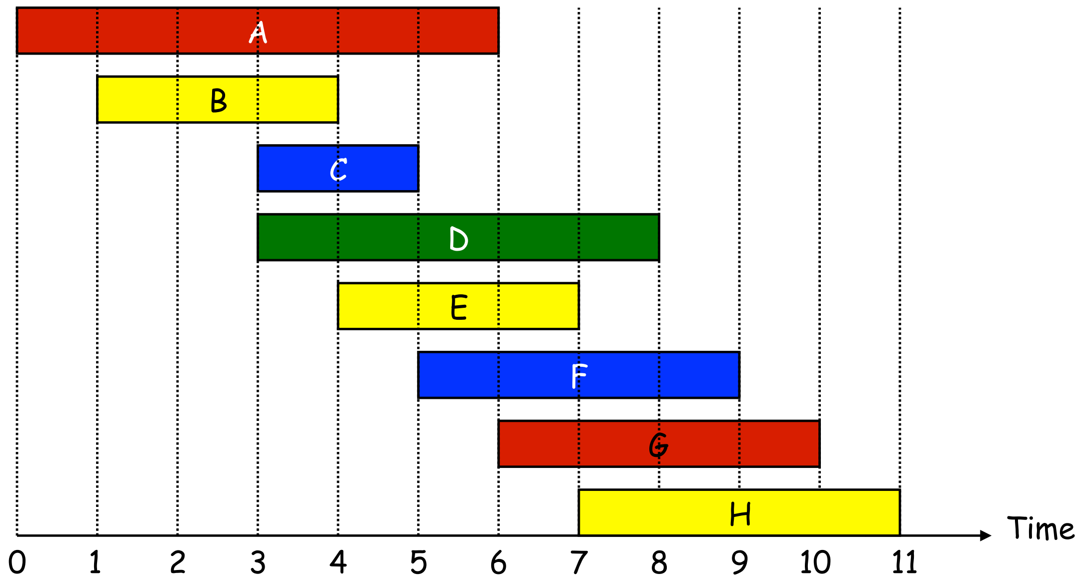
Greedy Algorithms

COMP 6651 – Algorithm Design Techniques

Denis Pankratov

Example: Interval Scheduling problem (CLRS 16.1)

- Suppose you have a list of computational jobs which can be executed on a server
- Job j starts at time s_j and finishes at f_j
- Two jobs are **compatible** if they don't overlap in time
- The **goal** is to find a maximum subset of mutually compatible jobs



Some feasible solutions:

$\{A, G\}$

$\{B, E, H\}$

$\{C, F\}$

$\{D\}$

Optimal solution: $\{B, E, H\}$

This is what is known as a **combinatorial optimization problem**

Basic structure:

- **Input** is a collection $\mathcal{C} = \{I_1, I_2, \ldots, I_n\}$ of n items
- Algorithm needs to produce a **solution** S
- There is an **objective function** $OBJ(\mathcal{C}, S) \in \mathbb{R} \cup \{\pm\infty\}$
- There is also a **goal function** g, which is either $\min$ or $\max$

Interval Scheduling problem as a combinatorial optimization problem:

- **Input** is a collection of jobs modelled as half-open intervals $I_j = [s_j, f_j)$
- **Solution** is a subset of n jobs $S \subseteq \mathcal{C}$
- Objective $OBJ(\mathcal{C}, S) = \begin{cases} |S| & \text{if all jobs in S are compatible} \\ -\infty & \text{otherwise} \end{cases}$
- The **goal** function is $\max$

Solution S can be represented as a sequence of decisions $d_1, \dots, d_n$ where

$$d_j = \begin{cases} 1, & \text{job j is included in the solution} \\ 0, & \text{job j is excluded from the solution} \end{cases}$$

Greedy/Myopic Strategy

Build up the solution by making decisions for one input item after another in some natural order that offer the most obvious and immediate benefit

If greedy algorithms had a motto it would be **YOLO**

Observe the **irrevocable** nature of **decisions** – once a decision is made, the greedy algorithm doesn’t go back and reconsider the decision

Greedy algorithms are **easy to design** and are often **extremely efficient**, but are **seldom correct**

What order are input items considered in?

Can have a dramatic impact

Back to Interval Scheduling

$GreedyTemplate(\mathcal{C})$

reorder jobs in $\mathcal{C}$ in some way

initialize solution $S \leftarrow \emptyset$

$\boldsymbol{for}\ j = 1\ \boldsymbol{to}\ n$:

$\boldsymbol{if}\ I_j$ is compatible with all jobs in S:

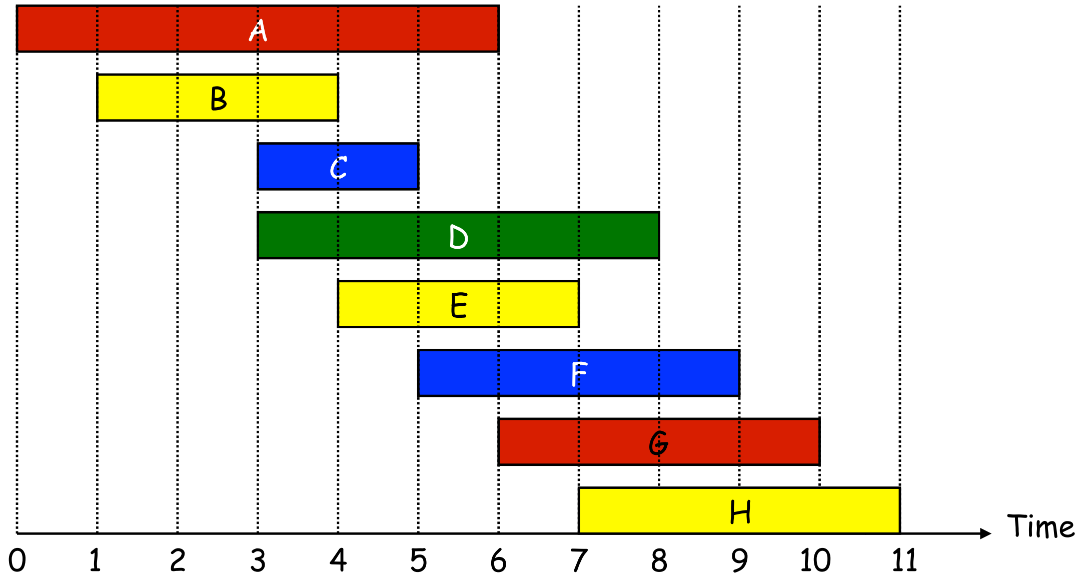
$S \leftarrow S \cup \{I_j\}$

$\boldsymbol{return}\ S$

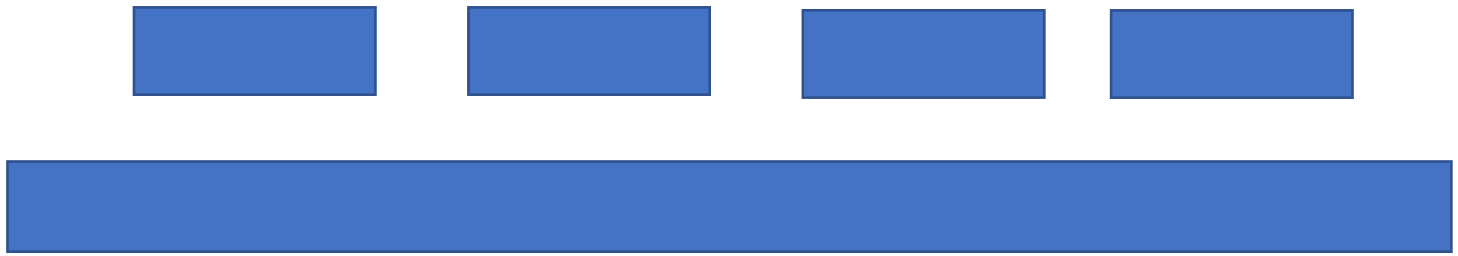
What order to use?

Possible natural orders

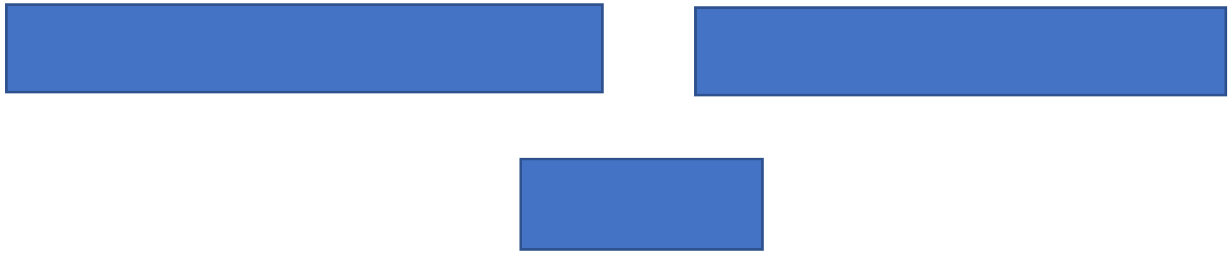
- **Earliest start time**: consider jobs according to non-decreasing s_j
- **Earliest finish time**: consider jobs according to non-decreasing f_j
- **Shortest interval**: consider jobs according to non-decreasing $f_j - s_j$
- **Fewest conflicts**: for each job j, count the remaining number of conflicting jobs c_j. Schedule according to non-decreasing c_j



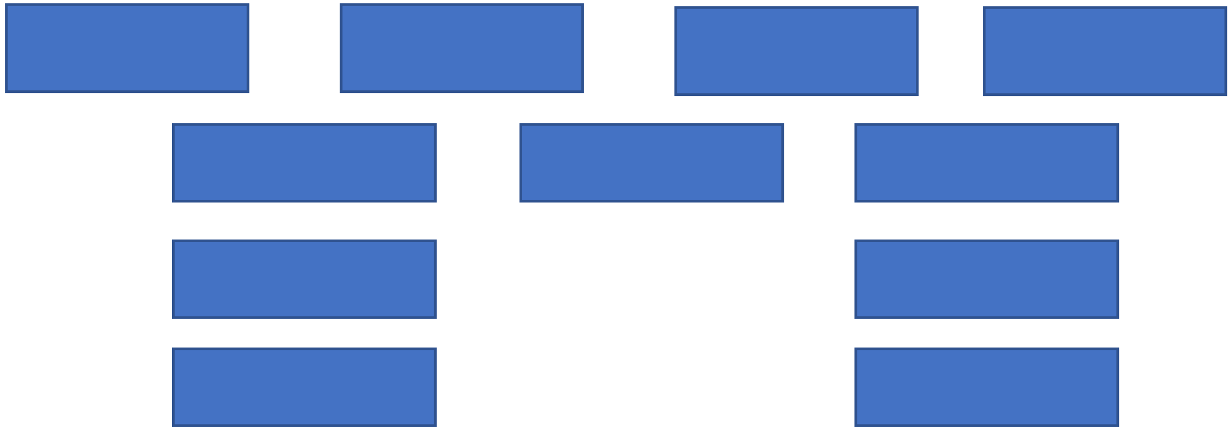
Counterexamples



Earliest start time counterexample



Shortest interval counterexample



Fewest conflicts counterexample

Turns out that earliest finishing time (EFT) algorithm is optimal

Observe that it is easier to come up with an incorrect greedy algorithm than it is to come up with a correct greedy algorithm

Therefore, proving that EFT is optimal is extremely important

This is called proof of optimality/proof of correctness/correctness argument

Theorem. EFT solves the Interval Scheduling problem optimally.

Proof.

Order jobs by non-decreasing finishing times $I_1, \dots, I_n$ such that $f_1 \leq f_2 \leq \cdots \leq f_n$

Let S denote the solution produced by EFT

Consider prefixes of length i, i.e., $I_1, \dots, I_i$

Let S_i denote $S \cap \{I_1, \dots, I_i\}$

Let $P(i)$ denote the statement that "S_i can be extended to some optimal solution"

Formally this means that there exists optimal solution OPT_i such that

$$S_i \subseteq OPT_i \subseteq S_i \cup \{I_{i+1}, \dots, I_n\}$$

… proof (continued)

$P(i)$ = "S_i can be extended to some optimal solution"

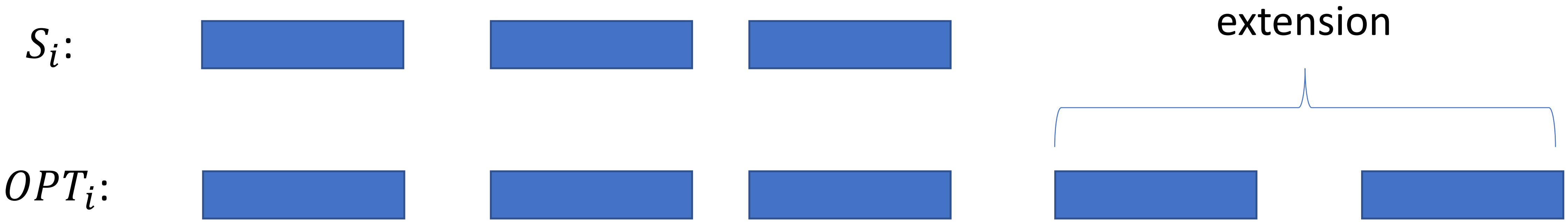
We wish to prove that $P(i)$ is true for all $i \in \{0, 1, \ldots, n\}$

Observe that $P(n) = S_n$ can be extended to some optimal solution, but there are no remaining input items, so S_n must be itself optimal

We shall prove $P(i)$ by induction on i

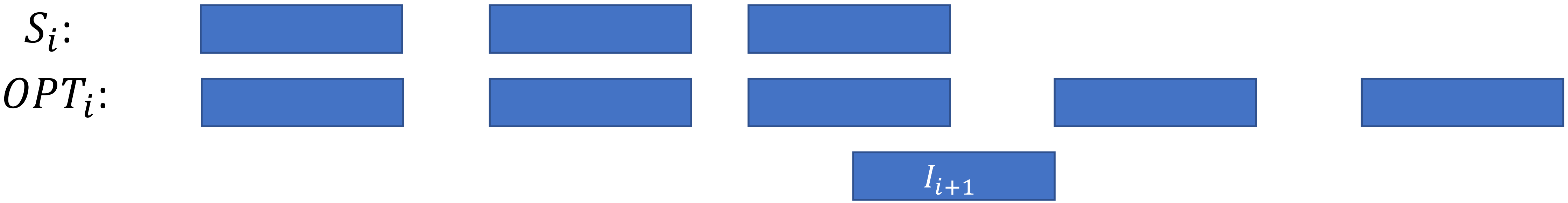
Base case $i = 0$: trivial

Inductive assumption: assume $P(i)$ is true for some ≥ 0



Inductive step:

Case 1: if $s_{i+1} < f_i$ then $S_{i+1} = S_i$, and we can set $OPT_{i+1} = OPT_i$



Case 2A: if $s_{i+1} \geq f_i$ and $I_{i+1} \in OPT_i$. Then we can set $OPT_{i+1} = OPT_i$

Case 2B: if $s_{i+1} \geq f_i$ and $I_{i+1} \notin OPT_i$. Then we can set $OPT_{i+1} = \left(OPT_i - \{I_j\}\right) \cup \{I_{i+1}\}$

QED

Outstanding issues: pseudocode, runtime?

$GreedyIntervalSelection(I[1..n])$

// each job $I[j]$ has start time attribute $I[j].s$ and finish time $I[j].f$

sort $I[1..n]$ by non-decreasing attribute f

$S \leftarrow \{I[1]\}$

$k \leftarrow 1$

$\boldsymbol{for}\ j = 2\ \boldsymbol{to}\ n$

$\quad \boldsymbol{if}\ I[j].s \geq I[k].f$

$\quad\quad S \leftarrow S \cup \{I[j]\}$

$\quad\quad k \leftarrow j$

$\boldsymbol{return}\ S$

$O(n \log n)$ time

$O(n)$ time

Overall $O(n \log n)$ time

Coding (CLRS 16.3)

ASCII encoding

- 128 characters (about 100 printable characters + special characters)
- Each character needs $\lfloor \log 128 + 1 \rfloor = 7$ bits

Character	Decimal	Binary
$\vdots$	$\vdots$	$\vdots$
A	65	1000001
B	66	1000010
C	67	1000011
D	68	1000100
E	69	1000101
$\vdots$	$\vdots$	$\vdots$

Consider a toy example of a small alphabet with the following encoding

Character	Decimal	Binary
c	0	000
o	1	001
m	2	010
p	3	011
u	4	100
space	5	101
newline	6	110

Consider a particular text file with text over the toy-example alphabet

Each character appears with a certain frequency in the file

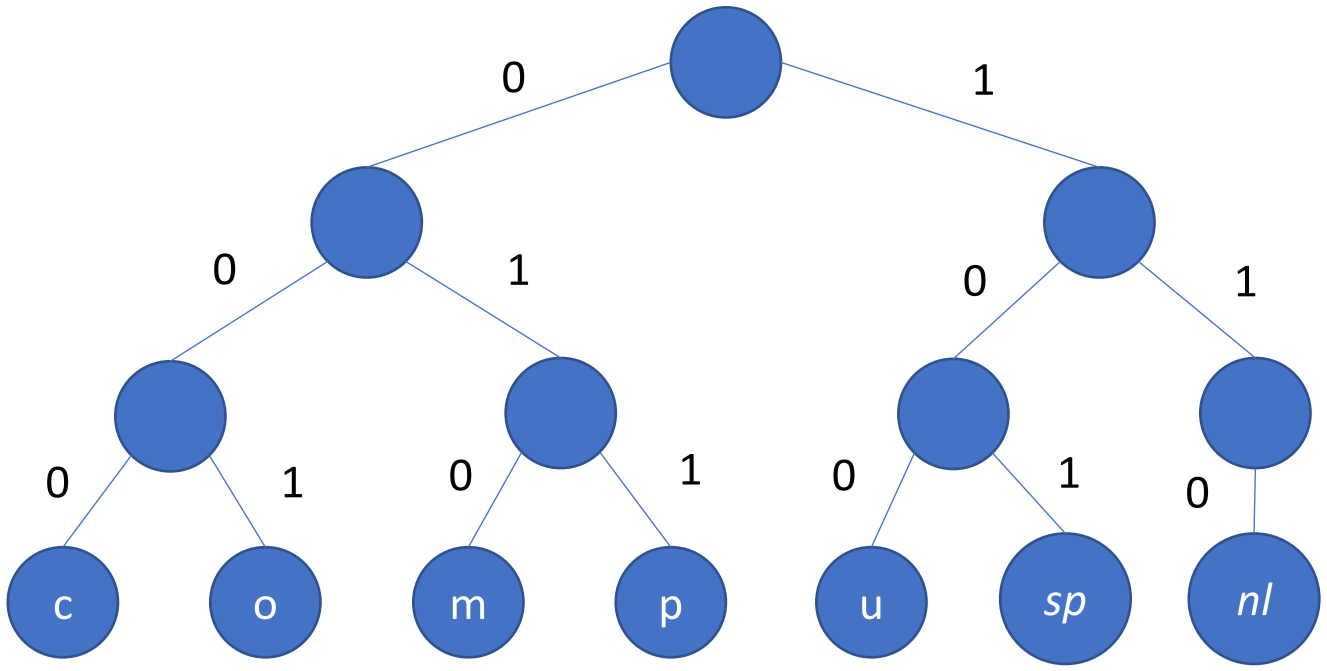
Since each character is encoded with 3 bits, we can compute total filesize

Character	Decimal	Binary	Frequency	Total Bits
c	0	000	20	60
o	1	001	35	105
m	2	010	15	45
p	3	011	5	15
u	4	100	5	15
space	5	101	15	45
newline	6	110	10	30
			TOTAL: 105	**TOTAL: 315**

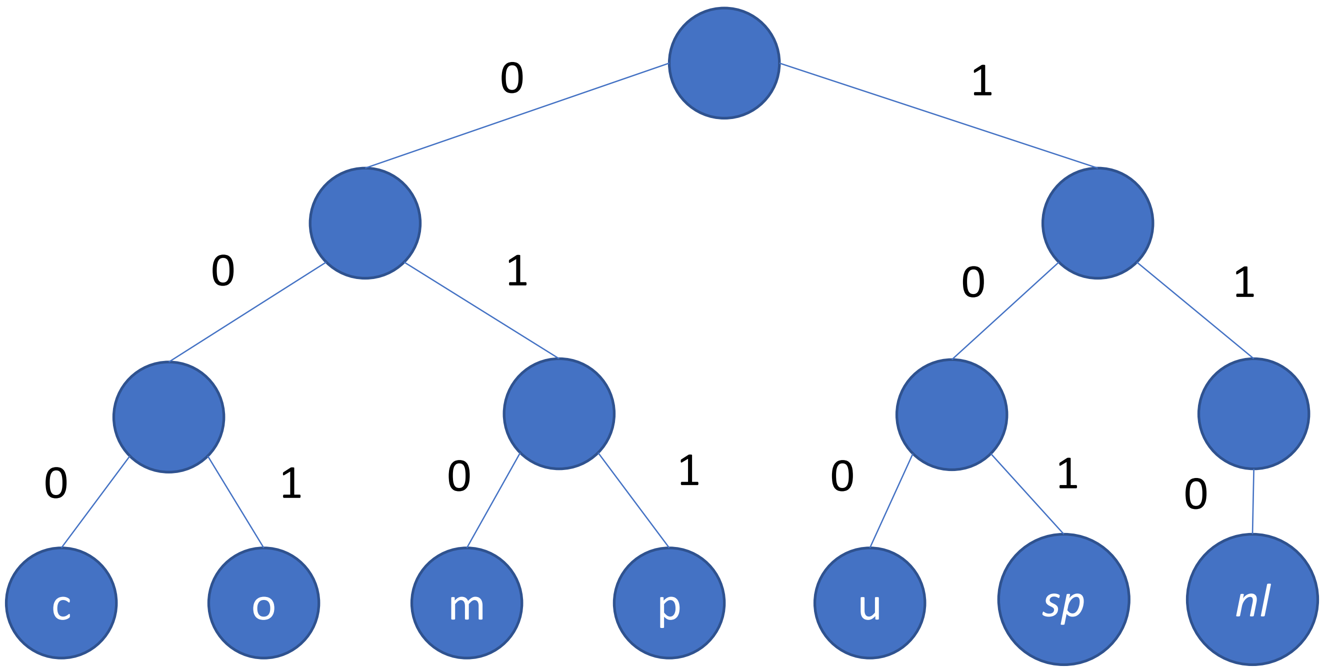
Prefix Trees

- Binary tree
- Each edge is labelled 0 or 1
- Leaves are labelled with characters of the alphabet
- Encoding of a character = concatenation of all labels on root-to-leaf path

Character	Binary
c	000
o	001
m	010
p	011
u	100
space (sp)	101
newline (nl)	110



Character	Binary
c	000
o	001
m	010
p	011
u	100
space (sp)	101
newline (nl)	110



d_i - depth of character i

f_i - frequency of character i in the file

$$\text{File size} = \sum_i d_i f_i$$

Question: Can we reorganize the tree to minimize the file size?

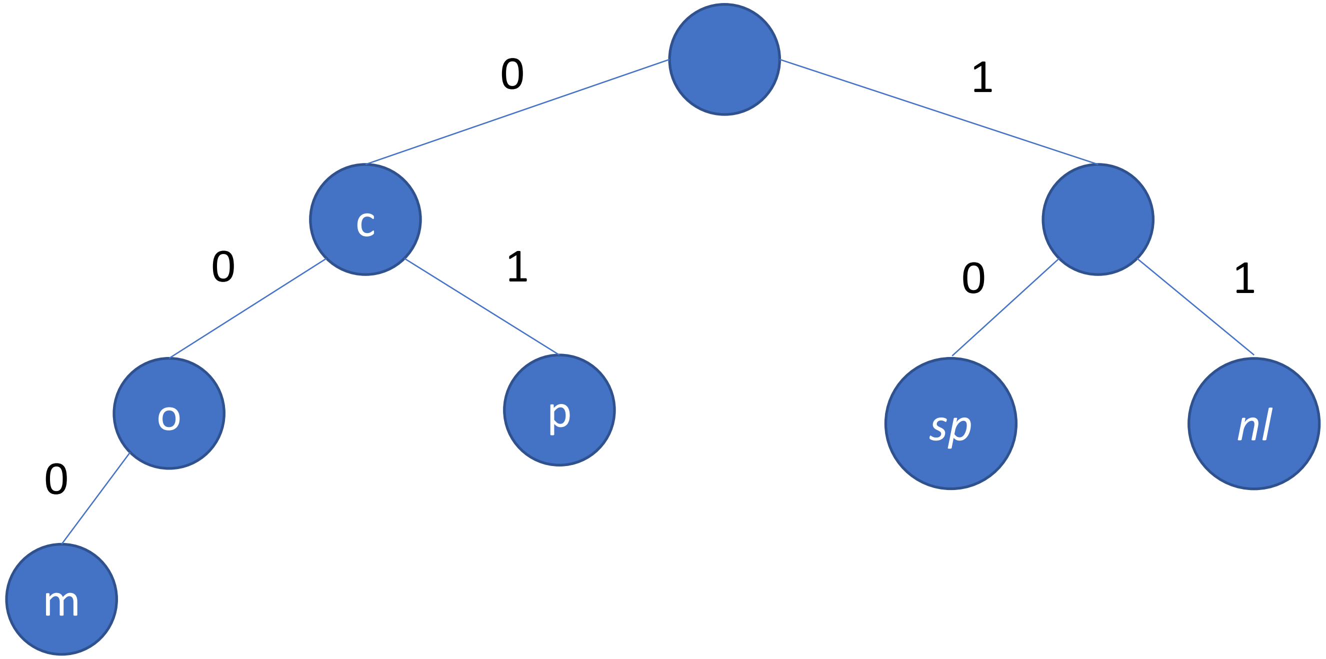
Prefix-free property:

Encoding of one character is not a prefix of an encoding of another character

Important for unique on-the-fly decoding

Equivalent to characters in prefix-tree appearing on the leaves only

What would go wrong if we put characters on internal nodes?



Consider encoded string:

$$\underbrace{000}_{m}\underbrace{11}_{nl}\underbrace{0}_{c}$$

$$\underbrace{00}_{o}\underbrace{01}_{p}\underbrace{10}_{sp}$$

Coding problem

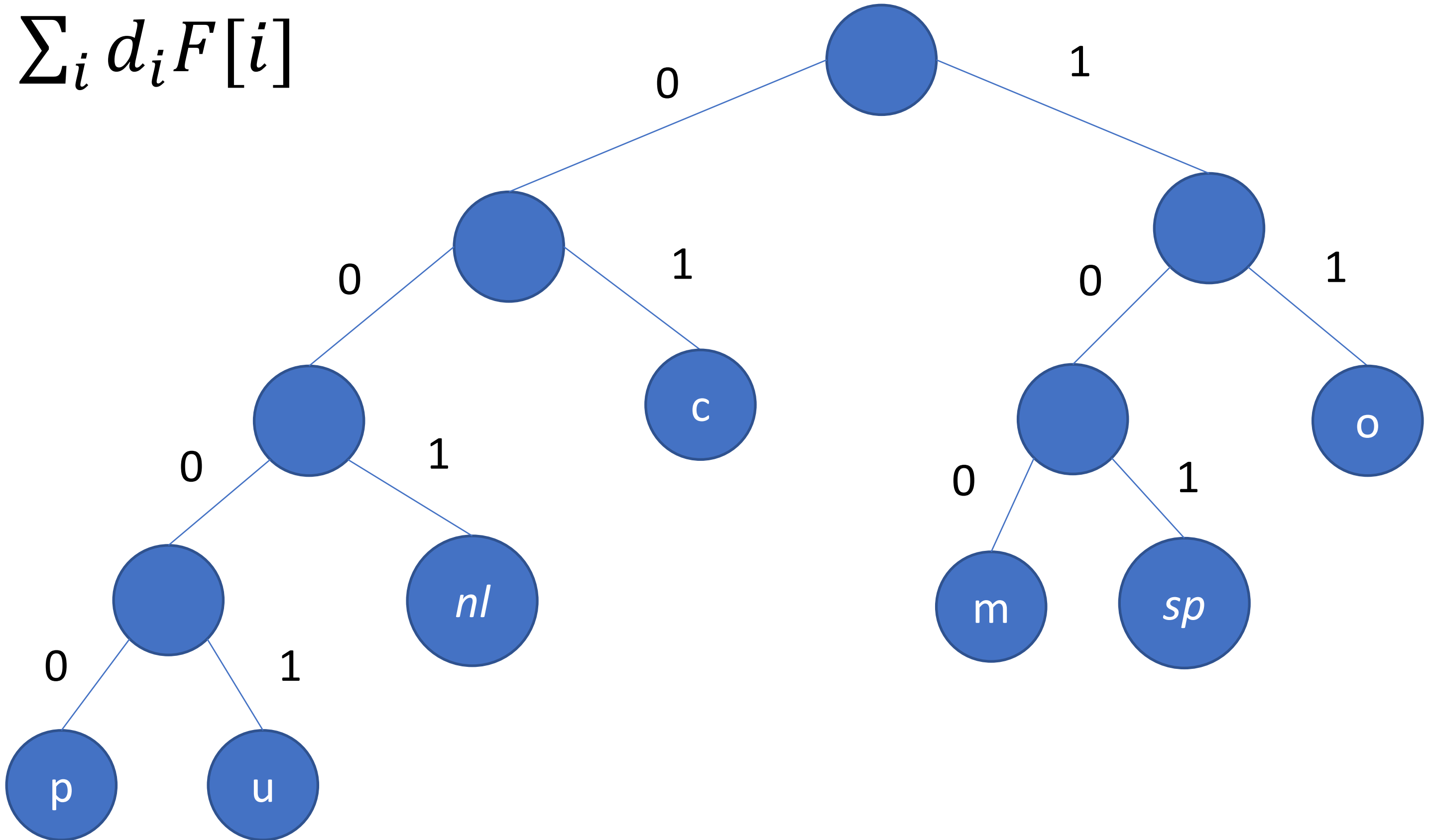
Input: $F[1..n]$ – array of frequencies of n characters

Output: prefix tree T where d_i is the depth of character i

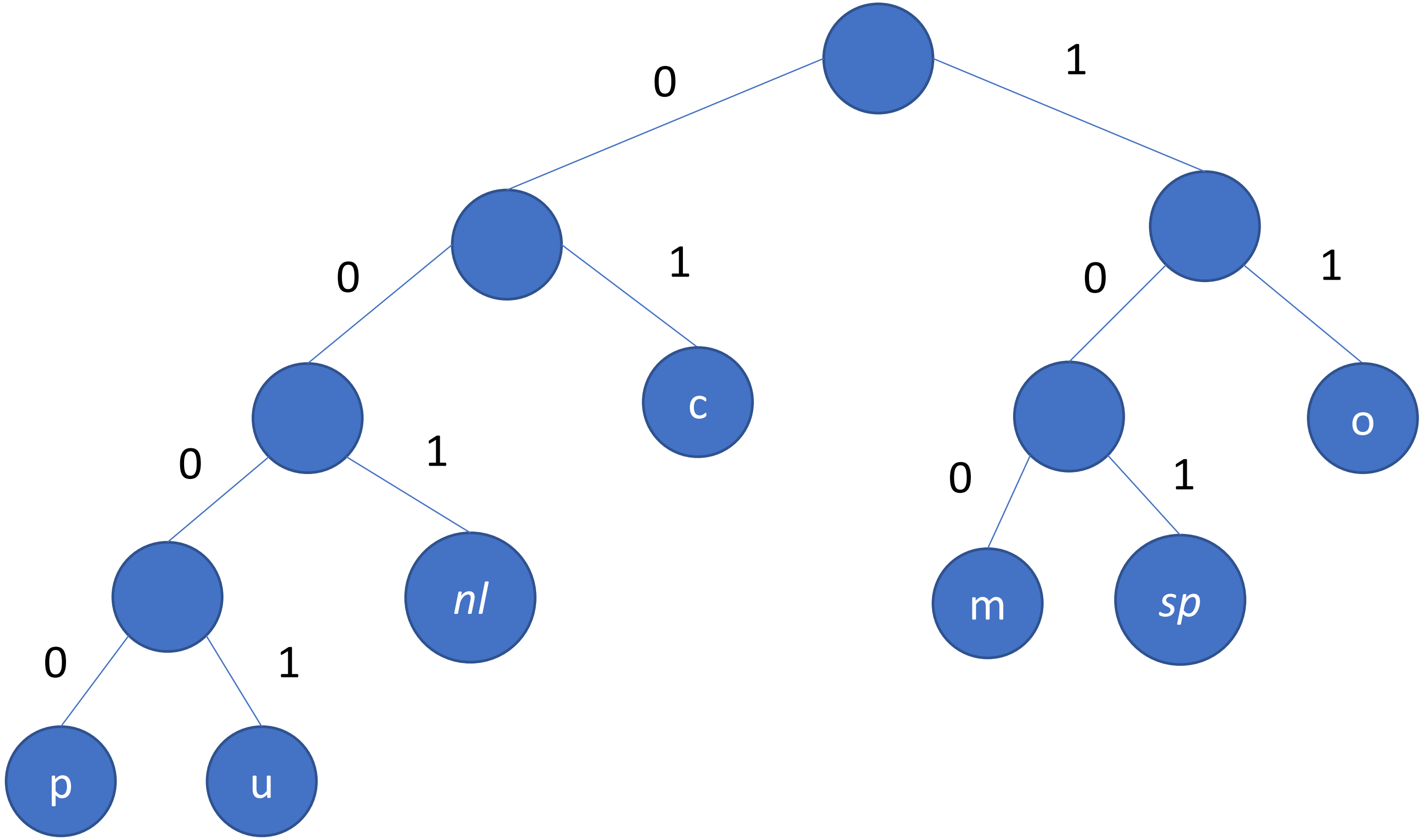
Objective: minimize $\sum_i d_i F[i]$

Example:

Character	Frequency
c	20
o	35
m	15
p	5
u	5
space	15
newline	10



Character	Frequency	Code	Total Bits
c	20	01	40
o	35	11	70
m	15	101	45
p	5	0000	20
u	5	0001	20
space	15	101	45
newline	10	001	30
			TOTAL: 270



Compare with previous encoding total of 315

Savings of $\approx 14\%$

Intuition: assign shorter codes to more frequent characters and longer codes to less frequent characters

Huffman's Algorithm

Greedily choose two least frequent characters

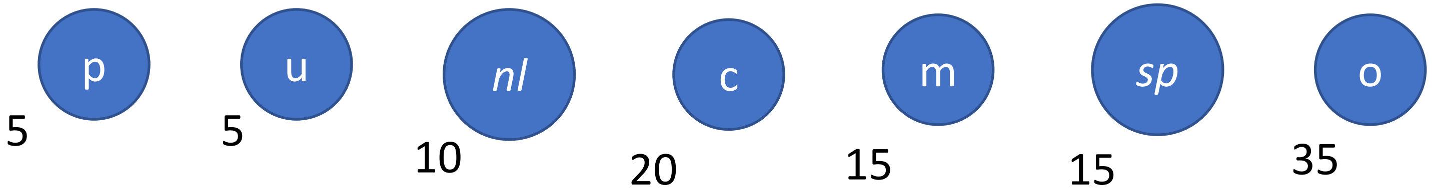
Make them leaves of a new node

Replace those two characters by a new virtual character with frequency equal to the sum of the frequencies of the two characters

Repeat

Character	Frequency
c	20
o	35
m	15
p	5
u	5
space	15
newline	10

Choose two lowest frequency characters

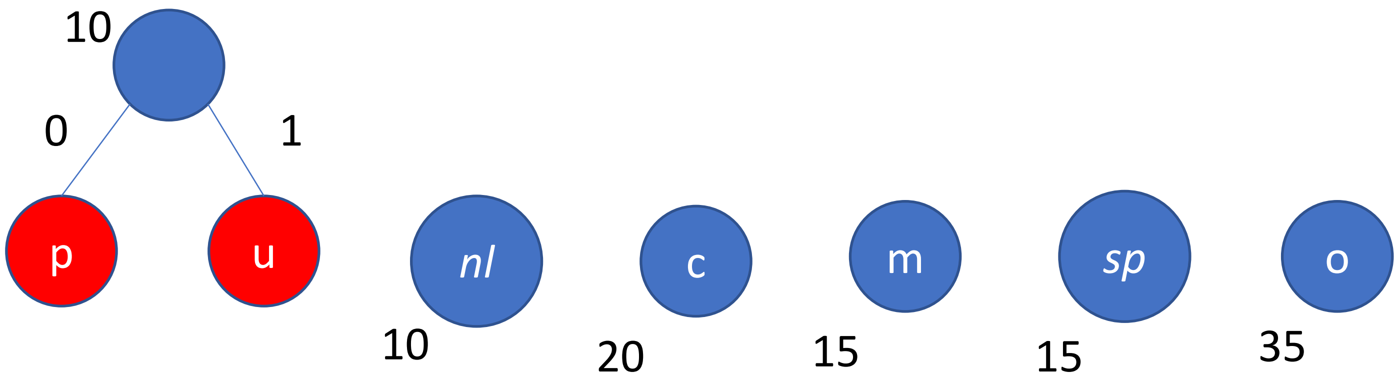


Character	Frequency
c	20
o	35
m	15
p	5
u	5
space	15
newline	10

Choose two lowest frequency characters

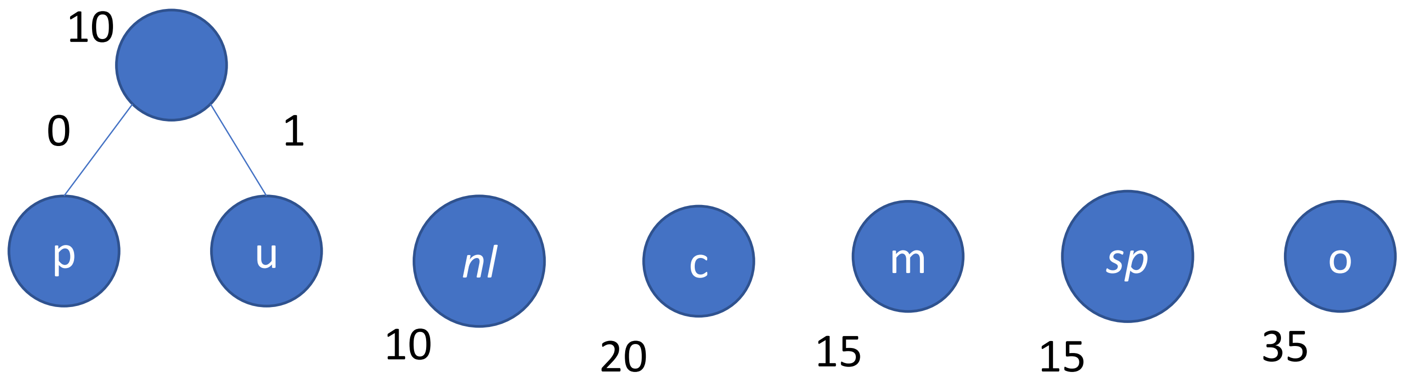
Merge them into a single character

Update frequency to be the sum

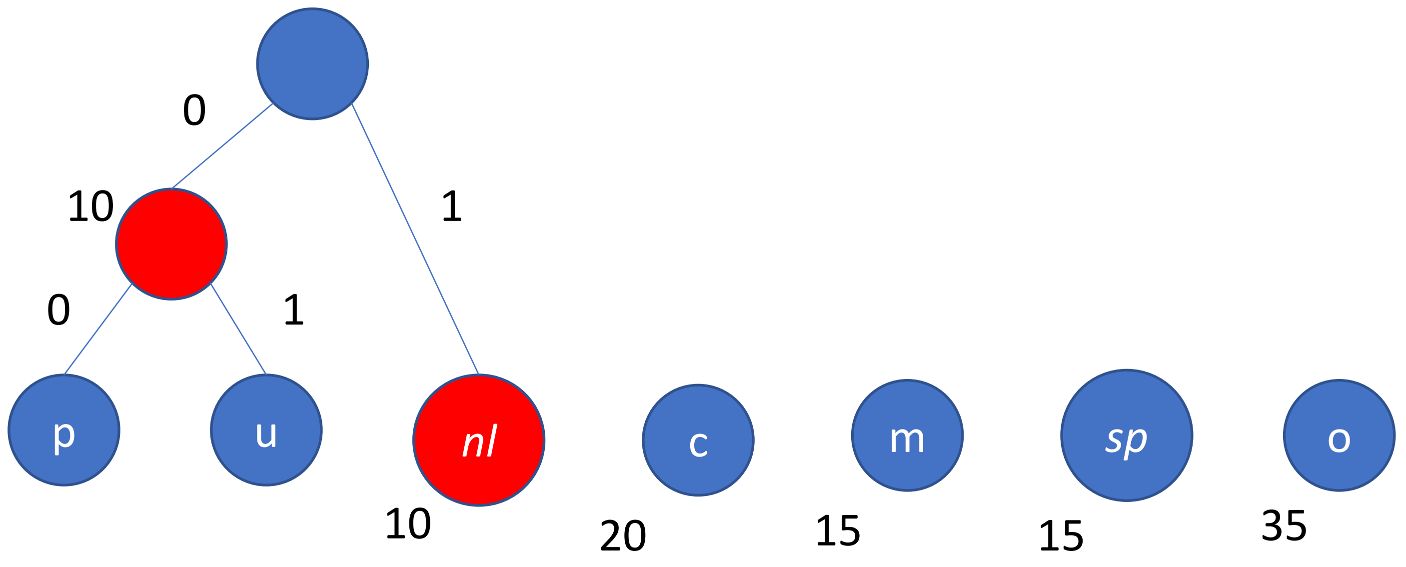


Character	Frequency
c	20
o	35
m	15
p	5
u	5
space	15
newline	10

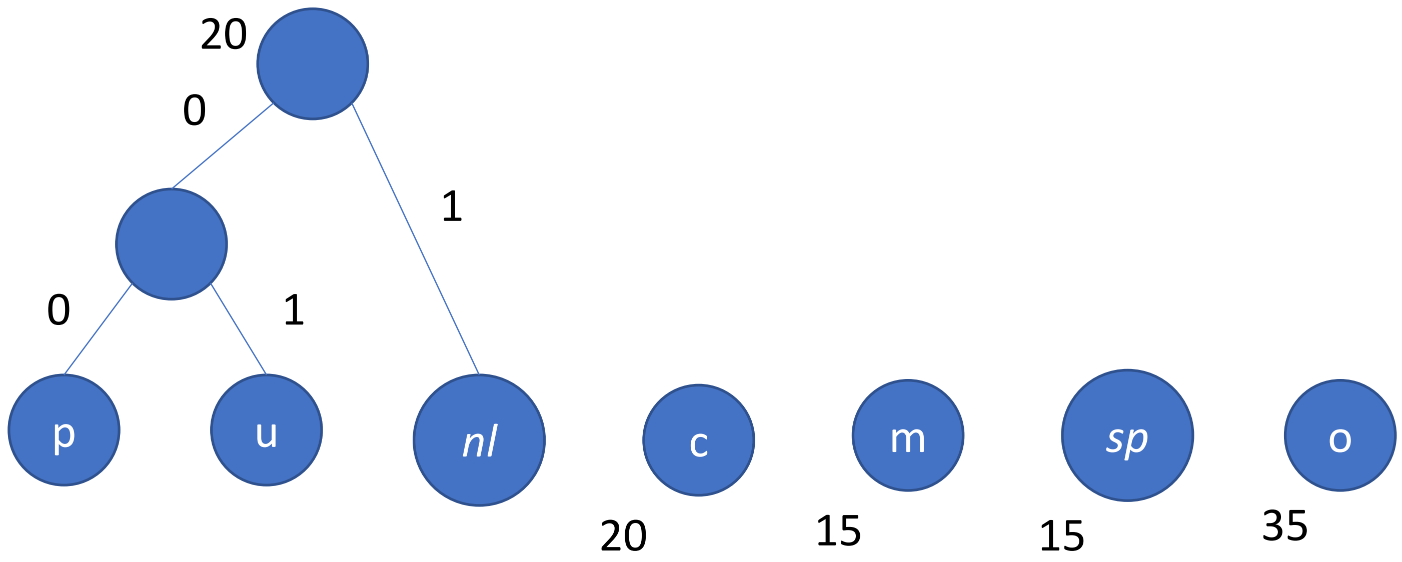
Repeat



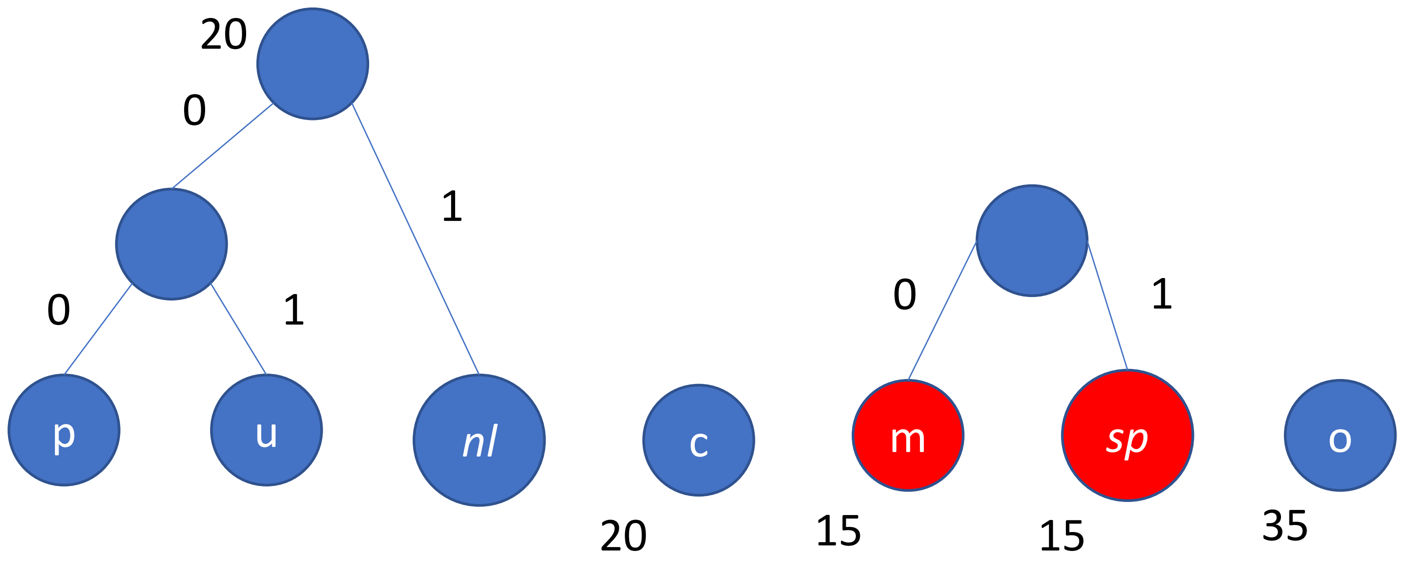
Character	Frequency
c	20
o	35
m	15
p	5
u	5
space	15
newline	10



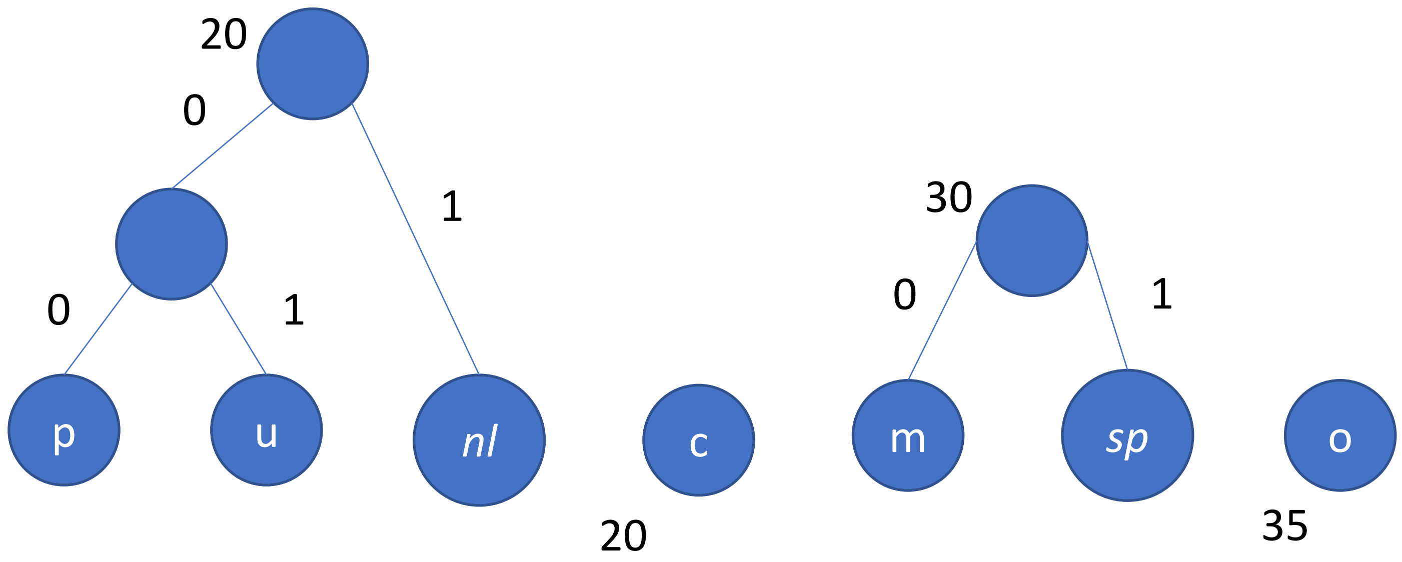
Character	Frequency
c	20
o	35
m	15
p	5
u	5
space	15
newline	10



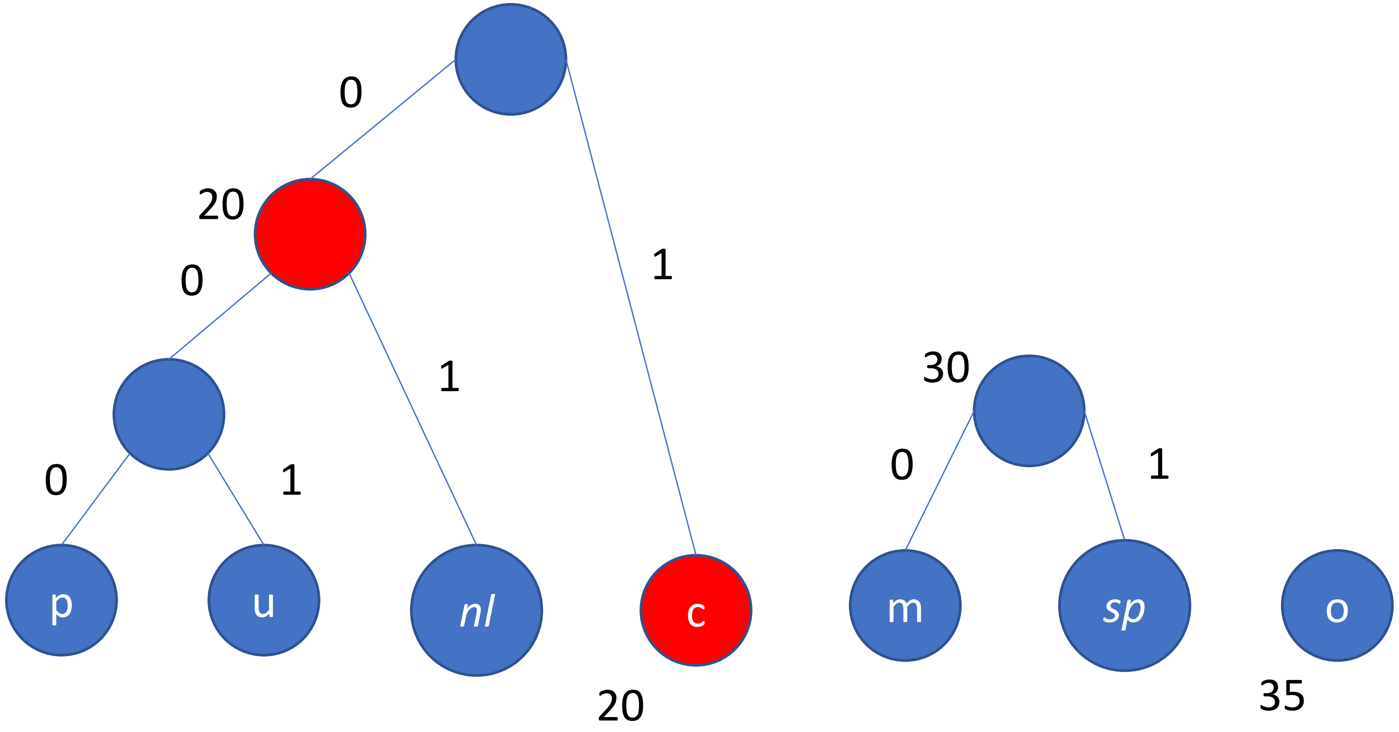
Character	Frequency
c	20
o	35
m	15
p	5
u	5
space	15
newline	10



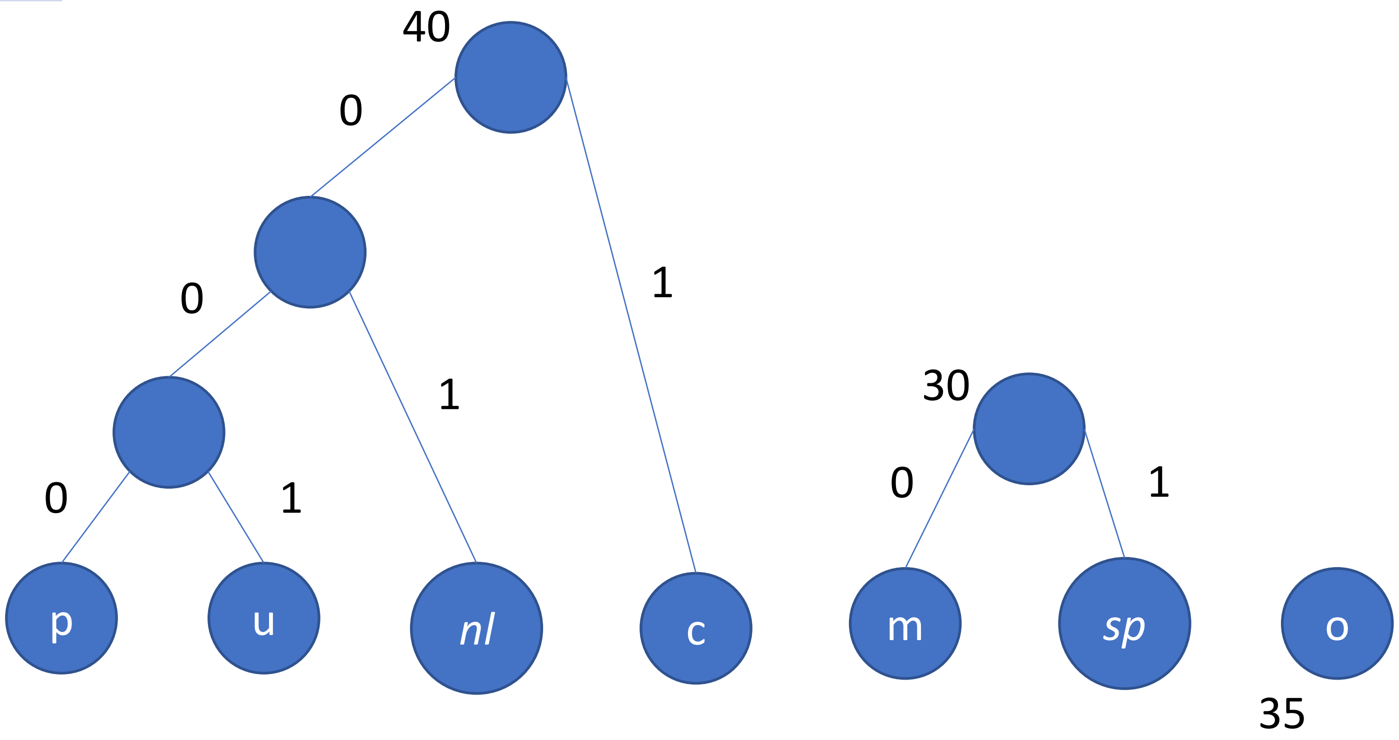
Character	Frequency
c	20
o	35
m	15
p	5
u	5
space	15
newline	10



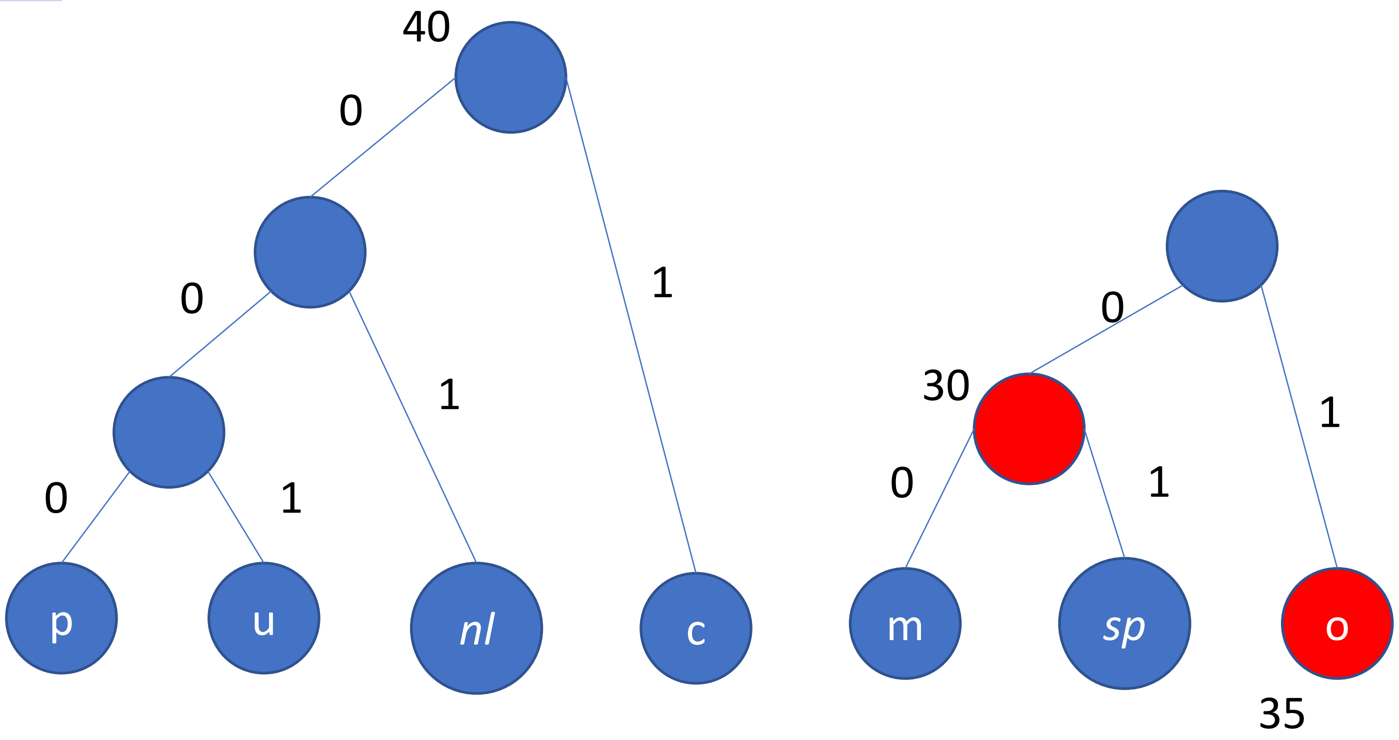
Character	Frequency
c	20
o	35
m	15
p	5
u	5
space	15
newline	10



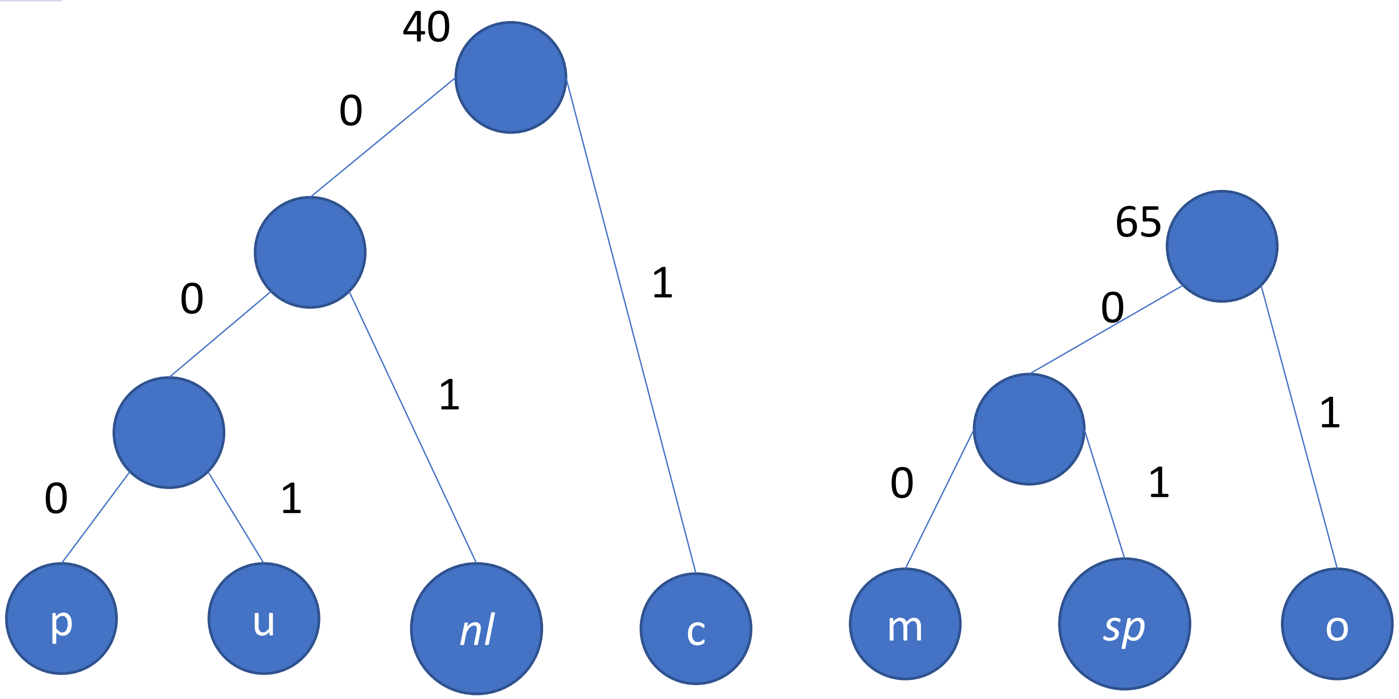
Character	Frequency
c	20
o	35
m	15
p	5
u	5
space	15
newline	10



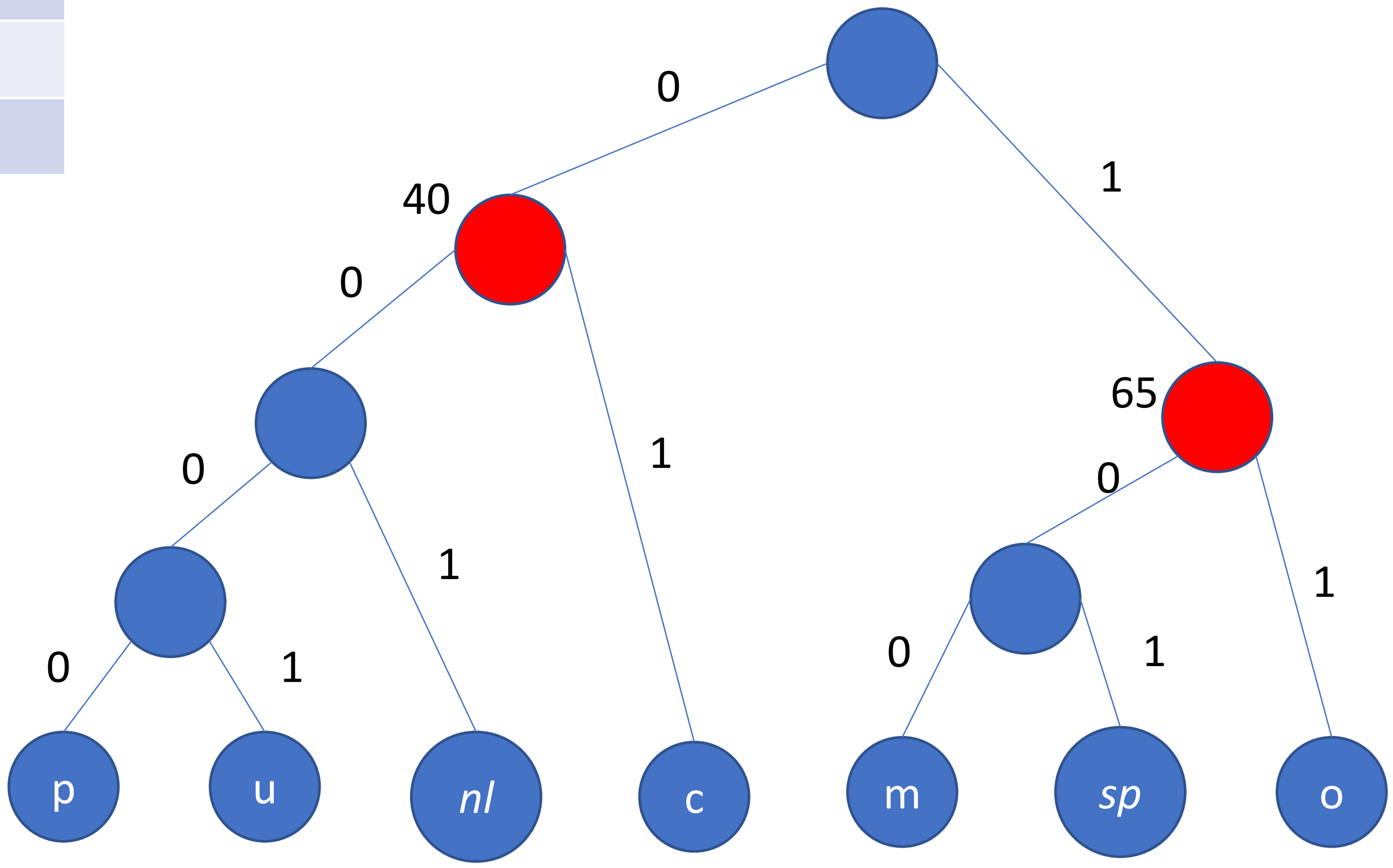
Character	Frequency
c	20
o	35
m	15
p	5
u	5
space	15
newline	10



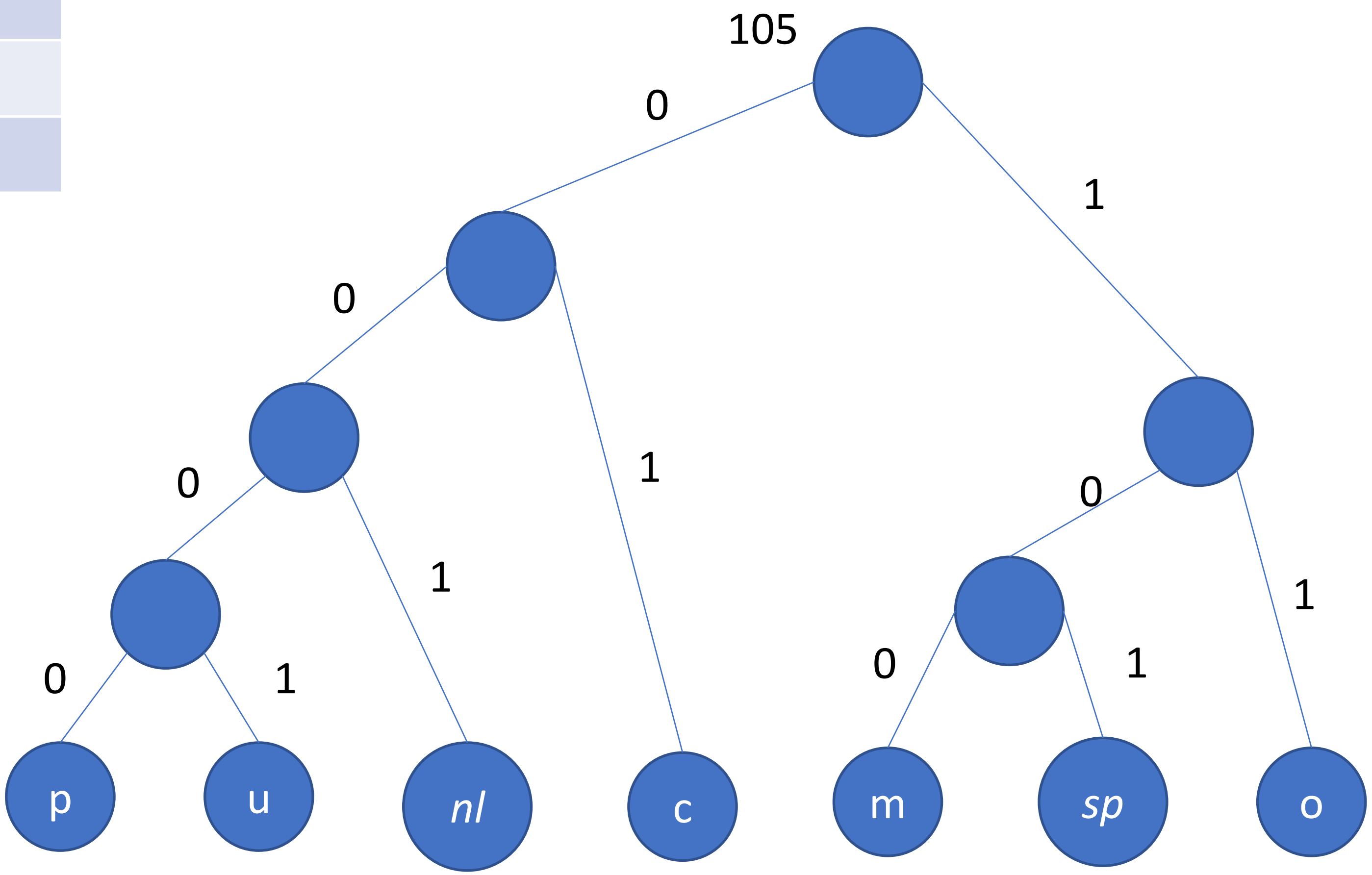
Character	Frequency
c	20
o	35
m	15
p	5
u	5
space	15
newline	10



Character	Frequency
c	20
o	35
m	15
p	5
u	5
space	15
newline	10



Character	Frequency
c	20
o	35
m	15
p	5
u	5
space	15
newline	10



Pseudocode

Overall, worst-case running time is $O(n \log n)$

$Huffman(F[1..n])$

Initialize min-priority queue Q to consist of elements i with priority field $i.freq = F[i]$

$\boldsymbol{for}\ i = 1\ \boldsymbol{to}\ n-1$

allocate a new node z

$z.left \leftarrow x \leftarrow ExtractMin(Q)$

$z.right \leftarrow y \leftarrow ExtractMin(Q)$

$z.freq \leftarrow x.freq + y.freq$

$Insert(Q, z)$

$\boldsymbol{return}\ ExtractMin(Q)$ // return the root of the tree

There exists priority queue implementation supporting operations $ExtractMin()$ and $Insert()$ in time $O(\log n)$

$Huffman$ performs $O(n)$ iterations and each iteration takes $O(\log n)$ with such priority queues.

Proof of correctness

Definition: $val(T) = \sum_k d_k f_k$ is the value of the objective achieved by T, where d_k is the depth of the kth character in T and f_k is its frequency.

Lemma. Let i and j be two characters with smallest frequencies. Then there is an optimal prefix tree in which these two letters are sibling leaves in the tree in the lowest level

Proof.

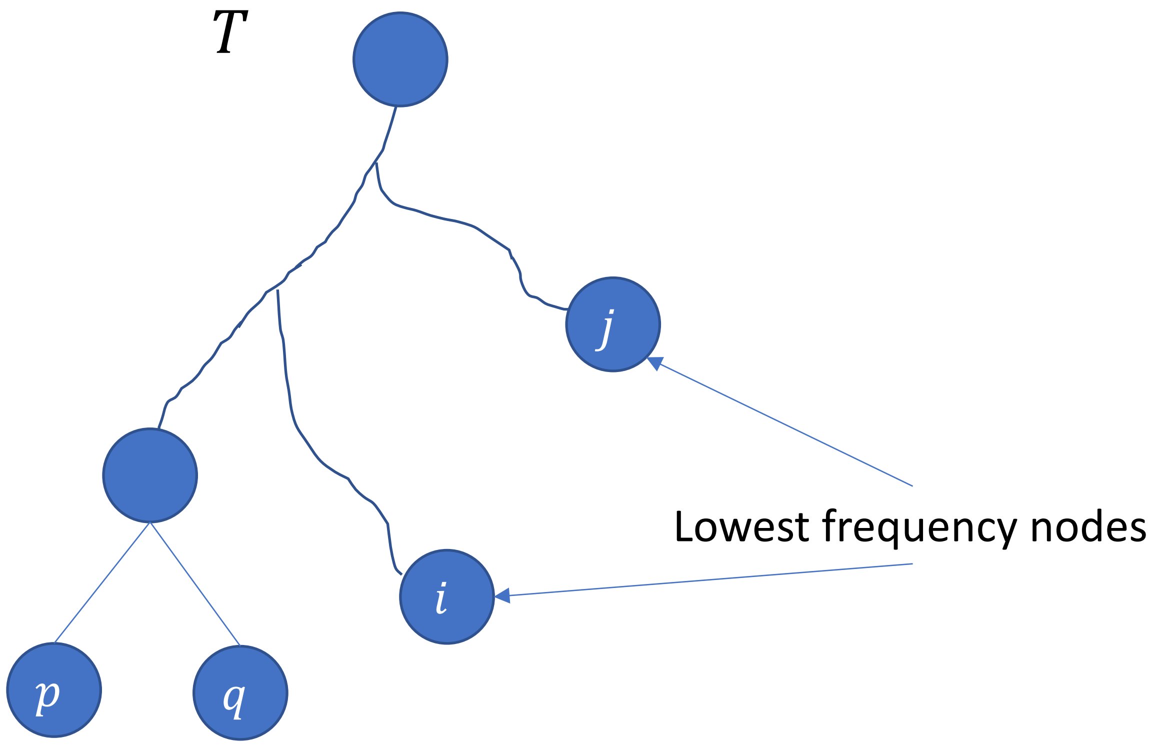
Let T be an optimal prefix tree. If it doesn't satisfy the conditions of the lemma, we show how to modify it to tree T' such that

- T' satisfies the conditions of the lemma, and
- $val(T') \leq val(T)$

… proof (continued)

We may assume T is *full*: every internal node has two children (next lemma).

Let p and q be two characters that are siblings and at lowest level.



Largest depth nodes that are siblings

Assume without loss of generality:

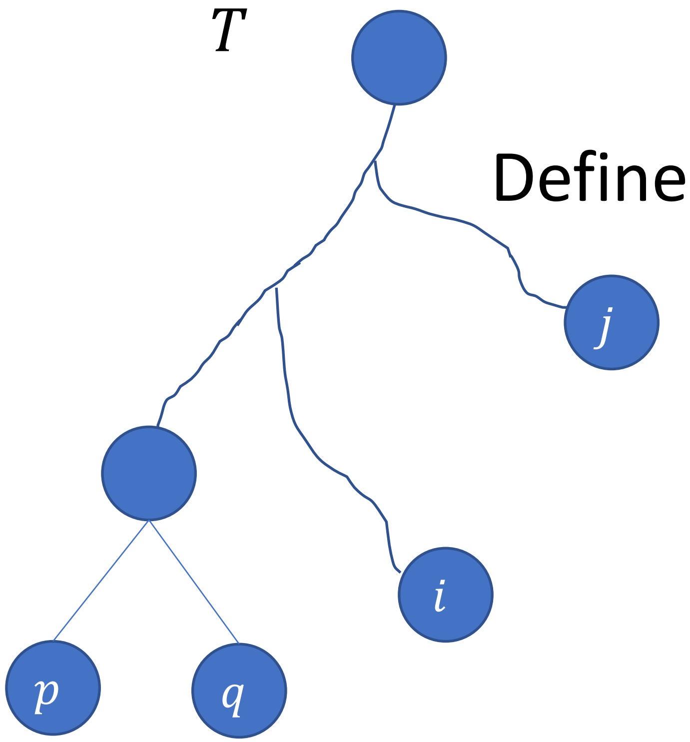
$$f_i \leq f_j$$
$$f_p \leq f_q$$

Observe that

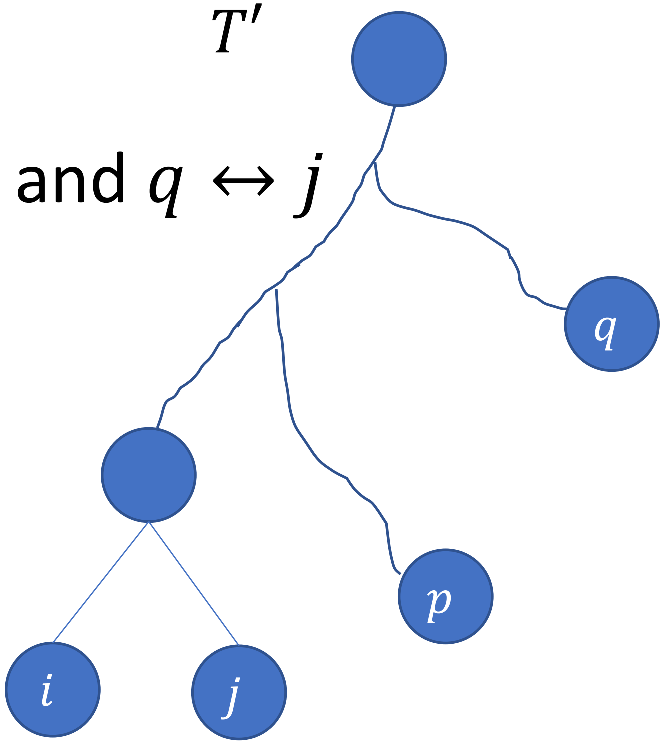
$$f_i \leq f_p$$
$$f_j \leq f_q$$

And

$$d_p, d_q \geq d_i$$
$$d_p, d_q \geq d_j$$



Define T' by swapping $p \leftrightarrow i$ and $q \leftrightarrow j$



$$val(T') = val(T) - d_p f_p - d_q f_q - d_i f_i - d_j f_j + d_p f_i + d_q f_j + d_i f_p + d_j f_q$$

$$= val(T) - d_p(f_p - f_i) - d_q(f_q - f_j) + d_i(f_p - f_i) + d_j(f_q - f_j)$$

$$= val(T) - \underbrace{(d_p - d_i)(f_p - f_i)}_{\geq 0} - \underbrace{(d_q - d_j)(f_q - f_j)}_{\geq 0} \leq val(T)$$

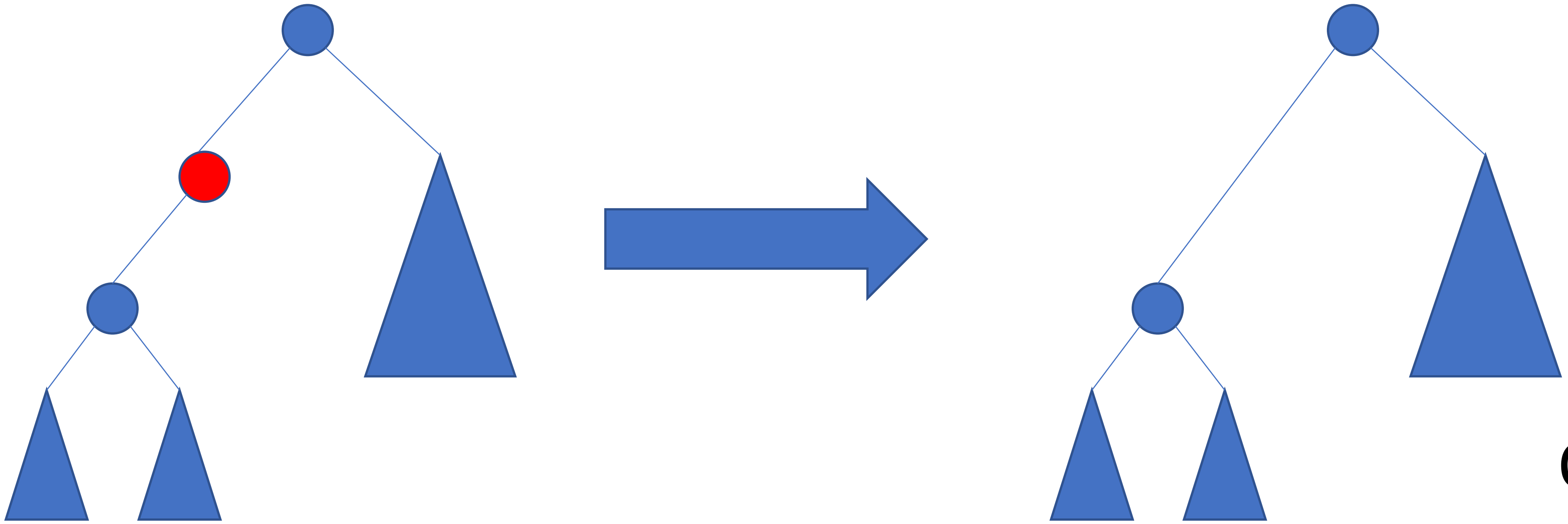
QED

Lemma. The tree for any optimal prefix code can be assumed to be full without loss of generality.

Proof

If some internal node has only one child then we can ”short-circuit” it by replacing the internal node with its unique child

This cannot increase the value of the objective $\sum d_k f_k$



QED

Theorem. Huffman's algorithm produces an optimal prefix tree.

Proof (by induction on n – number of characters)

Base case $n = 2$: obvious.

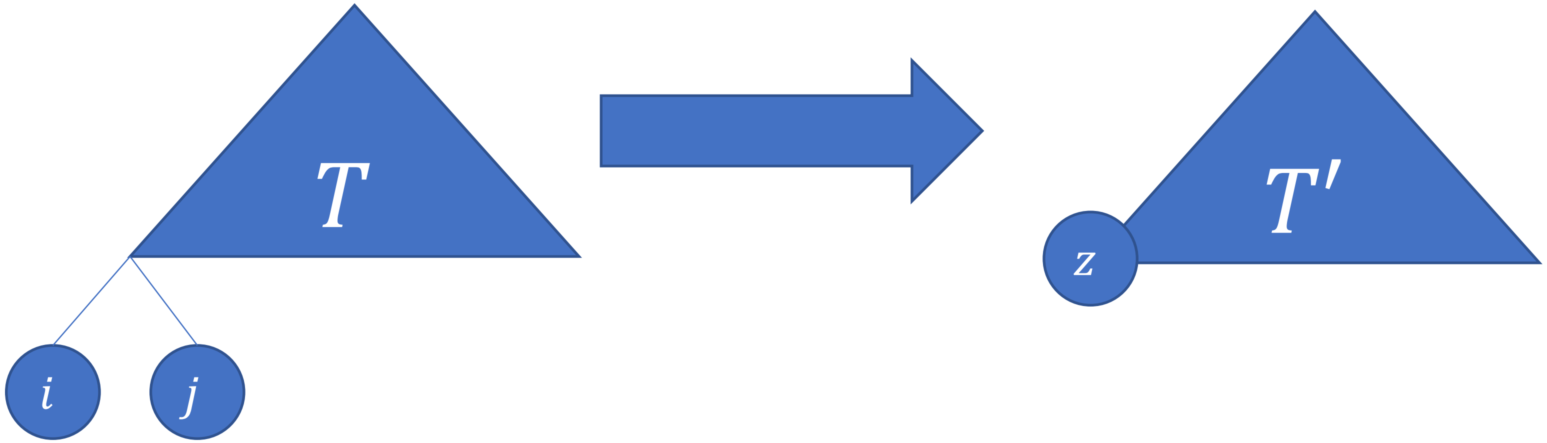
Inductive assumption: assume that Huffman's algorithm produces an optimal prefix tree for all inputs with at most n characters for some $n \geq 2$.

Inductive step: consider an input I with $n + 1$ characters with i and j being two least frequent characters.

Let T be an optimum tree for this input with i and j siblings at the lowest level (exists by the first lemma).

Define a new character z with associated frequency $f_z = f_i + f_j$

Let T' be the prefix tree for $I \cup \{z\} - \{i, j\}$ obtained by replacing parent of i, j with z



$$d_i = d_j = d_z + 1 \text{ and } f_z = f_i + f_j$$

Therefore

$$val(T') = val(T) - f_i d_i - f_j d_j + d_z f_z$$
$$= val(T) - (f_i + f_j)$$

By induction, Huffman's algorithm finds optimal tree H' for $I \cup \{z\} - \{i, j\}$

$$val(H') \leq val(T')$$

Also, by the algorithm's definition tree H for the original input I satisfies

$$val(H) = val(H') + f_i + f_j$$

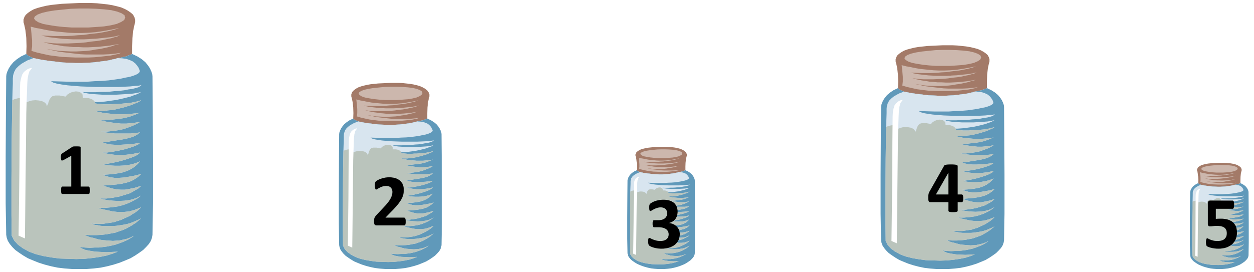
Combining everything we obtain:

$$
\begin{aligned}
val(H) = val(H') + f_i + f_j &\le val(T') + f_i + f_j \\
&= \left(val(T) - \left(f_i + f_j\right)\right) + f_i + f_j \\
&= val(T)
\end{aligned}
$$

Since T is optimal tree for the whole input I and $val(H) \le val(T)$ the tree output by Huffman's algorithm H is optimal too.

QED

Fractional Knapsack (CLRS 16.2)



Items:	1	2	3	4	5
Weight:	40 kg	25 kg	10 kg	35 kg	5 kg
Total Value:	$120	$125	$200	$70	$1000
Value per unit weight:	$3	$5	$10	$2	$200

“knapsack”
Capacity: 100 kg

Solution:
- 5 kg of 5
- 10 kg of 3
- 25 kg of 2
- 40 kg of 1
- 20 kg of 4

Empty 4 out a bit until it is 20 kg and then put it on donkey

Fractional Knapsack formally

Input: n items described by two parameters given as arrays of numbers:

$W[1..n]$ – where $W[i]$ is the weight of the ith item

$V[1..n]$ - where $V[i]$ is the total value of the ith item

C – capacity of the knapsack

Output: $X[1..n]$ – array of fractions such that

$X[i] \in [0, 1]$

$\sum_{i=1}^{n} X[i] \cdot W[i] \leq C$

$\sum_{i=1}^{n} X[i] \cdot V[i]$ is as large as possible

Knapsack notes

This problem is called **Fractional** Knapsack because items are divisible, i.e., we can take **fractions** $X[i] \in [0,1]$

In another version of the problem called $\mathbf{0-1}$ or **Integral** Knapsack items are indivisible, i.e., we can either take the whole item or not $X[i] \in \{0,1\}$

We will revisit Integral Knapsack in dynamic programming
Integral Knapsack is a harder problem and doesn't seem to have an efficient (strongly polynomial time) algorithm

Divisibility property of items makes the problem much easier!

Natural greedy algorithm

Indeed, since items are divisible it makes sense to start taking as much of an item that gives the bigger bang for the buck as possible

Sort items in non-increasing order of $V[i]/W[i]$ (value per unit weight)

Process items in this order and for each item:

> if the remaining capacity of the knapsack can accommodate the entire item, take the entire item
>
> otherwise take the largest portion of the item you can filling the knapsack to capacity

$FractionalKnapsack(W[1..n], V[1..n], C)$

initialize $X[1..n]$ to all zeros

initialize $I[1..n] = \{1,2, \dots, n\}$ to keep track of original indices

sort W, V, I simultaneously in non-increasing order of $V[i]/W[i]$

$for\ i = 1\ to\ n$

let $X\big[I[i]\big] \leftarrow \min(C, W[i])\ /W[i]$

$C \leftarrow C - X\big[I[i]\big] \cdot W[i]$

$return\ X$

The running time is dominated by the sorting procedure, so $O(n \log n)$

Correctness

For simplicity assume that the input is already sorted

$$\frac{V[1]}{W[1]} \geq \frac{V[2]}{W[2]} \geq \cdots \geq \frac{V[n]}{W[n]}$$

Let $X[1..i]$ denote the partial solution restricted to items $1..i$

If all items fit in the knapsack then our algorithm finds this solution and it is optimal. Therefore assume that $\sum_{i=1}^{n} W[i] > C$

Consider the following statement:

$P(i)$="Solution $X[1..i]$ can be extended to an optimal solution to the overall instance"

Proving $P(i)$ for all $i \in \{0, \ldots, n\}$ by induction on i resolves the correctness

Claim. Suppose $\sum_{i=1}^{n} W[i] > C$. For $i \in \{0, 1, \dots, n\}$ the following statement is true $P(i)$="Solution $X[1..i]$ can be extended to an optimal solution to the overall instance"

Proof (by induction on i)

Base case $i = 0$: obvious since $X[1..0]$ is empty.

Inductive assumption: assume $P(i)$ is true for some $i \geq 0$

Inductive step: let $X[1..i]$ be the solution to the first i items, OPT_i be its extension to optimal solution overall, and consider the $i + 1$st item

If $X[1..i]$ already exhausted the entire capacity C then $i + 1$ is not taken. Observe that the same happens in OPT_i so $X[1..i + 1]$ can be extended to optimal OPT_i

Otherwise, since greedy takes as much of item $i + 1$ as possible we have $OPT_i[i + 1] \leq X[i + 1]$

… **proof** (continued)

If $OPT_i[i+1] = X[i+1]$ then the claim follows since OPT_i can be considered an optimal extension of $X[1..i+1]$

Thus, suppose that $OPT_i[i+1] < X[i+1]$

Since total capacity is the same in OPT_i and X there must be some $j > i+1$ such that $OPT_i[j] > X[j]$

$\epsilon = \min\big((OPT_i[j] - X[j])W[j], (X[i+1] - OPT_i[i+1]) \cdot W[i+1]\big)$

We can transfer ϵ weight from item j to item i in OPT_i to get another feasible solution OPT'

Suppose that $\epsilon = (OPT_i[j] - X[j]) \cdot W[j]$. Then we have

$$\sum_k OPT'[k]\,V[k] = \sum_k OPT_i[k]V[k] - \epsilon \cdot \frac{V[j]}{W[j]} + \epsilon \cdot \frac{V[i+1]}{W[i+1]}$$

$$= \sum_k OPT_i[k]V[k] + \epsilon \cdot \left(\frac{V[i+1]}{W[i+1]} - \frac{V[j]}{W[j]}\right) \geq \sum_k OPT_i[k]V[k]$$

Similar calculation shows that OPT' achieves objective value no worse than OPT_i when $\epsilon = (X[i+1] - OPT_i[i+1]) \cdot W[i+1]$

Thus, in either case OPT' is also an optimal and feasible solution.

If $OPT'[i+1] = X[i+1]$, we are done, otherwise we can apply the same argument to OPT' and repeat.

This finishes the proof by induction and the proof of correctness.

QED

Satisfiability of Horn Formulas

A specific framework for basic logical reasoning – expressing logical facts and deriving conclusions.

Boolean variables $x_1, x_2, \ldots, x_n$ take on values from binary alphabet $\{0,1\}$

They are formal variables, but might have some meaning associated, e.g.

$x =$ Cookie-Monster runs

$y =$ Cookie-Monster eats vegetables

$z =$ Cookie-Monster does yoga

$w =$ Cookie-Monster eats cookies

A **literal** is either a variable (e.g., x) or its negation ($\bar{x}$)

Horn formulas consist of two kinds of clauses:

(1) Implications

Left-hand side is an AND of any number of positive literals and write-hand side is a single positive literal.

("If conditions on the left hold then the conclusion on the right also holds")

Example: $x \wedge y \wedge z \Rightarrow w$

"If Cookie-Monster runs, eats vegetables, and does yoga then Cookie-Monster eats cookies"

Note: left-hand side could be empty (means right-hand side must be true)

$$\Rightarrow w$$

(2) **Pure negative clauses**

OR of any number of negative literals

(“all statements can’t be true”)

$$\bar{x} \lor \bar{y} \lor \bar{w}$$

Given a Horn formula the goal is to find an assignment of true/false (or $1/0$) to the variables that satisfies all clauses, if such an assignment exists.

Such an assignment is called a satisfying assignment.

Satisfiability of Horn Formulas problem

Input: Horn formula ϕ with m clauses and n variables

Output: a satisfying assignment, if one exists

Example:

$$\bar{x} \vee \bar{y}\ ,\ \Rightarrow x\ ,\ \Rightarrow z\ ,\ (x \wedge z) \Rightarrow w\ ,\ \bar{y} \vee \bar{w}\ ,\ \Rightarrow w$$

Satisfying assignment $\tau(x) = \tau(z) = \tau(w) = 1, \tau(y) = 0$

Greedy algorithm for Satisfiability of Horn Formulas

(1) Initially, set all variables to false

(2) While there is an implication that is not satisfied:
    set the right-hand side of the implication to true

(3) If all purely negative clauses are satisfied:
    return the assignment
  Else:
    return “Formula is not satisfiable”

Trivial implementation: $O(n \cdot m)$ running time, but can be done in $O(n + m)$ with more careful implementation

Example

$$\bar{x} \vee \bar{y}\ ,\ \Rightarrow x\ ,\ \Rightarrow z\ ,\ (x \wedge z) \Rightarrow w\ ,\ \bar{y} \vee \bar{w}\ ,\ \Rightarrow w$$

Initially, set all variables to false:

$$\tau(x) \leftarrow false, \tau(y) \leftarrow false, \tau(z) \leftarrow false, \tau(w) \leftarrow false$$

While implications are unsat:

$\Rightarrow x$ is unsat: update $\tau(x) \leftarrow true$

$\Rightarrow z$ is unsat: update $\tau(z) \leftarrow true$

$(x \wedge z) \Rightarrow w$ became unsat: update $\tau(w) \leftarrow true$

Check purely negative clauses: $\bar{x} \vee \bar{y}$ and $\bar{y} \vee \bar{w}$ are both sat by $\tau(y) = false$

Return: $\tau(x) = true, \tau(y) = false, \tau(z) = true, \tau(w) = true$

Correctness of the greedy algorithm

Lemma. If a certain variable is set to true by the greedy algorithm then it must be true in any satisfying assignment to the formula

Proof:

The claim follows by a simple induction on the number of iterations of the main while loop

Base case: prior to the first iteration of the loop all variables are set to false.

The only implication clauses that are violated are of the form $\Rightarrow x$

The degenerate implications of the form $\Rightarrow x$ can only be satisfied by $\tau(x) = true$, so x must be set to true in any satisfying assignment

… proof (continued)

Inductive assumption: assume that the claim is true for all variables set by the kth iteration of the main while loop, for some $k \geq 0$

Inductive step: suppose that in the $(k+1)$st iteration, variable x is set to true as a result of the algorithm trying to satisfy the implication

$$(x_1 \wedge \cdots \wedge x_\ell) \Rightarrow x \text{ (*)}$$

Let τ be an arbitrary satisfying assignment.
Since this implication was chosen in the algorithm, variables on the left $x_1, \ldots, x_\ell$ must have been set by the algorithm to true in prior iterations

By induction, $\tau(x_1) = \tau(x_2) = \cdots = \tau(x_\ell) = true$

Since τ is satisfying, it must satisfy clause (*), so $\tau(x) = true$

Since τ is arbitrary, the claim follows.

QED

Finishing the argument of correctness

Since the algorithm does an explicit check to see if an assignment is satisfying, it can return only satisfying assignments

Need to argue that if the algorithm doesn't return an assignment then there is no sat assignment

Suppose for contradiction, the algorithm doesn't return an assignment but there is a satisfying assignment τ

Previous lemma implies that all variables set by algorithm to true are also set to true in τ

But the explicit check finds a violated pure clause by these variables being set to true

Therefore τ cannot be satisfying.

You should now be able to…

- Describe the structure of combinatorial optimization problems.
- Explain the greedy/myopic strategy to solving combinatorial optimization problems.
- Discuss benefits/drawbacks of greedy/myopic strategies
- Come up with and analyze correctness/runtime of various greedy/myopic strategies for Interval Scheduling, Huffman Coding, Fractional Knapsack, and Satisfiability of Horn Formulas problems.

Review Questions

- Formally write down the basic structure of Huffman Coding problem as a combinatorial optimization problem

- Same as the previous question for Fractional Knapsack and Satisfiability of Horn Formulas

- Provide more detailed pseudocode for the greedy algorithm for Satisfiability of Horn Formulas. Can you make the algorithm run in linear time, i.e., $O(m+n)$?