ETH zürich

Übung 9

Datenstrukturen und Algorithmen, D-MATH, ETH Zurich

Programm von heute

Feedback letzte Übung

Repetition Theorie
- Aktivitätenauswahl
- Rekursive Problemlösestrategien
- Huffman Codierung

In-Class-Exercise (praktisch)

Hinweise zu den Aufgaben

1. Feedback letzte Übung

2. Repetition Theorie

DP-Beispiel: Längste gemeinsame Teilfolge

Definition

Eine *Teilfolge* einer Folge entsteht, indem man einige oder keine der Elemente der ursprünglichen Folge entfernt. Zum Beispiel ist "AC" eine Teilfolge von "ABC".

Problem

Gegeben zwei Sequenzen X und Y, finde die Länge der längsten gemeinsamen Teilfolge von X und Y.

Beispiel: Längste gemeinsame Teilfolge

Beispiel

X: PROGRAM
Y: ARMOR

Antwort?

Beispiel: Längste gemeinsame Teilfolge

Beispiel

X: PROGRAM
Y: ARMOR

Antwort

Länge 3: ROR

Teilprobleme?

String X der Länge m und String Y der Länge n:
Welche Teilprobleme gibt es?

Teilprobleme?

String X der Länge m und String Y der Länge n:
Welche Teilprobleme gibt es?

- falls letztes Zeichen gleich: $+1$ und beide Strings um eins kürzer machen
- X um eins kürzer machen, Y gleich
- Y um eins kürzer machen, X gleich

LCS Rekursive Lösung

```
int lcs(const std::string& X, const std::string& Y, int m, int n) {
    if (m == 0 || n == 0) {
        return 0;
    }
    if (X[m - 1] == Y[n - 1]) {
        return 1 + lcs(X, Y, m - 1, n - 1);
    } else {
        return std::max(lcs(X, Y, m - 1, n),
                        lcs(X, Y, m, n - 1));
    }
}
```

LCS Dynamische Programmierung

Stattdessen können wir dynamische Programmierung verwenden, um dieses Problem zu lösen, indem wir eine Tabelle erstellen, um die Längen der längsten gemeinsamen Teilfolgen der Präfixe von X und Y zu speichern:

LCS Dynamische Programmierung

Stattdessen können wir dynamische Programmierung verwenden, um dieses Problem zu lösen, indem wir eine Tabelle erstellen, um die Längen der längsten gemeinsamen Teilfolgen der Präfixe von X und Y zu speichern:

- Aktualisieren Sie die Tabellenwerte von oben links nach unten rechts.

LCS Dynamische Programmierung

Stattdessen können wir dynamische Programmierung verwenden, um dieses Problem zu lösen, indem wir eine Tabelle erstellen, um die Längen der längsten gemeinsamen Teilfolgen der Präfixe von X und Y zu speichern:

- Aktualisieren Sie die Tabellenwerte von oben links nach unten rechts.
- Wenn die Zeichen an der aktuellen Position übereinstimmen, setzen Sie den Wert der aktuellen Zelle auf den Wert der diagonalen Zelle, erhöht um eins, oder auf eins, wenn es nicht vorhanden ist.

LCS Dynamische Programmierung

Stattdessen können wir dynamische Programmierung verwenden, um dieses Problem zu lösen, indem wir eine Tabelle erstellen, um die Längen der längsten gemeinsamen Teilfolgen der Präfixe von X und Y zu speichern:

- Aktualisieren Sie die Tabellenwerte von oben links nach unten rechts.
- Wenn die Zeichen an der aktuellen Position übereinstimmen, setzen Sie den Wert der aktuellen Zelle auf den Wert der diagonalen Zelle, erhöht um eins, oder auf eins, wenn es nicht vorhanden ist.
- Wenn sie nicht übereinstimmen, setzen Sie den Wert der aktuellen Zelle auf das Maximum der linken und oberen Zellwerte oder auf Null, wenn sie nicht vorhanden sind.

LCS Tabelle

X/Y	P	R	O	G	R	A	M
A							
R							
M							
O							
R							

LCS Tabelle

X/Y	P	R	O	G	R	A	M
A	0	0	0	0	0	1	1
R							
M							
O							
R							

LCS Tabelle

X/Y	P	R	O	G	R	A	M
A	0	0	0	0	0	1	1
R	0	1	1	1	1	1	1
M							
O							
R							

LCS Tabelle

X/Y	P	R	O	G	R	A	M
A	0	0	0	0	0	1	1
R	0	1	1	1	1	1	1
M	0	1	1	1	1	1	2
O							
R							

LCS Tabelle

X/Y	P	R	O	G	R	A	M
A	0	0	0	0	0	1	1
R	0	1	1	1	1	1	1
M	0	1	1	1	1	1	2
O	0	1	2	2	2	2	2
R							

LCS Tabelle

X/Y	P	R	O	G	R	A	M
A	0	0	0	0	0	1	1
R	0	1	1	1	1	1	1
M	0	1	1	1	1	1	2
O	0	1	2	2	2	2	2
R	0	1	2	2	3	3	3

LCS

LCS finden (Lösung rekonstruieren)?

LCS

LCS finden (Lösung rekonstruieren)?

Um die LCS zu finden, gehen Sie von der unteren rechten Ecke aus rückwärts und markieren Sie den Anfangsbuchstaben jedes diagonalen Pfeils.

LCS

LCS finden (Lösung rekonstruieren)?

Um die LCS zu finden, gehen Sie von der unteren rechten Ecke aus rückwärts und markieren Sie den Anfangsbuchstaben jedes diagonalen Pfeils.

X/Y	P	**R**	**O**	G	**R**	A	M
A	0	0	0	0	0	1	1
R	0	**1**	1	1	1	1	1
M	0	1	1	1	1	1	2
O	0	1	**2**	2	2	2	2
R	0	1	2	2	**3**	3	3

LCS Zeitkomplexität

Frage

Wie vergleicht sich die Zeitkomplexität des DP-Algorithmus mit der des naiven rekursiven Algorithmus?

LCS Zeitkomplexität

Frage

Wie vergleicht sich die Zeitkomplexität des DP-Algorithmus mit der des naiven rekursiven Algorithmus?

Naiver Algorithmus

Der naive Algorithmus hat eine exponentielle Zeitkomplexität von $O(2^{n+m})$, wobei n und m die Längen der beiden Sequenzen sind.

Dynamischer Programmieralgorithmus

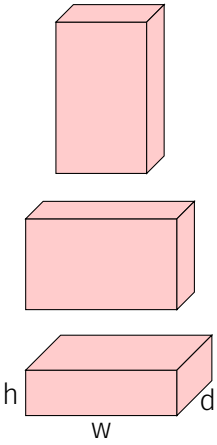
Der dynamische Programmieralgorithmus hat eine polynomiale Zeitkomplexität von $O(n \cdot m)$.

Quiz: Boxen Stapeln

- Gegeben: n Boxen mit Grössen $w_i \times d_i \times h_i$
- Gesucht: maximale Höhe eines erlaubten Stapels
- Erlaubter Stapel: Grundfläche gestapelter Boxen muss in beiden Richtungen (Breite und Tiefe) strikt kleiner werden

Boxen Stapeln

Wir gehen davon aus, dass es genügend Boxen jeder Sorte gibt, so dass jede Box in jeder Orientierung verfügbar ist (Abbildung rechts).

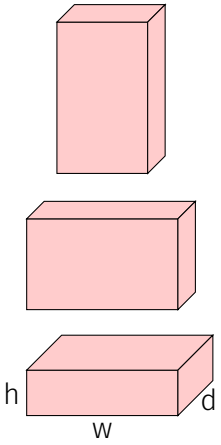


Box	1	2	3	4	5	6
$[w \times d \times h]$	$[1 \times 2 \times 3]$	$[1 \times 3 \times 2]$	$[2 \times 3 \times 1]$	$[3 \times 4 \times 5]$	$[3 \times 5 \times 4]$	$[4 \times 5 \times 3]$

Boxen Stapeln

Wir gehen davon aus, dass es genügend Boxen jeder Sorte gibt, so dass jede Box in jeder Orientierung verfügbar ist (Abbildung rechts).

Erstellen Sie einen DP Algorithmus zum Finden der maximalen Höhe eines erlaubten Stapels.



Box	1	2	3	4	5	6
$[w \times d \times h]$	$[1 \times 2 \times 3]$	$[1 \times 3 \times 2]$	$[2 \times 3 \times 1]$	$[3 \times 4 \times 5]$	$[3 \times 5 \times 4]$	$[4 \times 5 \times 3]$

Lösung?

Denken Sie darüber nach, wie Sie die Aufgabe mit dynamischer Programmierung lösen könnten.

Lösungsidee

- $n \times n$ Tabelle
- Eintrag in Zeile i, Spalte j: Höhe eines höchsten Turmes mit maximal i Boxen und Basisbox j.

$[w \times d]$	$[1 \times 2]$	$[1 \times 3]$	$[2 \times 3]$	$[3 \times 4]$	$[3 \times 5]$	$[4 \times 5]$
h	3	2	1	5	4	3
1	<u>3</u>	2	1	5	4	3
2	3	2	<u>4</u>	8	8	8
3	3	2	4	<u>9</u>	8	11
4	3	2	4	9	8	<u>12</u>

Bestimmung der Tabelle: $\Theta(n^3)$, für jeden Eintrag müssen alle Einträge der vorherigen Zeile durchlaufen werden. Berechnung der optimalen Lösung durch Rückverfolgung im schlechtesten Fall $\Theta(n^2)$.

Alternative Lösungsidee

- $1 \times n$ Tabelle, topologisch sortiert[1] nach Halbordnung Stapelbarkeit
- Eintrag an Position j: Höhe eines höchsten Turmes mit Basisbox j.

$[w \times d]$	$[1 \times 2]$	$[1 \times 3]$	$[2 \times 3]$	$[3 \times 4]$	$[3 \times 5]$	$[4 \times 5]$
h	3	2	1	5	4	3
	3	2	4	9	8	12

Topologisches Sortieren in $\Theta(n^2)$. Durchlaufen von rechts nach links in $\Theta(n)$, insgesamt $\Theta(n^2)$. Rückverfolgung auch $\Theta(n^2)$

[1] Erklärung folgt

Gierige Auswahl

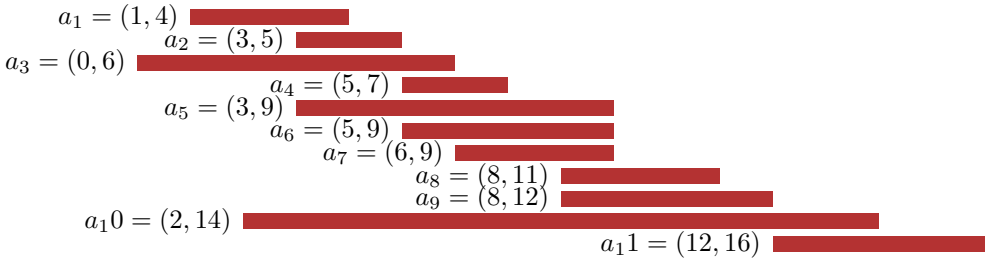
Ein rekursiv lösbares Optimierungsproblem kann mit einem **gierigen (greedy) Algorithmus** gelöst werden, wenn es die folgende Eigenschaften hat:

- Das Problem hat **optimale Substruktur**: die Lösung eines Problems ergibt sich durch Kombination optimaler Teillösungen.
- Es gilt die **greedy choice property**: Die Lösung eines Problems kann konstruiert werden, indem ein lokales Kriterium herangezogen wird, welches nicht von der Lösung der Teilprobleme abhängt.

Beispiele: Gebrochenes Rucksackproblem, Huffman-Coding (s.u.)
Gegenbeispiele: Rucksackproblem. Optimaler binärer Suchbaum.

Aktivitäten Auswahl

Koordination von Aktivitäten, die gemeinsame Resource exklusiv nutzen. Aktivitäten $S = \{a_1, a_2, \ldots, a_n\}$ mit Start und Endzeiten $0 \leq s_i \leq f_i < \infty$, aufsteigend sortiert nach Endzeiten.



Aktivitäten-Auswahl-Problem: Finde maximale Teilmenge (maximal in Anzahl Elementen) kompatibler (nichtüberlappender) Aktivitäten.

Dynamic Programming Ansatz?

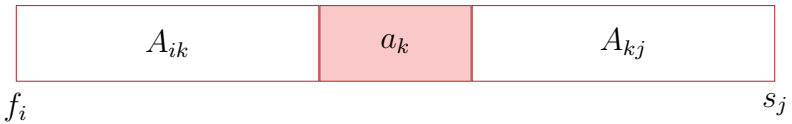
Sei $S_{ij} = \{a_k : f_i \leq s_k \wedge f_k \leq s_j\}$.

Dynamic Programming Ansatz?

Sei $S_{ij} = \{a_k : f_i \leq s_k \wedge f_k \leq s_j\}$.
Sei A_{ij} eine maximale Teilmenge kompatibler Aktivitäten aus S_{ij}.

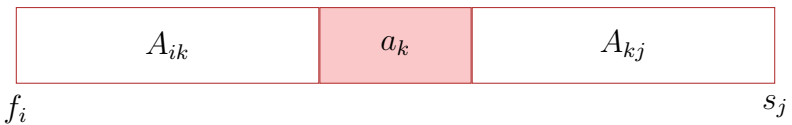
Dynamic Programming Ansatz?

Sei $S_{ij} = \{a_k : f_i \leq s_k \wedge f_k \leq s_j\}$.
Sei A_{ij} eine maximale Teilmenge kompatibler Aktivitäten aus S_{ij}.
Sei $a_k \in A_{ij}$ und $A_{ik} = S_{ik} \cap A_{ij}$, $A_{ki} = S_{kj} \cap A_{ij}$, also $A_{ij} = A_{ik} + \{a_k\} + A_{kj}$.



Dynamic Programming Ansatz?

Sei $S_{ij} = \{a_k : f_i \leq s_k \wedge f_k \leq s_j\}$.
Sei A_{ij} eine maximale Teilmenge kompatibler Aktivitäten aus S_{ij}.
Sei $a_k \in A_{ij}$ und $A_{ik} = S_{ik} \cap A_{ij}$, $A_{ki} = S_{kj} \cap A_{ij}$, also $A_{ij} = A_{ik} + \{a_k\} + A_{kj}$.



Klar: A_{ik} und A_{kj} müssen maximal sein, sonst wäre $A_{ij} = A_{ik} + \{a_k\} + A_{kj}$ nicht maximal.

Dynamischer Programmieransatz?

Kurze Wiederholung

Warum müssen A_{ik} und A_{kj} maximale Teilmengen von kompatiblen Aktivitäten sein, damit A_{ij} ebenfalls maximal ist?

Dynamischer Programmieransatz?

Kurze Wiederholung

Warum müssen A_{ik} und A_{kj} maximale Teilmengen von kompatiblen Aktivitäten sein, damit A_{ij} ebenfalls maximal ist?

Der Grund dafür ist, dass, wenn entweder A_{ik} oder A_{kj} nicht maximal wären, zusätzliche kompatible Aktivitäten zu diesen Teilmengen hinzugefügt werden könnten.

Dynamic Programming Ansatz?

Sei $c_{ij} = |A_{ij}|$.
Dann gilt folgende Rekursion

$$c_{ij} = \begin{cases} 0 & \text{falls } S_{ij} = \emptyset, \\ \max_{a_k \in S_{ij}} \{c_{ik} + c_{kj} + 1\} & \text{falls } S_{ij} \neq \emptyset. \end{cases}$$

$\Rightarrow$ Dynamische Programmierung.

Dynamic Programming Ansatz?

Sei $c_{ij} = |A_{ij}|$.
Dann gilt folgende Rekursion

$$c_{ij} = \begin{cases} 0 & \text{falls } S_{ij} = \emptyset, \\ \max_{a_k \in S_{ij}} \{c_{ik} + c_{kj} + 1\} & \text{falls } S_{ij} \neq \emptyset. \end{cases}$$

$\Rightarrow$ Dynamische Programmierung.

Aber es geht noch einfacher.

Greedy

Intuition: Wähle zuerst die Aktivität, die die früheste Endzeit hat (a_1). Das lässt maximal viel Platz für weitere Aktivitäten.

Greedy

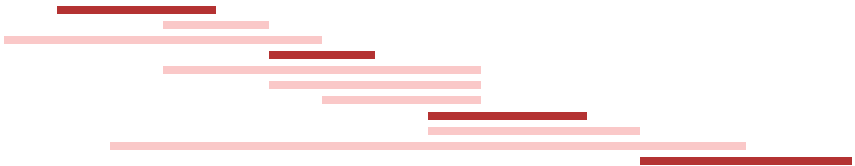
Intuition: Wähle zuerst die Aktivität, die die früheste Endzeit hat (a_1). Das lässt maximal viel Platz für weitere Aktivitäten.

Verbleibendes Teilproblem: Aktivitäten, die starten nachdem a_1 endet. (Es gibt keine Aktivitäten, die vor dem Start von a_1 enden.)

Greedy

Intuition: Wähle zuerst die Aktivität, die die früheste Endzeit hat (a_1). Das lässt maximal viel Platz für weitere Aktivitäten.

Verbleibendes Teilproblem: Aktivitäten, die starten nachdem a_1 endet. (Es gibt keine Aktivitäten, die vor dem Start von a_1 enden.)



Greedy

Theorem 1

Gegeben: Teilproblem S_k, und eine Aktivität a_m aus S_k mit frühester Endzeit. Dann ist a_m in einer maximalen Teilmenge von kompatiblen Aktivitäten aus S_k enthalten.

Sei A_k maximal grosse Teilmenge mit kompatiblen Aktivitäten aus S_k, und a_j eine Aktivität aus A_k mit frühester Endzeit. Wenn $a_j = a_m \Rightarrow$ fertig. Wenn $a_j \neq a_m$, dann betrachte $A'_k = A_k - \{a_j\} \cup \{a_m\}$. Dann besteht A'_k aus kompatiblen Aktivitäten und ist auch maximal, denn $|A'_k| = |A_k|$.

■

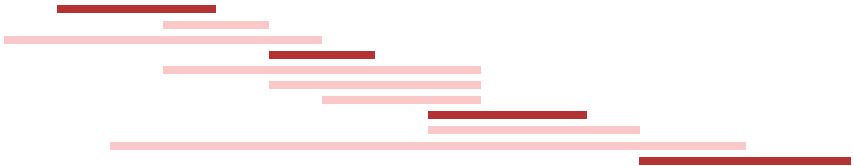
Algorithmus RecursiveActivitySelect(s, f, k, n)

Input: Folge von Start und Endzeiten (s_i, f_i), $1 \leq i \leq n$, $s_i < f_i$, $f_i \leq f_{i+1}$ für alle i. $1 \leq k \leq n$

Output: Maximale Menge kompatibler Aktivitäten.

$m \leftarrow k + 1$
while $m \leq n$ and $s_m \leq f_k$ **do**
    $m \leftarrow m + 1$

if $m \leq n$ **then**
    return $\{a_m\} \cup$ RecursiveActivitySelect(s, f, m, n)
else
    return $\emptyset$



Algorithmus IterativeActivitySelect(s, f, n)

Input: Folge von Start und Endzeiten (s_i, f_i), $1 \leq i \leq n$, $s_i < f_i$, $f_i \leq f_{i+1}$ für alle i.

Output: Maximale Menge kompatibler Aktivitäten.

$A \leftarrow \{a_1\}$
$k \leftarrow 1$
for $m \leftarrow 2$ **to** n **do**
    if $s_m \geq f_k$ **then**
        $A \leftarrow A \cup \{a_m\}$
        $k \leftarrow m$
return A

Laufzeit beider Algorithmen:

Algorithmus IterativeActivitySelect(s, f, n)

Input: Folge von Start und Endzeiten (s_i, f_i), $1 \leq i \leq n$, $s_i < f_i$, $f_i \leq f_{i+1}$ für alle i.

Output: Maximale Menge kompatibler Aktivitäten.

$A \leftarrow \{a_1\}$
$k \leftarrow 1$
for $m \leftarrow 2$ **to** n **do**
    if $s_m \geq f_k$ **then**
        $A \leftarrow A \cup \{a_m\}$
        $k \leftarrow m$
return A

Laufzeit beider Algorithmen: $\Theta(n)$

Klassenproblem

Betrachten Sie den folgenden Satz von Aktivitäten mit ihren jeweiligen Start- und Endzeiten:

Aktivität	Startzeit	Endzeit
A	0	4
B	5	6
C	0	2
D	3	7
E	8	9
F	5	9

Finden Sie die maximale Menge von kompatiblen Aktivitäten, die mit dem gierigen Algorithmus zur Aktivitätsauswahl geplant werden können.

Lösung: Gieriger Algorithmus

1. Sortieren Sie Aktivitäten nach Endzeiten:

$$C \to A \to B \to D \to E \to F$$

2. Initialisieren Sie die Liste der ausgewählten Aktivitäten:

$$\text{Ausgewählt} = \{C\}$$

3. Durchlaufen Sie die verbleibenden Aktivitäten:
   - A ist nicht kompatibel mit C (überspringe A)
   - B ist kompatibel mit C:

$$\text{Ausgewählt} = \{C, B\}$$

   - $\cdots$

4. Die maximale Menge von kompatiblen Aktivitäten ist:

$$\text{Ausgewählt} = \{C, B, E\}$$

Rekursive Problemlösestrategien

Brute Force Enumeration	Backtracking	Divide and Conquer	Dynamic Programming	Greedy

Rekursive Problemlösestrategien

Brute Force Enumeration	**Backtracking**	**Divide and Conquer**	**Dynamic Programming**	**Greedy**
Rekursive Aufzählbarkeit	Prüfbare Randbedingung, Partielle Validierung	Optimale Substruktur	Optimale Substruktur, Überlappende Teilprobleme	Optimale Substruktur, Gierige Auswahl Eigenschaft

Rekursive Problemlösestrategien

Brute Force Enumeration	**Backtracking**	**Divide and Conquer**	**Dynamic Programming**	**Greedy**
Rekursive Aufzählbarkeit	Prüfbare Randbedingung, Partielle Validierung	Optimale Substruktur	Optimale Substruktur, Überlappende Teilprobleme	Optimale Substruktur, Gierige Auswahl Eigenschaft
DFS, BFS, Alle Permutationen, Baumtraversieren	n Damen, Sudoku, m-Färbung, SAT-Solving, naiver TSP	Binäre Suche, Mergesort, Quicksort, Türme von Hanoi, FFT	Bellman Ford, Warshall, Rod-Cutting, LAT, Editierdistanz, Knapsack Problem DP	Dijkstra, Kruskal, Huffmann Coding

Huffmans Idee

Baum Konstruktion von unten nach oben

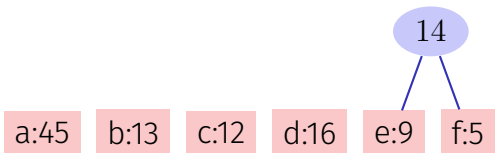
- Starte mit der Menge C der Codewörter

a:45 b:13 c:12 d:16 e:9 f:5

Huffmans Idee

Baum Konstruktion von unten nach oben

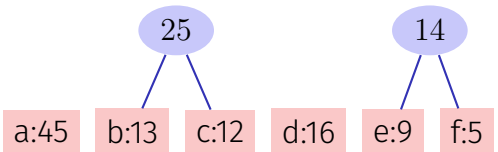
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Huffmans Idee

Baum Konstruktion von unten nach oben

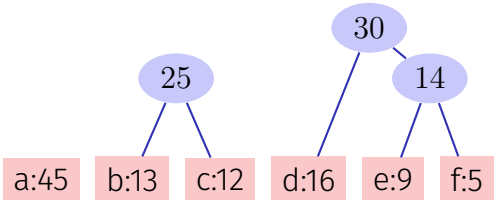
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Huffmans Idee

Baum Konstruktion von unten nach oben

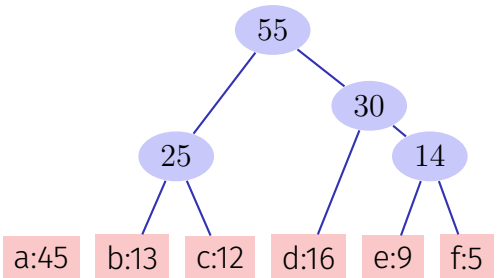
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Huffmans Idee

Baum Konstruktion von unten nach oben

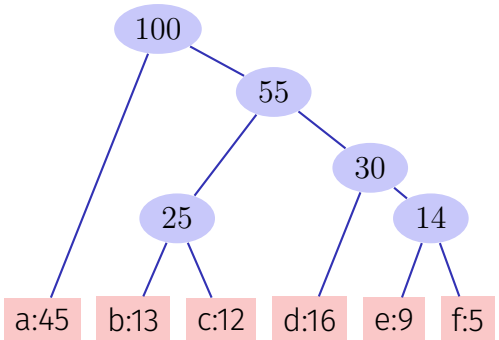
- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Huffmans Idee

Baum Konstruktion von unten nach oben

- Starte mit der Menge C der Codewörter
- Ersetze iterativ die beiden Knoten mit kleinster Häufigkeit durch ihren neuen Vaterknoten.



Algorithmus Huffman(C)

Input: Codewörter $c \in C$
Output: Wurzel eines optimalen Codebaums

$n \leftarrow |C|$
$Q \leftarrow C$
for $i = 1$ **to** $n - 1$ **do**
    Alloziere neuen Knoten z
    z.left $\leftarrow$ ExtractMin(Q) // Extrahiere Wort mit minimaler Häufigkeit.
    z.right $\leftarrow$ ExtractMin(Q)
    z.freq $\leftarrow$ z.left.freq + z.right.freq
    Insert(Q, z)
return ExtractMin(Q)

3. In-Class-Exercise (praktisch)

Vervollständigen Sie die DP Implementation zur Berechnung des optimalen Suchbaumes $\longrightarrow$ CodeExpert

4. Hinweise zu den Aufgaben

Huffman Coding

Huffman Code:

Benutze `std::map` (`#include <map>`)

```
std::map<std::string,int> observations;
// simple access to elements
++observations["cat"];
++observations["mouse"];
++observations["mouse"];

// a map is a collection of std::pair
// show all entries
for (auto x:observations){
  std::cout << "observations of " << x.first << ":" << x.second << std::end
}
```

Huffman Code:

Benutze `std::priority_queue` **(**`#include <queue>`**)**

```
struct MyClass {
    int x;
    MyClass(int X): x{X} {};
};

struct compare{
    bool operator() (const MyClass& a, const MyClass& b){
        return a.x < b.x;
    }
};
//...
std::priority_queue<MyClass, std::vector<MyClass>, compare> q;
q.push(MyClass(10));
```

Huffman Code:

Benutze Smart Pointers `std::shared_ptr` (`#include <memory>`)

```
struct List {
    int value;
    std::shared_ptr<List> next;
    List(std::shared_ptr<List> n, int v): value{v}, next{n} {};
};
...
// automatic memory management, we do not need to care
std::shared_ptr<List> l = std::make_shared<List>(nullptr, 10);
l = std::make_shared<List>(l, 20);
while (l != nullptr){ // output: 20 10
  std::cout << l->value << std::endl;
  l = l->next;
}
```

Huffman Node

```
using SharedNode=std::shared_ptr<Node>;
struct Node{
    char value;
    int frequency;
    SharedNode left;
    SharedNode right;

    // constructor for leafs
    Node(char v, int f): value{v}, frequency{f},
      left{nullptr}, right{nullptr} {}
    // constructor for inner nodes
    Node(SharedNode l, SharedNode r): value{0},
      frequency{l->frequency + r->frequency}, left{l}, right{r} {};
};
```