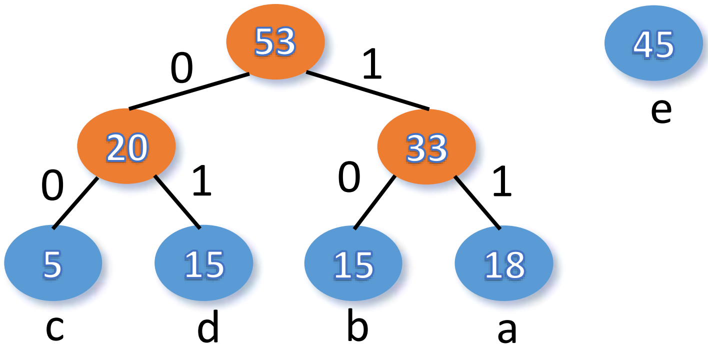


Next, merge nodes with freq 20 and 33

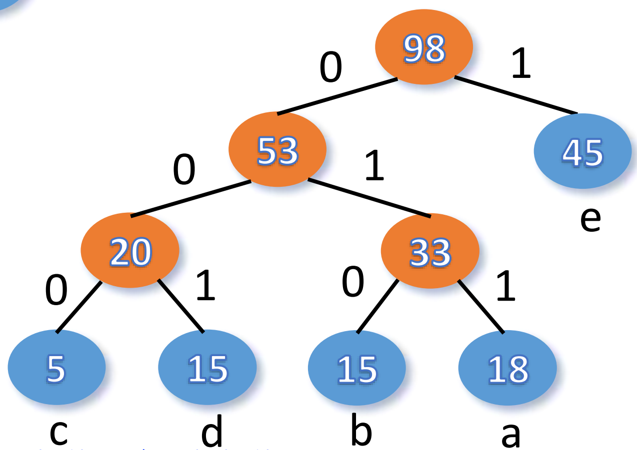


Example of Huffman Coding

- New alphabet A' is empty



Next, merge node with freq 53 and the only remaining character e

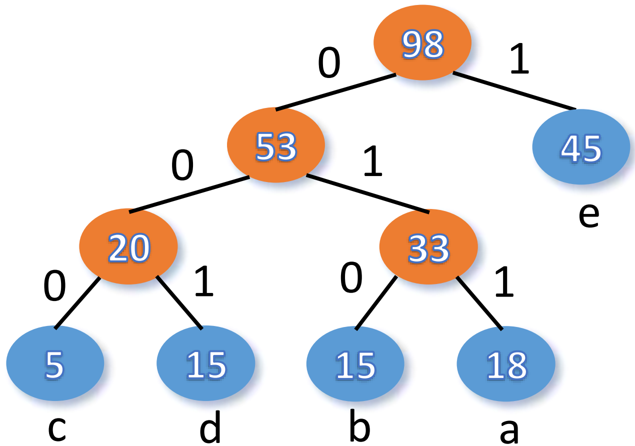


Example of Huffman Coding

- Algorithm terminates and Huffman tree is obtained
- The Huffman codes are:

$$a: 011, b: 010, c: 000, d: 001, e: 1$$

- Running time is $\Theta(n \log n)$ [Proof is deferred]

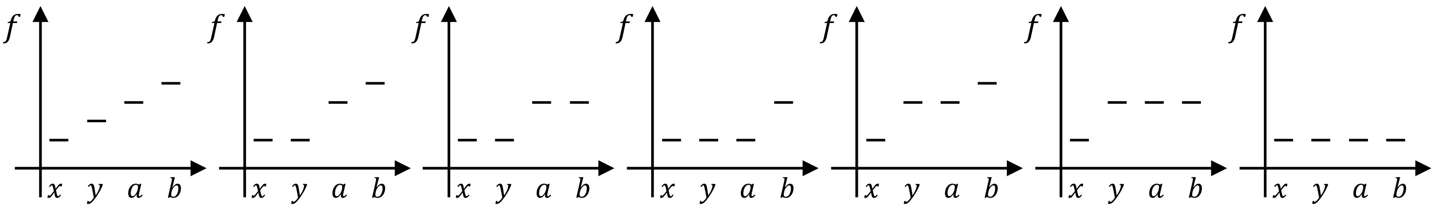


Optimality of Huffman Coding

- Lemma 1: *Let x, y be two characters in alphabet A with two smallest frequencies. Then there exists an optimal prefix code tree for A in which the codewords for x and y are sibling leaves in the tree in the lowest level*
- Proof: (The idea) – Take a tree T representing an arbitrary optimal prefix code tree. Assume some other characters a, b sit at the bottom as sibling nodes.
  We will try to modify this tree such that x, y (sitting somewhere else in the tree) are exchanged with a, b.
  If the modified tree is also at least as good as T then we are done proving the lemma
- Quick quiz: Can there be two or more optimal trees?

Optimality of Huffman Coding

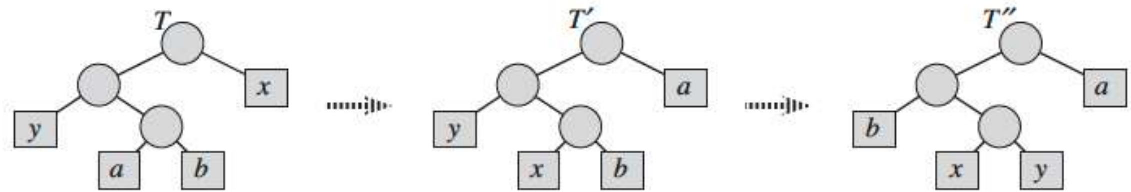
- Proof: – Let us consider the tree T and a, b sit at the maximum depth as sibling nodes
- Without loss of generality, lets assume $f(a) \leq f(b)$ and $f(x) \leq f(y)$. Since, x, y have the lowest two frequencies, the above $\Rightarrow$ $f(x) \leq f(a)$ and also $f(y) \leq f(b)$



- The last case makes the lemma trivially true (as exchanging a, b and x, y does not change anything on the objective function)
- So we will assume $f(x) \neq f(b)$ [Other wise that will make $f(x) = f(y) = f(a) = f(b)$]

Optimality of Huffman Coding

- We exchange the positions of a and x in T to produce a tree T'
- Then we exchange the positions of b and y in T' to produce T''
- In T'', x and y are sibling leaves at maximum depth



- $B(T) - B(T') = \sum_{i \in A} f(i) d_T(i) - \sum_{i \in A} f(i) d_{T'}(i)$

$= f(x)d_T(x) + f(a)d_T(a) - (f(x)d_{T'}(x) + f(a)d_{T'}(a))$
[Rest of the nodes, as unchanged, get cancelled]

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

$= \underbrace{\big(f(a) - f(x)\big)}_{\geq 0}\underbrace{\big(d_T(a) - d_T(x)\big)}_{\geq 0}$

≥ 0

Optimality of Huffman Coding

- $B(T') \leq B(T)$
- Similarly $B(T'') \leq B(T')$
- So, $B(T'') \leq B(T)$
- But, T is optimal. So, $B(T) \leq B(T'')$
- So, $B(T) = B(T'')$
- Thus, T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth– from which the lemma follows

Optimality of Huffman Coding

- Let x and y be two characters in alphabet A with minimum frequency. Let A' be the alphabet A with x and y removed and a new character z added, so that $A' = A - \{x, y\} \cup \{z\}$. Frequencies for characters in A' is same as for A, except that $f(z) = f(x) + f(y)$. Let T' be any optimal prefix code tree for A'. Then the tree T, obtained from T' by replacing the leaf node z with an internal node having x and y as children, represents an optimal prefix code tree for A
- What do we get if the above lemma is true?
- This is exactly how Huffman code is formed. Once, you get the final z, you can make the whole tree by recursively replacing z with its two children

Optimality of Huffman Coding

- Lemma: *Let x and y be two characters in alphabet A with minimum frequency. Let A' be the alphabet A with x and y removed and a new character z added, so that $A' = A - \{x, y\} \cup \{z\}$. Frequencies for characters in A' is same as for A, except that $f(z) = f(x) + f(y)$. Let T' be any optimal prefix code tree for A'. Then the tree T, obtained from T' by replacing the leaf node z with an internal node having x and y as children, represents an optimal prefix code tree for A.*
- Lets try to express $B(T)$ in terms of $B(T')$
- For each character $i \in A - \{x, y\}$, we have $d_T(i) = d_{T'}(i)$. So, $f(i)d_T(i) = f(i)d_{T'}(i)$ … (1)
- Since, $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have $f(x)d_T(x) + f(y)d_T(y) = (f(x) + f(y))(d_{T'}(z) + 1) = f(z)d_{T'}(z) + (f(x) + f(y))$ … (2)
- Adding (1) for all i to both sides of (2), we get,

$$B(T) = B(T') + f(x) + f(y) \text{ … (3)}$$

Optimality of Huffman Coding

- Rearranging, $B(T') = B(T) - f(x) - f(y)$ ... (3)
- We will prove the lemma by contradiction. Suppose T does not represent an optimal prefix code tree for A
- Then there exists an optimal tree T'' such that $B(T'') < B(T)$ ... (4)
- Again, by the previous lemma, T'' has x and y as siblings
- Let T''' be the tree T'' with common parent of x and y replaced by a <u>leaf</u> z with frequency $f(z) = f(x) + f(y)$
- Then, $B(T''') = B(T'') - [f(x)d_{T''}(x) + f(y)d_{T''}(y)] + f(z)(d_{T''}(x) - 1)$

$= B(T'') - [f(x) + f(y)]d_{T''}(x) + f(z)(d_{T''}(x) - 1)$ [as, $d_{T''}(y) = d_{T''}(x)$]

$= B(T'') - f(z)d_{T''}(x) + f(z)(d_{T''}(x) - 1)$ [as, $f(x) + f(y) = f(z)$]

$= B(T'') - f(z) = B(T'') - f(x) - f(y)$

$< B(T) - f(x) - f(y)$ [by, (4)]

$= B(T')$ [by, (3)] → This is a contradiction that T' is optimal for A'. Thus T must be optimal for A

Thank You