CSCI 4470 Algorithms

Part IV Advanced Design and Analysis Techniques

- 14 Dynamic Programming
- **15 Greedy Algorithms**
- 16 Amortized Analysis

Chapter 15: Greedy Algorithms

```
15 Greedy Algorithms
  15.1 An Activity-Selection Problem
  15.2 Elements of the Greedy Strategy
  15.3 Huffman Codes
  15.4 Offline Caching
```

Three Methods of Solving Optimization Problem

1. Greedy Methods (Algorithms)
2. Dynamic Programming
3. Branch and Bound

Greedy Algorithms

Greedy Algorithms aim to find the optimal solution for optimization problems by making the best choice at each step.

Notes: For a greedy algorithm to be effective, it must satisfy two primary conditions which are elements of greedy strategy:

Elements of Greedy Strategy

1. Optimal Substructure:

- The problem's optimal solution should encompass optimal solutions to its sub-problems.

2. Greedy Choice Property:

- A global optimum is achievable by consistently choosing a local optimum.

- It means that the choice made by the greedy choice property and that choice alone dictates the choices to be made in the subsequent steps.

3. Feasibility Check:

- Before making a choice, check if it's feasible with respect to the problem constraints.

4. Solution Building:

- Once a choice is made, the problem is typically reduced to a subproblem, and the process is repeated.

5. Iterative Approach:

- Greedy algorithms often work in an iterative manner, making one greedy choice after another until a solution is reached.

6. Immediate Decision:

- Greedy algorithms make decisions based on the current information without worrying about the consequences of the decision.

7. Additional:

- In essence, greedy strategy involves making the best local choices at each step, ensuring feasibility, and iteratively building the solution. However, it's crucial to remember that while greedy approaches can be highly efficient, they don't guarantee the optimal solution for every problem.
- Before applying a greedy algorithm, it's essential to validate these properties to ensure the derived solution is optimal.

Greedy-choice Property

- An optimal solution can be found by making locally optimal (greedy) choices
- The best current choice will lead to the best overall solution
- This property does not apply to the dynamic programming problems

Optimal substructure

- Optimal solutions to the problem contain optimal solutions to sub-problems
- **If A is optimal for X and $s_i \in A$, then $A - s_i$ is optimal for $X - s_1$**
- This also applies when we use dynamic programming

1. If A is optimal for X: This means that we have a solution, denoted as A, which is considered the best or optimal for the original problem, denoted as X.
2. and $s_i \in A$: This part indicates that within the optimal solution A, there exists an element s_i which is one of the items or components included in the solution.

3. then $A - s_i$ is optimal for $X - s_1$: This means that if we remove the element s_i from the optimal solution A, the resulting solution, denoted as $A - s_i$, is still considered the best or optimal. Moreover, the problem we're solving is no longer the original problem X, but a modified problem $X - s_1$ where s_1 is a different element.

0-1 KNAPSACK VS. FRACTIONAL KNAPSACK

Which problem exhibits greedy choice property?

- Only the fractional knapsack problem exhibits the greedy choice property.

Which one exhibits optimal-substructure property?

- Both the 0-1 knapsack problem and the fractional knapsack problem exhibit the optimal substructure property.

PROPERTIES OF GREEDY ALGORITHMS

Candidate set:

- Solution is chosen from this set

Selection function:

- Used to choose best candidate to add to solution

Feasibility function:

- Determines if candidate can contribute to solution

Objective function:

- Assigns a value to solution (or partial solution)

Solution function:

- Indicates when complete solution is discovered

GREEDY ALGORITHM STRUCTURE

- The selection function is linked to the objective function.
- They may be the same, but often, there are multiple valid choices.
- At each step, choose the best candidate without concern for the future.
- Once a candidate is added to the solution, it remains there permanently.
- Once a candidate is rejected, it's never reconsidered.
- Greedy algorithms do NOT always guarantee optimal solutions.

- They work optimally for many problems, but it's crucial to ensure that the greedy-choice and optimal substructure properties hold.

Fractional Knapsack Problem

Given: A knapsack with capacity W and n items, each with value v_i and weight w_i. Items can be fractioned to maximize value without exceeding W.

Objective:

1. Maximize the knapsack's total value.
2. Ensure the total weight is ≤ W

Strategy: Choose items by **value-to-weight ratio,** $\frac{v_i}{w_i}$. If an item can't fit, fraction it.

- $\frac{V_1}{W_1} = \frac{72}{6} = 12$, $\frac{V_2}{W_2} = \frac{90}{9} = 10$, $\frac{V_3}{W_3} = \frac{15}{5} = 3$

Example:
Given a knapsack with capacity $W = 14$ and three items:

i	1	2	3
v_i	72	90	15
w_i	6	9	5
$\frac{v_i}{w_i}$	12	10	3

Solution:

Calculate the value-to-weight ratio for each item:

$\frac{V_1}{W_1} = \frac{72}{6} = 12$, $\frac{V_2}{W_2} = \frac{90}{9} = 10$, $\frac{V_3}{W_3} = \frac{15}{5} = 3$

Choose items with the highest ratios. For this example, the total value is $72 + (10 \times 8) = 152$.

Example Breakdown:

1. Initial State:

- Knapsack capacity: $W = 14$.

2. First Item (Ratio = 12):

- Added entirely: Weight = 6, Value = 72.

3. Remaining Capacity:

- $14 - 6 = 8$. $14 - 6 = 8$。

4. Second Item (Ratio = 10):

- Partially added: Weight = 8, Value = $10 \times 8 = 80$.

5. Total Value:

- $72 + 80 = 152$. $72 + 80 = 152$。

Significance:

The value-to-weight ratio ensures the greedy algorithm selects items that provide the most value for the least weight, leading to an optimal solution.

Fractional Knapsack Problem (Video)

In the Fractional Knapsack Problem, you are given a knapsack with a maximum weight capacity $W = 15$ and $n = 7$ items, each with a value v_i and weight w_i. Unlike the 0/1 knapsack problem, items may be broken into smaller pieces, allowing you to maximize the total value in the knapsack.

Objective:

1. Maximize the total value in the knapsack.
2. Ensure the total weight doesn't exceed W.
3. Select items by using value-to-weight ratio $\frac{v_i}{w_i}$

Items i	1	2	3	4	5	6	7
Values v_i	10	5	15	7	6	18	3
Weights w_i	2	3	5	7	1	4	1
$\frac{v_i}{w_i}$	5	1.67	3	1	6	4.5	3
x_i	x_1	x_2	x_3	x_4	x_5	x_6	x_7
	1	$\frac{2}{3}$	1	0	1	1	1

The Explanations of $\sum x_i w_i$ and $\sum x_i v_i$:

- Total knapsack capacity is 15, The highest ratio is item 5, which weight is 1: $15 - 1 = 14$, The second highest ratio is item 1, which weigh is 2: $14 - 2 = 12$, ... $12 - 4 = 8$, $8 - 5 = 3$,

1.$\sum x_i w_i$:

- This represents the total weight of the items selected. For each item i, x_i is the fraction of the item chosen (either 0, 1, or any fraction in between for the fractional knapsack), and w_i is the weight of that item. The summation calculates the combined weight of all selected items.

$$\sum x_i w_i = 1 \times 2 + \frac{2}{3} \times 3 + 1 \times 5 + 0 \times 7 + 1 \times 1 + 1 \times 4 + 1 \times 1$$

$$\sum x_i w_i = 2 + 2 + 5 + 0 + 1 + 4 + 1 = 15$$

- **Constraint: $\sum x_i w_i \leq W$ is satisfied**

2. $\sum x_i v_i$:

- This represents the total value of the items selected. For each item i, x_i is the fraction of the item chosen, and v_i is the value of that item. The summation calculates the combined value of all selected items.

$$\sum x_i v_i = 1 \times 10 + \frac{2}{3} \times 5 + 1 \times 15 + 1 \times 6 + 1 \times 18 + 1 \times 3$$
$$\sum x_i v_i = 10 + 3.33 + 15 + 6 + 18 + 3 = 55.33$$

- **Constraint: max$\sum x_i v_i$ is maximized**

Items Unsplittable

In the knapsack problem, items can either be split (fractional knapsack) or remain whole (0-1 knapsack). This section discusses the 0-1 variant.

What if Items are "Unsplittable":

- Selecting items by value-to-weight ratio isn't always optimal since items can't be fractioned.
- If items cannot be split, then the strategy of selecting items based on their value-to-weight ratio might not yield the optimal solution. This is because you can't take a fraction of an item; you either take the whole item or none of it.

Example:

Given three items and a knapsack with capacity $W = 14$:

i	1	2	3
v_i	72	90	15
w_i	6	9	5

Optimal Solution:

- The best combination of items for the 0-1 knapsack problem (where items are unsplittable) is Item 2 and Item 3. This gives a total value of $90 + 15 = 105$.

Greedy Strategy Result:

- If we were to use the greedy strategy (based on value-to-weight ratio), we would select Item 1 and Item 3. This gives a total value of $72 + 15 = 87$, which is not the optimal solution.

Pseudocode of FRACTIONAL-KNAPSACK

```
FRACTIONAL-KNAPSACK(v, w, W)
1.  // v and w contain value/weight if n items
2. Index items so that (v_i)/(w_i) ≥ v_{1+1}/w_{i+1} //Sort order is value per weight
3. i = 1
4. load = 0
5. S = Array[1 ... n] of 0's
6. while load < W and i ≤ n
7.     if w_i ≤ W - load
8.         take all of item i
9          load = load + w_i;
10            S[i] = 1
11.     else
12.         load = W; S[i] = (W-load)/w_i
13.     i = i + 1
14. return S
```

Explanation:

1. Line 1: The function `FRACTIONAL-KNAPSACK(v, w, W)` is defined, where v and w are arrays containing the values and weights of n items, and W is the maximum weight the knapsack can hold.
2. Line 2: Items are indexed in a way that the ratio of value to weight $\frac{v_i}{w_i}$ is in non-increasing order. This means items with higher value-to-weight ratios are considered first.
3. Line 3-5: Initialize `i` to 1, `load` to 0, and `S` as an array of zeros. `i` is the current item being considered, `load` is the current weight in the knapsack, and `S` will store the fraction of each item that is taken.
4. Line 6: A while loop runs as long as `load` is less than `W` and `i` is less than or equal to `n`.
5. Line 7-9: If the weight of the current item w_i is less than or equal to the remaining capacity in the knapsack `(W - load)`, take all of the item `i`. Add w_i to `load` and set `S[i]` to 1.
6. **Line 10-11**: If the current item w_i does not fit entirely, take only the fraction needed to fill the knapsack. Set `load` to `W` and `S[i]` to $\frac{W-load}{w_i}$.
7. Line 12: Increment `i` to move to the next item.
8. Line 13: Return the array `S`, which contains the fraction of each item that is taken.

Time Complexity:

- The dominant step is sorting the items based on the value-to-weight ratio, which takes $O(n \log n)$ time.
- If the items are already sorted, the time complexity is $O(n)$.

GREEDY PROPERTIES FOR FRACTIONAL KNAPSACK

Candidate set:

- The items themselves are the candidate set.
- S = {item1, item2, item3, …}

Selection function:

- The selection function is to choose the item with the maximum value-to-weight ratio, $\frac{v_i}{w_i}$.

Feasibility function:

- The feasibility is to ensure that the total weight of the selected items is less than or equal to the knapsack's capacity, i.e., total weight ≤ W.

Objective function:

- The objective function is to maximize the sum of the product of the fraction of each item (x_i) and its weight (w_i) in the solution, expressed as $max \sum_{i=1}^{n} x_i w_i$.

Solution function:

- The solution function indicates that the weight W has been reached or selected, meaning the knapsack is full.

Explanation:

These properties outline the key aspects of the greedy approach for the fractional knapsack problem. It defines how items are selected based on their value-to-weight ratio and how the feasibility and objective functions ensure that a solution is both valid and optimal. Finally, the solution function specifies when the knapsack is full and the algorithm stops.

STEP-BY-STEP GREEDY APPROACH:

1. Let C be the remaining candidates, and let S be the chosen candidates.

- Initially, $S = \emptyset$ (empty).

2. Iteratively find the best available candidate $x \in C$:

- Use the selection function:
  - If $S \cup \{x\}$ is feasible, then update S to $S \cup \{x\}$.
  - Remove x from C.

• If $S \cup \{x\}$ is feasible, then $S = S \cup \{x\}$
• $C = C \{x\}$

3. Each time S is enlarged, check if S constitutes a solution to the ˜/problem.

- The first solution found in this manner is always optimal.

```
GENERIC-GREEDY(C)    // C is the set of all candidates
1. S = ∅       // S is the solution under construction
2. while not solution(S) and C ≠ O
3.    x = an element of C maximizing select(x)
4.    C ← C \ {x}
5. if feasible(x)
6.       S = S ∪ {x}
7. if solution(S)
8. return S
9. else return "no feasible solution"
```

EXAMPLES OF GREEDY ALGORITHMS

Scheduling:

- Activity Selection
- Minimizing time in the system
- Earliest Deadline First (EDF) scheduling algorithm

Graph Algorithms:

- Minimum Spanning Trees
- Dijkstra's (shortest path) Algorithm Dijkstra

Greedy Heuristics:

- Not optimal, but provably close approximations
- Graph coloring
- Traveling Salesman
- Set covering

EXAMPLE: MINIMIZING TIME IN THE SYSTEM

1. Setup:

- In this example, we have a single server, such as a processor, gas pump, or cashier, and **a queue of n customers**, denoted as $c_1, c_2, \ldots, c_n$.
- $n = \{c_1, c_2, \ldots, c_n\}$

2. Selection Criteria:

$\min(t_i)$ **and** $j \in \text{candidates}$

- The system aims to minimize the total system time, and to achieve this, it selects the customer with the minimum service time ($min(t_i)$) from the set of possible candidates ($j \in \text{candidates}$).

This **Greedy Property** ensures that the system always processes the customer with the shortest service time first, which in turn helps in minimizing the total system time.

$min(t_i)$ **and** $j \in \text{candidates}$

- $min(t_i)$ suggests that we are looking for the minimum service time among all customers.
- $j \in \text{candidates}$ implies that j is an index from a set of possible candidates. In this example, the candidates refer to the set of all 3 customers: $\{c_1, c_2, c_3\}$.

Assumption about m customers

- Assume m customers have already been served, and we are currently considering the service time for the j^{th} customer, t_j, to determine its position in the sequence.

3. Processing Time

- **The processing time is given by the equation** $\sum_{k=1}^{m} t_{ik} + t_j$**.**
- t_{ik} is described as a constant. This might represent the service time for the k^{th} customer in the first m customers. The summation $\sum_{k=1}^{m} t_{ik}$ then represents the total service time for the first m customers.
- t_j is the processing time of the j^{th} customer.
- Therefore, the entire expression $\sum_{k=1}^{m} t_{ik} + t_j$ represents the total processing time for the first m customers plus the processing time for the j^{th} customer.

4. Service Time:

- Each **customer** c_i requires a certain amount of **time** t_i, to be served.

5. Objective:

- **Minimize the total system time** T**,** which includes both the wait time in the queue and the service time for all customers.
- Minimize $T =$ total system time for all customers (both wait time and service time)

FOR THREE CUSTOMERS in the EXAMPLE

To find the total system time for different sequences, we sum up the cumulative wait and service times for each customer in the sequence.

1. Given 3 customers with the following service times:

- $t_1 = 5, t_2 = 10, t_3 = 3$

2. Service Sequences S_i:

Calculate the total time for each possible sequence of serving the three customers.

For example, for the sequence (t_1, t_2, t_3):

- First customer t_1 waits 0 units and is served for 5 units: 5
- Second customer t_2 waits 5 units and is served for 10 units: $5 + 10$
- Third customer t_3 waits 15 units and is served for 3 units: $5 + 10 + 3$

Total time = $5 + (5 + 10) + (5 + 10 + 3) = 38$

Using the same logic, we can compute the total time for other sequences.

- $(t_1 t_2 t_3) = 5 + (5 + 10) + (5 + 10 + 3) = 38$
- $(t_1 t_3 t_2) = 5 + (5 + 3) + (5 + 3 + 10) = 31$
- $(t_2 t_1 t_3) = 10 + (10 + 5) + (10 + 5 + 3) = 43$
- $(t_2 t_3 t_1) = 10 + (10 + 3) + (10 + 3 + 5) = 41$
- $(t_3 t_1 t_2) = 3 + (3 + 5) + (3 + 5 + 10) = 29$ **The Minimum**
- $(t_3 t_2 t_1) = 3 + (3 + 10) + (3 + 10 + 5) = 34$

The sequence (t_3, t_1, t_2) gives the minimum total time of 29 units.

3. Objective Function:

The function to minimize is the sum of the wait and service times for all customers in a sequence.

$$\min \left[\sum_{i=1}^{n} \sum_{j=1}^{i} t_i \right]$$

This can be simplified to:

$$\min \left[\sum_{i=1}^{n} (n - i + 1) t_i \right]$$

- The goal is to minimize the sum of service times, weighted by the number of times each service time is repeated.

4. Proof of Optimality:

4.1 Setup:

- We have two sequences, S_1 and S_2. Both sequences are identical except for two elements t_a and t_b which are swapped.
- **S_1 and S_2 represent sequences of customers based on their service times.

$$S_1 = t_1, t_2, ..., t_a, t_b, ..., t_n$$

$$S_2 = t_1, t_2, ..., t_b, t_a, ..., t_n$$

4.2 Assumption:

- For the sake of proving optimality, assume $b - a > 0$ (meaning t_b comes after t_a in sequence S_1) and $t_b > t_a$ (meaning the service time of t_b is greater than that of t_a).
- Assuming to prove S_2 is the optimal solution, $b - a > 0$, $t_b > t_a$

4.3 Calculation:

- Compute the total time for both sequences, taking into account the order of t_a and t_b.

Given:

$$S_1 = (n - a + 1)t_a + (n - b + 1)t_b$$

$$S_2 = (n - b + 1)t_b + (n - a + 1)t_a$$

We want to compute $S_2 - S_1$.

Expanding S_2: $S_2 = nt_b - bt_b + t_b + nt_a - at_a + t_a$ and Expanding S_1: $S_1 = nt_a - at_a + t_a + nt_b - bt_b + t_b$

Now,

$S_2 - S_1 = nt_b - bt_b + t_b + nt_a - at_a + t_a - [nt_a - at_a + t_a + nt_b - bt_b + t_b]$
$S_2 - S_1 = nt_b - bt_b + t_b + nt_a - at_a + t_a - nt_a + at_a - t_a - nt_b + bt_b - t_b$

Grouping like terms: $= (nt_b - nt_a) + (-bt_b + bt_a) + (t_b - t_a)$.

1. Grouping terms with nt_b and nt_a: $nt_b - nt_a$. This is the difference in the number of times t_b and t_a are repeated.
2. Grouping terms with bt_b and bt_a: $-bt_b + bt_a$. This represents the difference in the positions of t_b and t_a.
3. Grouping terms with t_b and t_a: $t_b - t_a$. This is the difference in the service times of t_b and t_a.

Combining these grouped terms: $S_2 - S_1 = (nt_b - nt_a) + (-bt_b + bt_a) + (t_b - t_a)$

Now, factor out the common terms: $S_2 - S_1 = (b - a)t_b - (b - a)t_a$

This is the simplified expression for the difference between S_2 and S_1.

This simplifies to: $S_2 - S_1 = (b - a)t_b - (b - a)t_a$

4.4 Comparison:

- The difference $S_2 - S_1$ is computed. If this value is positive, it indicates that S_2 has a greater total time than S_1.
  - $S_2 - S_1 = (b - a)t_b - (b - a)t_a > 0$, $S_2 > S_1$

4.5 Conclusion:

- Since $S_2 - S_1 > 0$ and $t_b > t_a$, this proves that serving the customer with a shorter service time (in this case t_a) earlier is optimal. Thus, the greedy choice is validated.
- To prove the greedy choice is optimal, compare two sequences S_1 and S_2 where $t_b > t_a$ and $b - a > 0$.
- The difference $S_2 - S_1$ is positive, indicating S_2 has a greater total time than S_1.

- WRONG $S_2 - S_1 = -at_b - bt_a + at_a + bt_b$, is not equivalent to the above result.
- The correct difference between S_2 and S_1 is $S_2 - S_1 = (b - a)t_b - (b - a)t_a$.

DESIGNING A GREEDY CUSTOMER SVC ALGORITHM

Candidate set, feasibility function and solution function are given

- Need to find objective function and selection function (if one exists)

Assume $c_{i1}, \ldots, c_{im}$ have been served and c_j is next customer to serve

- How much does T increase?

Increases by $t_{i1} + t_{i2} + \ldots + t_{im} + t_j$

Choose $min_{c_j \in C} t_j$

Which c_j minimizes the amount T increases?

2. Brute Force Method:

- Initially, a brute force method was used to arrange customers in all possible ways and calculate their processing time.

The Activity Selection Problem

Definition: Activities have **start time** s_i and **finish time** f_i. The goal is to select as many activities as possible that don't overlap.

Activities: A set $S = a_1, a_2, \ldots, a_n$ of activities, and Each a_i has **start time** s_i and **finish time** f_i

Greedy Solution for Activity Selection:

- The greedy solution is to pick the activity with the earliest finish time.

Objective:
To select the maximum number of activities that don't overlap with each other.

Greedy Choice Property:
Always pick the next activity that finishes first, assuming activities are pre-sorted by their finish times.

Given Data:

Activities are represented by their start time s_i and finish time f_i.
Based on the table, the activities are already sorted by their finish times.

Goal: Schedule as many mutually compatible activities as possible

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

- A set $\{a_1, a_2, ..., a_{11}\}$ of activities. Activity a_i has **start time** s_i and **finish time** f_i.

The activity a_1 has start time s_i, and finish time f_i.

a_1 is compatible with a_2, If a_1 starts and ends before a_2 or a_2 starts and ends before $a1$,
a_1, a_2 start and end before a_3

discarded $a_i\{a_2, a_3, a_5, a_6, a_7, a_9, a_{10}\}$

	s_i	f_i
a_2	3	5
a_3	0	6
a_5	3	9
a_6	5	9
a_7	6	10
a_9	8	12
a_{10}	2	14

selected $a_i\{a_1, a_4, a_8, a_{11}\}$

	s_i	f_i
a_1	1	4
a_4	5	7
a_8	7	11
a_{11}	12	16

a_1 and a_3 are compatible, a_1 and a_2 are not compatible, a_2 and a_3 are not compatible

Solution:

1. Select the first activity (it always gets selected as it finishes first).
2. For the next activity to be selected, it should start after the previously selected activity has finished.

Example based on the table:

1. Activity 1 is selected (finishes at 4).
2. Activity 2 starts at 3 which is before Activity 1 finishes. So, skip Activity 2.
3. Activity 3 starts at 0 which is before Activity 1 finishes. So, skip Activity 3.
4. Activity 4 starts at 5 which is after Activity 1 finishes. So, Activity 4 is selected.
5. Continue this process…
   By following the greedy algorithm, the selected activities are: 1, 4, 7, and 8.

Conclusion:

The Activity Selection Problem can be efficiently solved using the greedy algorithm by always selecting the next activity that finishes first.

Compatibility: Activities a_i and a_j are Compatible if they do not overlap

- i.e., if $f_i \leq s_j$ or $s_i \geq f_j$

3. Complexity:

- The complexity involves an initial sorting step ((n\log(n)) followed by a linear pass through the activities.

Prove the Greedy Algorithm for the Activity Selection Problem,
Consider a non-empty subproblem S_k. Let a_m be an activity in S_k with earliest finish time then a_m is part of maximal size of mutually compatible activities of S_k.
a_j that is an activity with earliest finish time but not a part of solution. A_k is a solution set. a_m is the part of A_k, a_m **is an activity with earliest finish times.**. $A'_k = A_k - a_m \bigcup a_j$, that $|A_k| = |A'_k|$

```
GREEDY-ACTIVITY-SELECTOR(S, F)
// s is the array of start times
// f is the array of finish times, fis fitt
1. n = s.length
2. A = {a_1}
3.    k = 1
4.    for m = 2 to n
5.      if s[m] ≥ f[k]
6.          A = A ∪ {a_m}
7.          k = m
8.    return A
```

Line 7, `k = m`, updates the reference to the last selected activity. This ensures that the next potential activity's start time is compared with the finish time of the most recently added activity, preventing overlaps.

```
7. k = m
```

Explanation:
This line updates the value of `k` to the index `m` of the most recently selected activity. By doing this, the algorithm keeps track of the last activity that was added to the set `A`. This is crucial because, in the next iteration of the loop, the algorithm will compare the start time of the next potential activity with the finish time of this most recently selected activity (as indicated by the condition in line 5). If the next activity's start time is after or exactly at the finish time of the last selected activity, it means they are compatible, and the next activity can be added to the set `A`.

In other words, by updating `k` to `m`, the algorithm ensures that it always checks compatibility against the last activity that was added to the solution set, ensuring no overlaps.

Huffman Encoding Using Greedy Algorithm

Definition: Given a document with a set of characters, the goal is to find an encoding for the characters that minimizes the size of the document.

Key Ideas:

1. Allow character encodings to have variable lengths.
2. Characters that appear more frequently are assigned shorter codes.

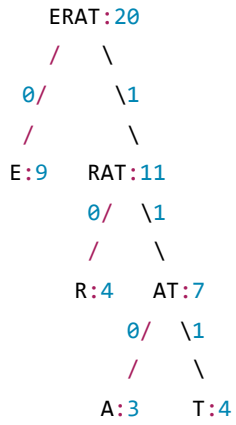
Prefix Code Property:

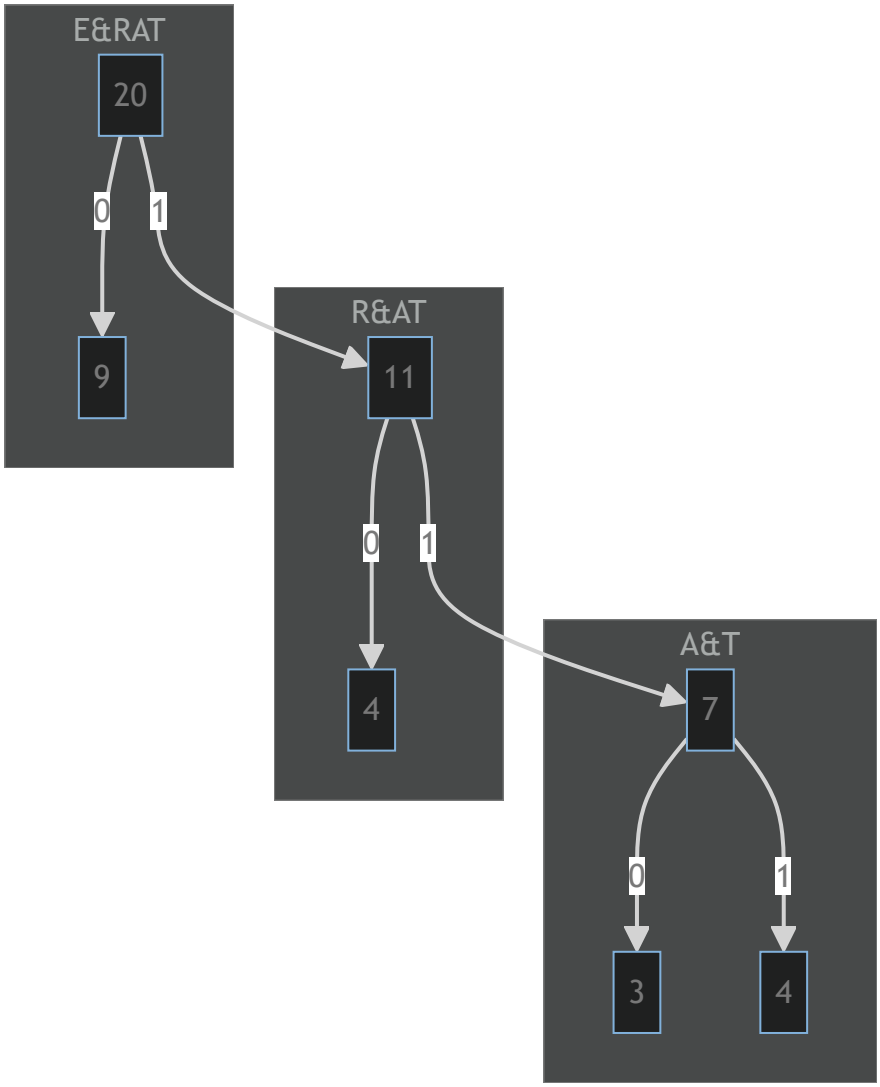
No encoding can be a prefix of another. This ensures that the encoded data can be uniquely decompressed back to the original data.

- E.g., With codes 011 and 0110, decoding 011 becomes ambiguous.

Example, Prefix code

A	T	R	E
3	4	4	9
110	**111**	**10**	**0**





Prefix Code Representation:

The encoding is represented using a binary tree where:

- Left branches are labeled `0`.
- Right branches are labeled `1`.
- All non-leaf nodes have 2 children.
- The number of leaves equals the number of characters, |C|.
- The codeword for a character is the path labels from the root to the leaf representing that character.

File Size for a Given Encoding:

Assuming a file uses an alphabet C and a character c appears f[c] times in the file:

If we encode C using a prefix code and character c has a depth $d_T[c]$ in the prefix code tree T, the resulting file size is determined by the product of the frequency of each character and its depth in the tree.

Huffman's Algorithm:

1. Create a node for each character and frequency.

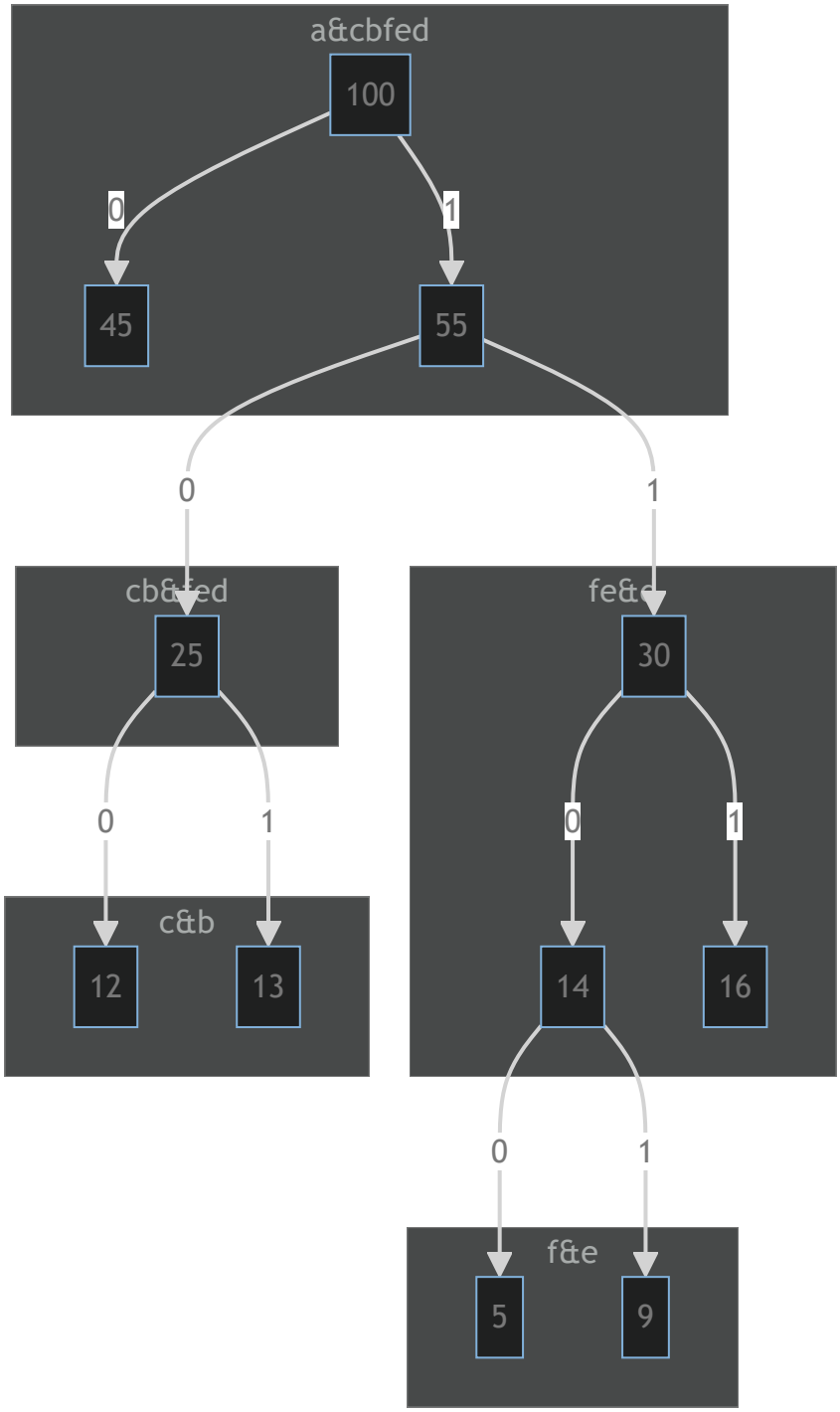
2. While there’s more than one node:
    - Select two nodes with the lowest frequency.
    - Merge them into a new node with their combined frequency.
3. The tree now represents the Huffman encoding.

By using Huffman’s algorithm, we can achieve optimal compression for a given set of characters and their frequencies.

Prefix CodeExample,

- `1 Byte = 8 Bits`

Size	Characters	a	b	c	d	e	f
800,000	Freq./Count	45	13	12	16	9	5
300,000	3 bits	000	001	010	011	100	101
220,000	Prefix Code	**0**	**101**	**100**	**111**	**1101**	**1100**



Prefix Code Representation

The encoding is represented by a binary tree

Left branch is labeled 0

Right branch is labeled 1

All non-leaves have 2 children

Number of leaves = |C|
• Codeword for char c is the path labels from root to leaf

FILE SIZE FOR A GIVEN ENCODING

Assume a file uses alphabet C, Character c appears f[c] times in the file

Encode C using a prefix code, Character c has depth $d_T[c]$ in the prefix code tree T

Resulting file size

```
HUFFMAN(C)
1. // C is the list of chars, f[x] is freq fn
2. n = C.Length
3. Q = priorityQueue(C)    // Q is a priority queue keyed on frequency
4. for i = 1 to n - l
5.    allocate a new node z
6.    z.left = left[z] = x = Q.Extract-Min()
7.    z.right = right[z] = y = Q.Extract-Min()
8.    z.f = x.f + y.f   // f[z] = f[x] + f[y]
9.    Q.Insert(z)   //L.Insert(z)
10. return Q.Extract-Min()    // Return the root of the Huffman tree
```

The provided code is a basic outline of the Huffman coding algorithm. It's a good starting point, but there are a few things to consider:

1. **Comments**: The comments in the code are brief and might not be clear to someone unfamiliar with Huffman coding. Consider expanding them for clarity.
2. **Data Structures**: The code assumes the use of a priority queue `Q` and some form of list `L`. The operations on these data structures, such as `Extract-Min` and `Insert`, need to be defined or referenced from a library.
3. **Node Structure**: The code assumes a node structure with `left`, `right`, and `f` attributes. This structure should be defined or mentioned.
4. **Return Value**: The code returns the root of the Huffman tree. This is correct, but you might want to clarify this in the comments.
5. **Typographical Errors**: There's a typo in the loop condition (`n - l` should probably be `n - 1`).
6. **Initialization**: The initialization of `Q` with `C` might be misleading. Typically, `C` is a list of characters and their frequencies, while `Q` should be initialized with nodes representing these characters, ordered by frequency.

Here's a slightly revised version of the code:

```
HUFFMAN(C)
1. // C is a list of nodes with chars and their frequencies
2. n = C.Length
3. Q = priorityQueue(C)    // Initialize Q with nodes from C, ordered by frequency
4. for i = 1 to n - 1
5.    allocate a new node z
6.    z.left = x = Q.Extract-Min()
7.    z.right = y = Q.Extract-Min()
8.    z.f = x.f + y.f
9.    Q.Insert(z)
10. return Q.Extract-Min()  // Return the root of the Huffman tree
```

If you include this code in your notes, make sure to provide context and explanations around it, especially if your audience might not be familiar with Huffman coding or the data structures used.

CORRECTNESS OF HUFFMAN©

Need to prove Huffman creates an optimal code
Optimality
Objective function

Correctness proved in two steps

- First consider codes for two least frequent characters x and y
- Then consider alphabet C_1 in which x and y are replaced by character z with f[z] = f[x]+f[y]

LEMMA 16.2

Let x and y be two characters in alphabet C that have the lowest frequencies.

There exists an optimal prefix code for C s.t. the codewords for x and y have the same length and differ only in the last bit.

PROOF IDEA

Start with any prefix code tree T
• Assume T is optimal
• Let B(T) = size of file encoded using T

Create T_1 so that x has max depth
• Swap x with a max depth leaf
• Show $B(T_1) \leq B(T)$

Create T_2, so that y and x are siblings
• Show $B(T_2) \leq B(T_1)$