CSE 5311: Design and Analysis of Algorithms

Greedy Algorithms

Alex Dillhoff

University of Texas at Arlington

Greedy Algorithms

Greedy algorithms are a class of algorithms that yield **locally** optimal solutions.

In cases where the local optimum is also the global optimum, greedy algorithms are ideal.

Even in cases where the global solution is more elusive, a local solution may be sufficient.

Activity Selection

Activity Selection

Given a set of activities that need to be scheduled using a common resource, the **activity selection** problem is to find the maximum number of activities that can be scheduled without overlapping.

Activity Selection

Definition

- Each activity has a start time s_i and finish time f_i, where $0 \leq s_i < f_i < \infty$.
- An activity a_i takes place over the interval $[s_i, f_i)$.
- Two activities a_i and a_j are mutually compatible if $s_i \geq f_j$ or $s_j \geq f_i$.

Activity Selection

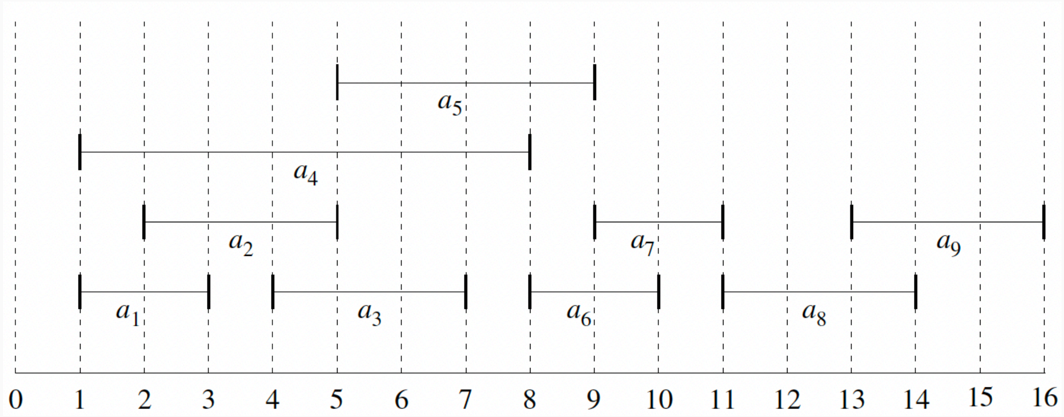
Objective: Sort the activities by finish time and find the largest subset of mutually compatible activities.

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

Activity Selection

How many mutually compatible sets are there?

Activity Selection



Activities over time (Cormen et al.).

Activity Selection

How many mutually compatible sets are there?

1. $\{a_1, a_3, a_6, a_8\}$
2. $\{a_1, a_3, a_6, a_9\}$
3. $\{a_1, a_3, a_7, a_9\}$
4. $\{a_1, a_5, a_7, a_8\}$
5. $\{a_1, a_5, a_7, a_9\}$
6. $\{a_2, a_5, a_7, a_8\}$
7. $\{a_2, a_5, a_7, a_8\}$

Optimal Substructure

How do we verify that this problem has optimal substructure?

Optimal Substructure

How do we verify that this problem has optimal substructure?

Step 1: Define the problem formally...

Optimal Substructure

Define S_{ij} as the set of all activities that start after a_i finishes and finish before a_j starts.

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$$

This defines a clear subset of the original set of data.

Optimal Substructure

Which activities are those in S_{ij} compatible with?

Optimal Substructure

Which activities are those in S_{ij} compatible with?

1. any a_i that finish by f_i
2. any a_i that start no earlier than s_j.

Optimal Substructure

Given this subset, our subproblem is that of finding a maximum set of mutually compatible activities in S_{ij}, denoted A_{ij}.

Optimal Substructure

If $a_k \in A_{ij}$, we are left with two subproblems:

- Find mutually compatible activities in S_{ik} – starts after a_i finishes and finish before a_k starts.
- Find mutually compatible activities in S_{kj} – starts after a_k finishes and finish before a_j start.

Optimal Substructure

- The two subsets are defined as $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$, respectively.
- Then $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$.
- The size of the set is given by $|A_{ij}| = |A_{ik}| + 1 + |A_{kj}|$.

Optimal Substructure

Claim: If our problem has optimal substructure, then the optimal solution A_{ij} must include optimal solutions for S_{ik} and S_{kj}.

This claim can be proven using the **cut-and-paste** method used in **Dynamic Programming**.

Optimal Substructure

This technique works by showing that if a solution to a problem is not optimal, then there exists a way to **cut** the suboptimal portion and **paste** an optimal one.

This will lead to a contradiction because the original was assumed to be optimal.

Optimal Substructure

Proof

- Suppose that A_{kj} is not optimal, and we could find a set A'_{kj} that is larger.
- Then we could replace A_{kj} with A'_{kj} in A_{ij} to obtain a larger set.
- This contradicts the assumption that A_{ij} is optimal.

Optimal Substructure

Simplified

- If the claim is that the given solution is optimal, and the solution is constructed from optimal solutions to subproblems, then there cannot exist any other solution that is better.

- Another way to look at this: if we construct optimal solutions to subproblems, then the solution to the original problem must be optimal.

Recursive Solution

Let $c[i,j]$ be the size of the optimal solution for S_{ij}.

The size is computed as

$$c[i,j] = c[i,k] + c[k,j] + 1.$$

Recursive Solution

This dynamic programming solution assumes we know the optimal solution for all subproblems.

To know this, we need to examine all possibilities which include a_k in the solution.

Recursive Solution

To know this, we need to examine all possibilities which include a_k in the solution.

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max\{c[i,k] + c[k,j] + 1 : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset. \end{cases}$$

Greedy Solution

The greedy solution is the naive one: **select an activity that leaves the resource available for as many other activities as possible, which is the activity that finishes first.**

If multiple activities finish at the same time, select one arbitrarily.

Greedy Solution

The optimal solution consists of all a_i that start after a_k finishes, where a_k is the last activity to finish:

$$S_k = \{a_i \in S : s_i \geq f_k\}.$$

Greedy Solution

Is the greedy solution optimal?

- Suppose that $a_m \in S_k$ is the activity that finishes first.

Greedy Solution

Is the greedy solution optimal?

- Suppose that $a_m \in S_k$ is the activity that finishes first.
- Then it must be included in the maximum size subset of mutually compatible activities A_k.

Greedy Solution

Is the greedy solution optimal?

- Suppose that $a_m \in S_k$ is the activity that finishes first.
- Then it must be included in the maximum size subset of mutually compatible activities A_k.
- Suppose we are given A_k and we look at $a_j \in A_k$, the activity that finishes first.

Greedy Solution

Is the greedy solution optimal?

- If $a_j = a_m$, then the greedy solution is optimal.

Greedy Solution

Is the greedy solution optimal?

- If $a_j = a_m$, then the greedy solution is optimal.
- If $a_j \neq a_m$, then we can replace a_j with a_m since they are both compatible with all other activities in A_k.

Greedy Solution

Is the greedy solution optimal?

- If $a_j = a_m$, then the greedy solution is optimal.
- If $a_j \neq a_m$, then we can replace a_j with a_m since they are both compatible with all other activities in A_k.
- Picking the activity that finishes first does not change the size of the optimal solution.

Greedy Solution

The solution is top-down:

1. pick the solution that finishes first,
2. remove all activities that are incompatible with the chosen activity,
3. repeat until no activities remain.

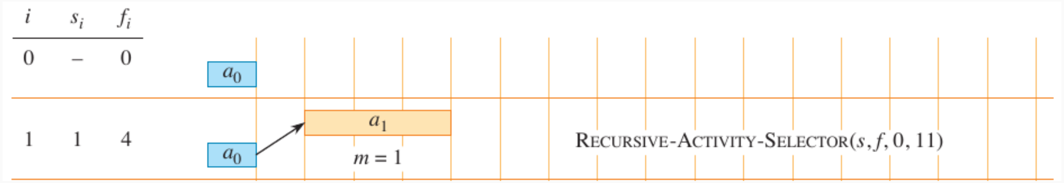
Recursive Greedy Algorithm

```python
def recursive_activity_selector(s, f, k, n):
    m = k + 1
    while m <= n and s[m] < f[k]:
        m += 1
    if m <= n:
        return [m] + recursive_activity_selector(s, f, m, n)
    else:
        return []
```

Recursive Greedy Algorithm

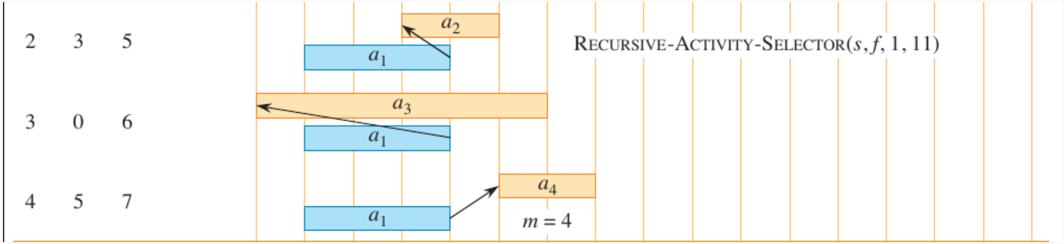
- Assumes that `f` is sorted in increasing order.
- `k` represents the index of the current subproblem.
- The number of activities is given by `n`.
- The **`while`** loop increments `m` until it finds an activity that starts after activity `k` finishes.
- If such an activity exists, it is added to the solution set and the algorithm is called recursively with the new subproblem.

Recursive Greedy Algorithm



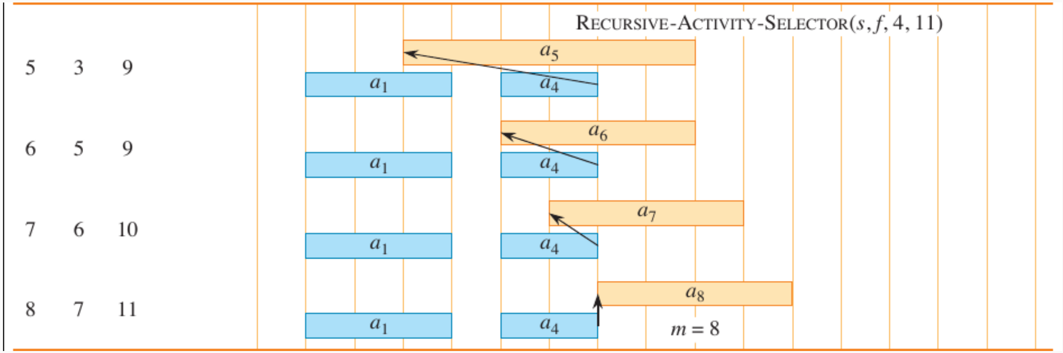
Initial call to `recursive_activity_selector`.

Recursive Greedy Algorithm



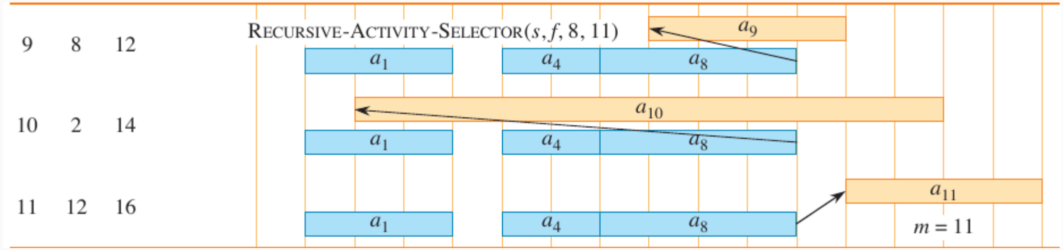
Call to `recursive_activity_selector` from a_1.

Recursive Greedy Algorithm



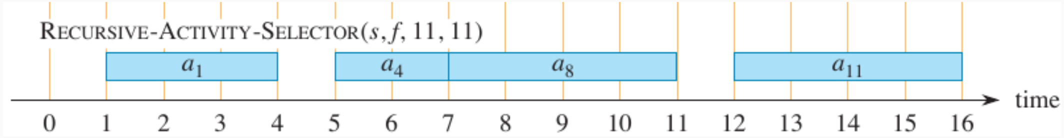
Call to `recursive_activity_selector` from a_4.

Recursive Greedy Algorithm



Call to `recursive_activity_selector` from a_8.

Recursive Greedy Algorithm



Result of `recursive_activity_selector`.

Analysis

At first glance, the algorithm appears to be $O(n^2)$ because of the `while` loop coupled with the recursive call.

Once an activity has been selected, the recursive call only considers the activities next $n - k$ activities.

The `while` loop picks up where the previous call left off, so the total number of iterations is n.

The algorithm is $O(n)$.

Iterative Algorithm

The previous solution can be adapted to an iterative one.

```
def greedy_activity_selector(s, f):
    n = len(s) - 1
    A = [1]
    k = 1
    for m in range(2, n + 1):
        if s[m] >= f[k]:
            A.append(m)
            k = m
    return A
```

Iterative Algorithm

In the `for` loop, the first line essentially asks if $a_m \in S_k$.

If so, then add it to the solution set and update k to m.

Analysis

The analysis of the iterative approach is much clearer.

The loop goes over all activities once, so the algorithm is $O(n)$.

Properties of Greedy Algorithms

Properties of Greedy Algorithms

You can probably imagine a problem for which a greedy solution would not provide the optimal solution.

Properties of Greedy Algorithms

You can probably imagine a problem for which a greedy solution would not provide the optimal solution.

Path planning is one such problem.

Properties of Greedy Algorithms

Path planning is one such problem.

If we greedily chose the shortest path at each step, we may have missed a shorter path that is not the shortest at each step.

The activity selection problem just so happens to be a perfect candidate for a greedy solution, **but what makes it so?**

Properties of Greedy Algorithms

Review of activity selection.

1. Determine the optimal substructure.

Properties of Greedy Algorithms

Review of activity selection.

1. Determine the optimal substructure.
2. Develop a recursive solution.

Properties of Greedy Algorithms

Review of activity selection.

1. Determine the optimal substructure.
2. Develop a recursive solution.
3. Show that making the greedy choice leaves only a single subproblem.

Properties of Greedy Algorithms

Review of activity selection.

1. Determine the optimal substructure.
2. Develop a recursive solution.
3. Show that making the greedy choice leaves only a single subproblem.
4. Prove that making the greedy choice leads to an optimal solution.

Properties of Greedy Algorithms

Review of activity selection.

1. Determine the optimal substructure.
2. Develop a recursive solution.
3. Show that making the greedy choice leaves only a single subproblem.
4. Prove that making the greedy choice leads to an optimal solution.
5. Develop a recursive algorithm.

Properties of Greedy Algorithms

Review of activity selection.

1. Determine the optimal substructure.
2. Develop a recursive solution.
3. Show that making the greedy choice leaves only a single subproblem.
4. Prove that making the greedy choice leads to an optimal solution.
5. Develop a recursive algorithm.
6. Convert it to an iterative algorithm.

Properties of Greedy Algorithms

The first couple of steps are common to dynamic programming problems.

In this case, we could have jumped straight to the greedy approach.

Properties of Greedy Algorithms

Filtering out these extra steps leaves us with:

1. Cast the optimization problem as one in which we make a choice and are left with a single subproblem.

Properties of Greedy Algorithms

Filtering out these extra steps leaves us with:

1. Cast the optimization problem as one in which we make a choice and are left with a single subproblem.
2. Prove that the greedy choice is optimal.

Properties of Greedy Algorithms

Filtering out these extra steps leaves us with:

1. Cast the optimization problem as one in which we make a choice and are left with a single subproblem.
2. Prove that the greedy choice is optimal.
3. Demonstrate optimal substructure: if you make a greedy choice, then you are left with a subproblem such that combining an optimal solution with the greedy choice made previously, you end up with an optimal solution to the original problem.

Properties of Greedy Algorithms

We need two properties to prove that a greedy solution is optimal:

1. the **greedy choice property** and the
2. **optimal substructure property**.

Greedy Choice Property

The **greedy choice property** states that the optimal solution can be found by making locally greedy choices.

In Dynamic programming the choices at each step are made from the knowledge of optimal solutions to subproblems.

Greedy Choice Property

A greedy solution also makes a choice at each step, but it is only based on local information.

They key of this property is to show that the greedy choice is optimal at each step.

Greedy Choice Property

For the activity selection problem, the steps were

1. Examine the optimal solution.

Greedy Choice Property

For the activity selection problem, the steps were

1. Examine the optimal solution.
2. If it has the greedy choice, then the greedy choice is optimal.

Greedy Choice Property

For the activity selection problem, the steps were

1. Examine the optimal solution.
2. If it has the greedy choice, then the greedy choice is optimal.
3. If it does not have the greedy choice, then replace the suboptimal choice with the greedy choice.

Optimal Substructure Property

A problem has optimal substructure if the optimal solution contains optimal solutions to subproblems.

We demonstrated this earlier for activity selection.

Optimal Substructure Property

Start with the assumption that we arrived to a subproblem by making greedy choices.

Next step: show that the optimal solution to the subproblem combined with the greedy choice leads to an optimal solution to the original problem.

Greedy vs. Dynamic Programming

Since there is such overlap between greedy algorithms and dynamic programming in terms of their properties, it is important to understand the differences between the two.

To illustrate these difference, we will look at two variations of the same problem.

$0-1$ Knapsack Problem

$0-1$ Knapsack Problem

Consider the $0-1$ **knapsack problem.**

- You have n items.
- Item i is worth v_i and weighs w_i.
- Find the most valuable subset of items with total weight less than or equal to W.
- Items cannot be divided.

$0-1$ Knapsack Problem

If the most valuable subset of items weighing at most W includes item j...

then the remaining weight must be the most valuable subset of items weighing at most $W - w_j$ taken from $n-1$ original items excluding item j.

Fractional Knapsack Problem

This is similar to the $0 - 1$ knapsack problem, but items can be divided.

The objective is to maximize the value of the items in the knapsack.

Fractional Knapsack Problem

Optimal substructure: if the most valuable subset weighing at most W includes the weight w of item j, then the remaining weight must be the most valuable subset weighing at most $W - w$ that can be taken from the $n - 1$ original items plus $w_j - w$ of item j.

There is some fraction of item j left after taking w of it.

Showing the Greedy Property

It is established that both problems have optimal substructure.

Only the fractional knapsack problem has the greedy property.

Showing the Greedy Property

```
def fractional_knapsack(v, w, W):
    n = len(v)
    load = 0
    i = 0
    while load < W and i <= n:
        if w[i] <= W - load:
            load += w[i]
            i += 1
        else:
            load += (W - load) * v[i] / w[i]
```

Showing the Greedy Property

If we are able to sort each item by its value-to-weight ratio, then the greedy choice is to take as much as possible from the most valuable item first, the second most valuable item next, and so on.

This considers n items in the worst case, and the items need to be sorted by value-to-weight ratio.

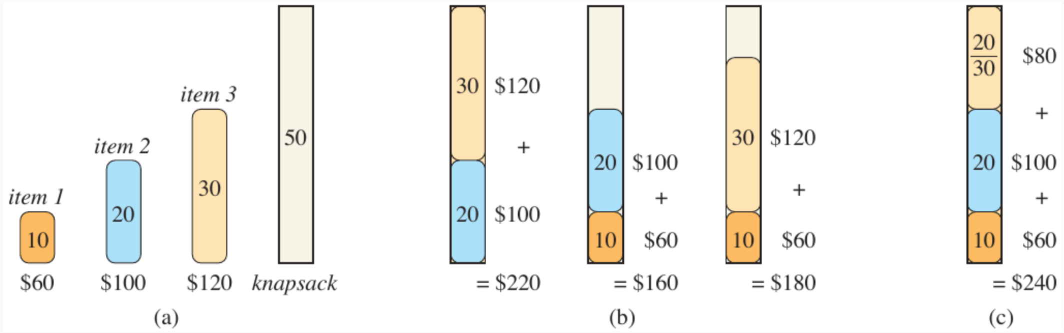
The algorithm is $O(n \log n)$.

Showing the Greedy Property

The $0 - 1$ knapsack problem does not have the greedy property.

The greedy choice is to take the item with the highest value-to-weight ratio.

Showing the Greedy Property



The 0 — 1 knapsack problem (Cormen et al.).

Showing the Greedy Property

In this problem, we have a knapsack whose total capacity is $W = 50$.

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

Showing the Greedy Property

The fractional algorithm would selection the first item since it has the greatest value-to-weight ratio.

The $0 - 1$ knapsack problem, however, would select the second and third items to maximize the value of the items in the knapsack.

Huffman Codes

Huffman Codes

Huffman coding is a lossless data compression algorithm that assigns variable-length codes to input characters, with lengths based on the frequencies of occurrence for those characters.

Originally developed by David A. Huffman in 1952 during his Ph.D. at MIT, he published the algorithm under the title ”A Method for the Construction of Minimum-Redundancy Codes”.

Huffman Codes

Huffman coding involves:

- Building a Huffman tree from the input characters and their frequencies.
- Traversing the tree to assign codes to each character.

Huffman Codes

Suppose we have a document consisting of 6 unique characters, each represented by a byte (8 bits).

- We could represent these 6 characters using 3 bits, since that is the minimum number of bits needed to represent 6 unique values.
- This is known as a **fixed-length code**.

Huffman Codes

Suppose we have a document consisting of 6 unique characters, each represented by a byte (8 bits).

- We could represent these 6 characters using 3 bits, since that is the minimum number of bits needed to represent 6 unique values.
- This is known as a **fixed-length code**.
- If we instead assigned a **variable-length code** to each character based on its frequency of occurrence, we would further reduce the footprint of the file size.

Huffman Codes

Character	Frequency (in thousands)	Variable-length code
A	45	0
B	13	101
C	12	100
D	16	111
E	9	1101
F	5	1100

Huffman Codes

- The encoding is optimal considering the overall but length of the encoded file.
- Based on the above **fixed-length code**, the file size is 300,000 bits.
- The **variable-length code** reduces the file size to 224,000 bits.

Huffman Codes

How many bits are needed to encode $n \geq 2$ characters?

Huffman Codes

How many bits are needed to encode $n \geq 2$ characters?

$$\lceil \lg n \rceil$$

Prefix-Free Codes

A **prefix-free code** is a code in which no codeword is also a prefix of another codeword.

This property simplifies decoding since the code can be read from left to right without ambiguity.

The codeword ”beef” has the encoding $1011101111011100 = 101 \cdot 1101 \cdot 1101 \cdot 1100$, where $\cdot$ denotes concatenation.

Prefix-Free Codes

If a code of 1 were used, then the code would be a prefix of all other codes.

This would make decoding ambiguous.

Prefix-Free Codes

How are the codes decoded?

Huffman Coding uses a binary tree.

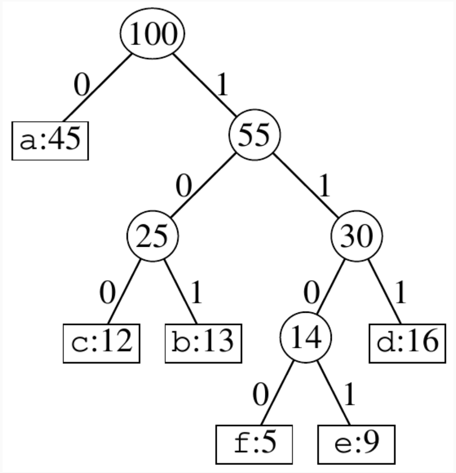
- Starting with the first bit in the encoded message, traverse the tree until a leaf node is reached.
- The character at the leaf node is the decoded character.
- The tree is known as a **Huffman tree**.

Prefix-Free Codes

A **full binary tree**, where each nonleaf node has two subnodes, is optimal for decoding.

If the tree has this property then an optimal prefix-free code has $|C|$ leaves and exactly $|C| - 1$ internal nodes.

Huffman Tree



A Huffman tree (Cormen et al.).

Huffman Tree

Given a character c with frequency $c.freq$, let $d_T(c)$ denote the depth of character c in tree T. The cost of the code in bits is given by

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c).$$

The depth of the character $d_T(c)$ is used since it also denotes the length of the codeword.

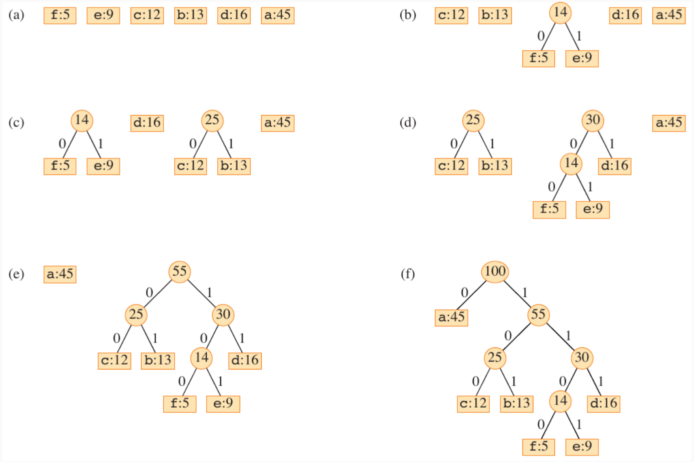
Huffman Codes

```
def huffman(C):
    n = len(C)
    Q = build_min_heap(C)
    for _ in range(n - 1):
        z = Node(0)
        z.left = x = extract_min(Q)
        z.right = y = extract_min(Q)
        z.freq = x.freq + y.freq
        insert(Q, z)
    return extract_min(Q)
```

Huffman Codes

- The function above builds a Huffman tree from a set of characters C.
- At each iteration, the two nodes with the smallest frequencies are extracted from the queue Q and are used to create a new node z.
- This node represents the sum of the frequencies of the two nodes.
- The node is then inserted back into the queue so that it can be used in future iterations.

Huffman Codes



(a) f:5 e:9 c:12 b:13 d:16 a:45
(b) c:12 b:13 14 f:5 e:9 d:16 a:45
(c) 14 f:5 e:9 d:16 25 c:12 b:13 a:45
(d) 25 c:12 b:13 30 14 f:5 e:9 d:16 a:45
(e) a:45 55 25 c:12 b:13 30 14 f:5 e:9 d:16
(f) 100 a:45 55 25 c:12 b:13 30 14 f:5 e:9 d:16
0
1

Building a Huffman tree (Cormen et al.).

Analysis

- If the priority queue is implemented as a binary min-heap, the call to `build_min_heap` initializes the priority queue in $O(n)$ time.
- The `for` loop runs $n - 1$ times, calling `extract_min` twice and `insert` once.
- Each call to `extract_min` takes $O(\lg n)$ time yielding a total of $O(n \lg n)$ time.