9. Algorithmenkonstruktion II

9.1 Dynamische Programmierung
Exkurs: Textsuche
9.2 Zufallsgesteuerte Algorithmen
9.3 Geometrische Algorithmen

9.1 Dynamische Programmierung

- Ziel:
  - Beschleunigung der Berechnung rekursiv definierter Funktionen
- Grundidee:
  - speichere Zwischenergebnisse in einer Tabelle, statt sie wiederholt zu berechnen.

- Voraussetzung:
  - optimale Lösung für ein Problem der Größe *n* im Innern ist aus optimalen Teillösungen kleinerer Größe zusammengesetzt. (Bellmansches Optimalitätsprinzip)
- Vorgehensweise
  - Charakterisiere den Lösungsraum und die Struktur einer erwünschten optimalen Lösung
  - Definiere rekursiv den Wert einer optimalen Lösung
  - Berechne den Wert in einer bottom-up Weise
  - Konstruiere eine optimale Lösung aus den berechneten Werten
- Wird typischerweise zur Lösung von Optimierungsproblemen eingesetzt.

9. Algorithmenkonstruktion II

9.1 Dynamische Programmierung
Exkurs: Textsuche
9.2 Zufallsgesteuerte Algorithmen
9.3 Geometrische Algorithmen

9.1 Dynamische Programmierung

- Ziel:
  - Beschleunigung der Berechnung rekursiv definierter Funktionen
- Grundidee:
  - speichere Zwischenergebnisse in einer Tabelle, statt sie wiederholt zu berechnen.

- Voraussetzung:
  - optimale Lösung für ein Problem der Größe *n* im Innern ist aus optimalen Teillösungen kleinerer Größe zusammengesetzt. (Bellmansches Optimalitätsprinzip)
- Vorgehensweise
  - Charakterisiere den Lösungsraum und die Struktur einer erwünschten optimalen Lösung
  - Definiere rekursiv den Wert einer optimalen Lösung
  - Berechne den Wert in einer bottom-up Weise
  - Konstruiere eine optimale Lösung aus den berechneten Werten
- Wird typischerweise zur Lösung von Optimierungsproblemen eingesetzt.

Fibonacci-Zahlen

- Fibonacci-Zahlen
  - fib(0) = 0, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2)
  - direkte Implementierung als rekursive Funktion führt zu exponentiell wachsender Zahl von Aufrufen. Komplexität: $O(2^n)$.
- Lösung mit dynamischer Programmierung
  - verwende 2 Hilfsvariable, in denen fib(n-1) und fib(n-2) gespeichert werden.
  -
    ```
    fib2 = 0;
    fib1 = 1;
    fib = 1;
    i = 2;
    while (i < n) {
     fib2 = fib1;
     fib1 = fib;
     fib = fib1 + fib2;
     i ++;
    }
    ```
  - Schleifeninvarianate?
  - Aufwand?

Pascalsches Dreieck

- Binomialkoeffizienten
  - $\binom{n}{k} = \frac{n!}{k!\,(n-k)!} = \frac{n(n-1)...(n-k+1)}{(n-k)!}$
  - Bei direkter Implementierung benötigt man
    - ⇨ 2(n-k) Multiplikationen und eine Division
  - Rekursionsformel $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$
    - ⇨ als Pascalsches Dreieck bekannt
    - ⇨
      ```
      b = new int [n] [n];
      b[0] [0]= 1;
      for i=1 to n {
           b[i][0] = 1;
           b[i][i] = 1;
           for j=1 to i-1 {
                  b[i][j]=b[i-1][j]+b[i-1][j-1];
           }
      }
      ```
    - ⇨ Aufwand?

Fibonacci-Zahlen

- ❑ Fibonacci-Zahlen
  - fib(0) = 0, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2)
  - direkte Implementierung als rekursive Funktion führt zu exponentiell wachsender Zahl von Aufrufen. Komplexität: O(2^n).
- ❑ Lösung mit dynamischer Programmierung
  - verwende 2 Hilfsvariable, in denen fib(n-1) und fib(n-2) gespeichert werden.
  -
    ```
    fib2 = 0;
    fib1 = 1;
    fib = 1;
    i = 2;
    while (i < n) {
     fib2 = fib1;
     fib1 = fib;
     fib = fib1 + fib2;
     i ++;
    }
    ```
  - Schleifeninvarianate?
  - Aufwand?

Pascalsches Dreieck

- ❑ Binomialkoeffizienten
  - $\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)...(n-k+1)}{(n-k)!}$
  - Bei direkter Implementierung benötigt man
    - ⇨ 2(n-k) Multiplikationen und eine Division
  - Rekursionsformel $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$
    - ⇨ als Pascalsches Dreieck bekannt
    - ⇨
      ```
      b = new int [n] [n];
      b[0] [0]= 1;
      for i=1 to n {
           b[i][0] = 1;
           b[i][i] = 1;
           for j=1 to i-1 {
                  b[i][j]=b[i-1][j]+b[i-1][j-1];
           }
      }
      ```
    - ⇨ Aufwand?

Mehrfache Matrixmultiplikation

- Gegeben
  - n Matrizen unterschiedlicher Größe
- Gesucht
  - Das Produkt dieser n Matrizen
- Fragestellung
  - Wie ist diese Folge von Matrizen am günstigsten zu klammern? (Matrixmultiplikation ist assoziativ)
    - ⇨ Multiplikation von 2 Matrizen pxq und qxr benötigt $p*q*r$ Multiplikationen
- Beispiel
  - 3 Matrizen M_1 (50 x 10), M_2 (10 x 20) und M_3 (20 x 5) sollen multipliziert werden
    - ⇨ $(M_1 * M_2) * M_3$
      ... wieviele Multiplikationen werden benötigt?
    - ⇨ $M_1 * (M_2 * M_3)$
      ... und wieviele Multiplikationen sind es hier?

Optimale Klammerung

- Naiver Algorithmus
  - Ausprobieren aller Klammerungen $K(n)$, um n Matrizen zu klammern

$$K(n) = \sum_{j=1}^{n-1} K(j) \cdot K(n-j), \qquad K(1) = 1$$

  - Lösung dieser Rekursionsgleichung sind die Catalanschen Zahlen $K(n)=C(n-1)$

$$C(n) = \binom{2n}{n} \cdot \frac{1}{n+1} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

  - Aufwand ist also exponentiell
- Optimale Klammerungder Matrizen $M_1, M_2 \ldots M_j$
  - $(M_1 \ldots M_k)(M_{k+1} \ldots M_j)$ ist optimal, wenn die beiden Zwischenprodukte optimal geklammert sind
  - Anzahl der Multiplikationen:

$$m[i,j] = \begin{cases} 0, & i = j \\ \min\limits_{i \le k < j}(m[i,k] + m[k+1,j] + d_{i-1} \cdot d_k \cdot d_j), & i < j \end{cases}$$

  - Algorithmus verwendet ein Array, das allerdings nur in der Form einer Dreiecksmatrix für Indizes $[i,j]$ mit $1 <= i <= j <= n$ verwendet wird.

Mehrfache Matrixmultiplikation

- ❑ Gegeben
  - n Matrizen unterschiedlicher Größe
- ❑ Gesucht
  - Das Produkt dieser n Matrizen
- ❑ Fragestellung
  - Wie ist diese Folge von Matrizen am günstigsten zu klammern? (Matrixmultiplikation ist assoziativ)
    - ➩ Multiplikation von 2 Matrizen pxq und qxr benötigt $p*q*r$ Multiplikationen
- ❑ Beispiel
  - 3 Matrizen M_1 (50 x 10), M_2 (10 x 20) und M_3 (20 x 5) sollen multipliziert werden
    - ➩ $(M_1 * M_2) * M_3$
      ... wieviele Multiplikationen werden benötigt?
    - ➩ $M_1 * (M_2 * M_3)$
      ... und wieviele Multiplikationen sind es hier?

Optimale Klammerung

- ❑ Naiver Algorithmus
  - Ausprobieren aller Klammerungen $K(n)$, um n Matrizen zu klammern

$$K(n) = \sum_{j=1}^{n-1} K(j) \cdot K(n-j), \qquad K(1) = 1$$

  - Lösung dieser Rekursionsgleichung sind die Catalanschen Zahlen $K(n)=C(n-1)$

$$C(n) = \binom{2n}{n} \cdot \frac{1}{n+1} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

  - Aufwand ist also exponentiell
- ❑ Optimale Klammerungder Matrizen M_1, M_2... M_j
  - $(M_1 \ldots M_k)(M_{k+1} \ldots M_j)$ ist optimal, wenn die beiden Zwischenprodukte optimal geklammert sind
  - Anzahl der Multiplikationen:

$$m[i,j] = \begin{cases} 0, & i = j \\ \min\limits_{i \le k < j}(m[i,k] + m[k+1,j] + d_{i-1} \cdot d_k \cdot d_j), & i < j \end{cases}$$

  - Algorithmus verwendet ein Array, das allerdings nur in der Form einer Dreiecksmatrix für Indizes $[i,j]$ mit $1 <= i <= j <= n$ verwendet wird.

Optimale Klammerung

- Algorithmus

```
for i=1 to n { m[i,i] = 0; }
for i=1 to n
 for j=i+1 to n { m[i,j] = maxint; }
for i=1 to n {
 for j=i+1 to n {
       for k=i to j-1 {
             test = m[i,k]+m[k+1,j]+d[i-1]*d[k]*d[j];
             if (test < min) {
                   min = test;
                   kmin[i,j] = k;
             }
       }
       m[i,j] = min;
}}
```

  - Aufwand?

Longest Common Subsequences

- Gegeben
  - zwei Zeichenketten $x = x_1 \ldots x_n$, $y = y_1 \ldots y_m$
- Gesucht
  - die längste Zeichenkette $lcs(x,y) = z = z_1 \ldots z_k$ so, daß z_i die sowohl in x als auch in y in der selben Reihenfolge vorkommen
- Beispiel
  - diff unter Unix. Behandelt ganze Zeilen als einfache Zeichen (mittels einer Hashfunktion).
  - x = abcbdab, y= bdcaba

    a b c b d a b

    b d c a b a

    $lcs(x,y)$ = ?
  - $| lcs(x,y) |$ ist durch $\min\{|x|, |y|\}$ beschränkt
- Naiver Ansatz
  - Bestimmen aller common subsequences
    - ⇨ Aufwand $O(2^n)$

Optimale Klammerung

- Algorithmus

  -
    ```
    for i=1 to n { m[i,i] = 0; }
    for i=1 to n
     for j=i+1 to n { m[i,j] = maxint; }
    for i=1 to n {
     for j=i+1 to n {
            for k=i to j-1 {
                    test = m[i,k]+m[k+1,j]+d[i-1]*d[k]*d[j];
                    if (test < min) {
                            min = test;
                            kmin[i,j] = k;
                    }
            }
            m[i,j] = min;
    }}
    ```
  - Aufwand?

Longest Common Subsequences

- Gegeben
  - zwei Zeichenketten $x = x_1 \ldots x_n$, $y = y_1 \ldots y_m$
- Gesucht
  - die längste Zeichenkette $lcs(x,y) = z = z_1 \ldots z_k$ so, daß z_i die sowohl in x als auch in y in der selben Reihenfolge vorkommen
- Beispiel
  - diff unter Unix. Behandelt ganze Zeilen als einfache Zeichen (mittels einer Hashfunktion).
  - x = abcbdab, y= bdcaba

    a b c b d a b

    b d c a b a

    $lcs(x,y) = ?$
  - $| lcs(x,y) |$ ist durch $\min\{|x|, |y|\}$ beschränkt
- Naiver Ansatz
  - Bestimmen aller common subsequences
    - ⇨ Aufwand $O(2^n)$

Longest Common Subsequence

- ❑ Folgende rekursive Formel gilt
  - $$L[i,j] = \begin{cases} 0 & \textit{falls } \mathrm{i = 0 \text{ oder } j = 0} \\ \mathrm{c[i-1,\ j-1]+1} & \textit{falls } \mathrm{i,j > 0 \text{ und } x_i = y_j} \\ \max\{\mathrm{c[i,j-1], c[i-1,j]}\} & \mathrm{falls\ i,j > 0 \text{ und } x_i \neq} y_j \end{cases}$$
- ❑ Algorithmus

  -
    ```
    for i=1 to n { L[i,0] = 0; }
    for i=1 to m { L[0,m] = 0; }
    for i=1 to m
     for j=1 to n {
            if (x[i] == y[j]) { L[i,j] = L[i-1,j-1]+1; }
              else { L[i,j]= max(L[i--1,j],L[i,j-1]); }
       }
    ```
  - Wie sieht die Matrix für das Beispiel aus?
  - Aufwand?

  - Falls *lcs* selbst benötigt wird, muß man Zeiger in die Matrix einbauen, die die Indexpaare miteinander verketten.

Exkurs: Textsuche

- ❑ Häufig werden bei der Textsuche Algorithmen mit Vorberechnung verwendet:
  - Teile der Eingabedaten werden aufbereitet, bevor man die eigentliche Aufgabe in Angriff nimmt.
  - Vorberechnungen sind Beispiele für dynamische Programmierung
- ❑ Textsuche
  - Gegeben
    - ⇨ Text: *T* [1..*n*]
    - ⇨ Pattern (Muster): *P*[1..*m*], mit *m*<= *n*
  - Gesucht:
    - ⇨ Vorkommen von *P* in *T*
- ❑ Naiver Algorithmus

  -
    ```
    res = 1; i = 1;
    while ((res==-1) && (i<=n-m)) {
     j = 1;
     while ((j<=m) && (T[i+j-1]==P[j]) {j++;}
     if (j>m) {res = i;}
            else {i++;}
    }
    ```
  - Komplexität ?

Longest Common Subsequence

- Folgende rekursive Formel gilt
  - $$L[i,j] = \begin{cases} 0 & \textit{falls}\ \mathrm{i} = 0 \text{ oder } \mathrm{j} = 0 \\ \mathrm{c[i-1,\ j-1]+1} & \textit{falls}\ \mathrm{i,j} > 0 \text{ und } \mathrm{x_i} = \mathrm{y_j} \\ \max\{\mathrm{c[i,j-1],c[i-1,j]}\} & \text{falls } \mathrm{i,j} > 0 \text{ und } \mathrm{x_i} \neq y_j \end{cases}$$
- Algorithmus

  -
    ```
    for i=1 to n { L[i,0] = 0; }
    for i=1 to m { L[0,m] = 0; }
    for i=1 to m
     for j=1 to n {
          if (x[i] == y[j]) { L[i,j] = L[i-1,j-1]+1; }
            else { L[i,j]= max(L[i--1,j],L[i,j-1]); }
      }
    ```
  - Wie sieht die Matrix für das Beispiel aus?
  - Aufwand?

  - Falls *lcs* selbst benötigt wird, muß man Zeiger in die Matrix einbauen, die die Indexpaare miteinander verketten.

Exkurs: Textsuche

- Häufig werden bei der Textsuche Algorithmen mit Vorberechnung verwendet:
  - Teile der Eingabedaten werden aufbereitet, bevor man die eigentliche Aufgabe in Angriff nimmt.
  - Vorberechnungen sind Beispiele für dynamische Programmierung
- Textsuche
  - Gegeben
    - ⇨ Text: *T* [1..*n*]
    - ⇨ Pattern (Muster): *P*[1..*m*], mit *m*<= *n*
  - Gesucht:
    - ⇨ Vorkommen von *P* in *T*
- Naiver Algorithmus

  -
    ```
    res = 1; i = 1;
    while ((res==-1) && (i<=n-m)) {
     j = 1;
     while ((j<=m) && (T[i+j-1]==P[j]) {j++;}
     if (j>m) {res = i;}
          else {i++;}
    }
    ```
  - Komplexität ?

Knuth-Morris-Pratt Algorithmus

- Beobachtung
  - Was sorgt für Ineffizienz beim naiven Algorithmus?
- Annahme
  - Suche startet an Position i und schlägt an Position $i+j-1$ fehl
    - ⇨ $T_i T_{i+1} \ldots T_{i+j-2} = P_1 P_2 \ldots P_j$
    - ⇨ $T_{i+j-1} <> P_j$
  - Es gibt
    - ⇨ q, $1 < q < j$ mit $T_{i+j-2-q} T_{i+j-1-q} \ldots T_{i+j-2} = P_1 P_2 \ldots P_q$ genau dann wenn $P_1 P_2 \ldots P_q = P_{j-q} P_{j-q+1} \ldots P_{j-1}$
- Resultat
  - Kennt man ein maximales q in Abhängigkeit von j, dann kann man sofort mit $i' = i+j-1-q$ weitermachen

---

Beispiel

- Beispiel
  - T = abrakadabraabracadabra
    P = abracadabra
  - Welcher Vergleich schlägt fehl?
  - Bei welcher Position kann man weitersuchen und weshalb?
  - Nächster Schritt?
  - Nächster Fehschlag?
  - Wie wird weitergesucht?
- Beobachtung
  - q hängt ausschließlich von P und j ab, nicht von T.
  - Folgende Tabelle kann im Voraus berechnet werden:

⇨ j:	1	2	3	4	5	6	7	8	9	10	11
P:	a	b	r	a	c	a	d	a	b	r	a
q:											

Knuth-Morris-Pratt Algorithmus

- ❑ Beobachtung
  - Was sorgt für Ineffizienz beim naiven Algorithmus?
- ❑ Annahme
  - Suche startet an Position i und schlägt an Position $i+j-1$ fehl
    - ⇨ $T_i T_{i+1} \ldots T_{i+j-2} = P_1 P_2 \ldots P_j$
    - ⇨ $T_{i+j-1} <> P_j$
  - Es gibt
    - ⇨ q, $1 < q < j$ mit $T_{i+j-2-q} T_{i+j-1-q} \ldots T_{i+j-2} = P_1 P_2 \ldots P_q$ genau dann wenn $P_1 P_2 \ldots P_q = P_{j-q} P_{j-q+1} \ldots P_{j-1}$
- ❑ Resultat
  - Kennt man ein maximales q in Abhängigkeit von j, dann kann man sofort mit $i´= i+j-1-q$ weitermachen

---

Beispiel

- ❑ Beispiel
  - T = abrakadabraabracadabra
    P = abracadabra
  - Welcher Vergleich schlägt fehl?
  - Bei welcher Position kann man weitersuchen und weshalb?
  - Nächster Schritt?
  - Nächster Fehschlag?
  - Wie wird weitergesucht?
- ❑ Beobachtung
  - q hängt ausschließlich von P und j ab, nicht von T.
  - Folgende Tabelle kann im Voraus berechnet werden:

⇨ j:	1	2	3	4	5	6	7	8	9	10	11
P:	a	b	r	a	c	a	d	a	b	r	a
q:											

Algorithmus

- Algorithmus
  -
  - Aufwand ?

Vorberechnung

- Verschiebetafel muß berechnet werden $\Pi[1..m]$ mit
  - $\Pi[q] = \max\{k<q \mid P[1..k] = P[q-k+1..q]\}$ $(q=1, ..., m)$
- Algorithmus zur Berechnung der Verschiebetafel
  -

```
q = 0;
for i=1 to n {
 while ((q>0) && (P[q+1] m<> T[i])) { q = Π[q]; }
 if (P[q+1] == T [i]) { q ++; }
 if ( q==m) {
        q = Π[q];
        return i-m;}
}
```

  - Aufwand ?
  - Bei welchen Mustern arbeitet der Algorithmus am besten?

Algorithmus

- Algorithmus
  -
  - Aufwand ?

Vorberechnung

- Verschiebetafel muß berechnet werden $\Pi[1..m]$ mit
  - $\Pi[q] = \max\{k<q \mid P[1..k] = P[q-k+1..q]\}$ $(q=1, ..., m)$
- Algorithmus zur Berechnung der Verschiebetafel
  -
    ```
    q = 0;
    for i=1 to n {
     while ((q>0) && (P[q+1] m<> T[i])) { q = Π[q]; }
     if (P[q+1] == T [i]) { q ++; }
     if ( q==m) {
            q = Π[q];
            return i-m;}
    }
    ```
  - Aufwand ?
  - Bei welchen Mustern arbeitet der Algorithmus am besten?

Boyer-Moore-Algorithmus

- Idee
  - vergleichbar mit Knuth-Morris-Pratt Algorithmus
  - das Muster wird entlang des Textes geschoben und bei einem Mismatch möglichst weit weitergeschoben
  - Vergleiche erfolgen aber von rechts nach links
    - Verschiebungen können bis zur Größenordnung der Patternlänge durchgeführt werden
  - Komplexität:
    - O(n/m) plus Initialisierungsphase zur Berechnung der Verschiebetafel
    - ist in der Praxis der schnellste Algorithmus zur Textsuche
- Verschiebung
  - "bad character"-Strategie
    - bei einem Mismatch wird bis zu dem Zeichen des Mismatches in der zu durchsuchenden Zeichenfolge weitergeschoben
  - "good suffix"-Strategie
    - es wird nach der am weitesten rechts stehenden gesucht und das Pattern kann soweit verschoben werden, bis die Textabschnitte zur Deckung kommen.
  - es wird jeweils diejenige Strategie mit der größten Verschiebung verwendet.
- Komplexität
  - Worst-Case-Komplexität ist O(nm), wie beim naiven Algorithmus
  - wann tritt dieses zum Beispiel ein?

Boyer-Moore-Algorithmus

- ❏ Idee
  - vergleichbar mit Knuth-Morris-Pratt Algorithmus
  - das Muster wird entlang des Textes geschoben und bei einem Mismatch möglichst weit weitergeschoben
  - Vergleiche erfolgen aber von rechts nach links
    - ⇨ Verschiebungen können bis zur Größenordnung der Patternlänge durchgeführt werden
  - Komplexität:
    - ⇨ O(n/m) plus Initialisierungsphase zur Berechnung der Verschiebetafel
    - ⇨ ist in der Praxis der schnellste Algorithmus zur Textsuche
- ❏ Verschiebung
  - "bad character"-Strategie
    - ⇨ bei einem Mismatch wird bis zu dem Zeichen des Mismatches in der zu durchsuchenden Zeichenfolge weitergeschoben
  - "good suffix"-Strategie
    - ⇨ es wird nach der am weitesten rechts stehenden gesucht und das Pattern kann soweit verschoben werden, bis die Textabschnitte zur Deckung kommen.
  - es wird jeweils diejenige Strategie mit der größten Verschiebung verwendet.
- ❏ Komplexität
  - Worst-Case-Komplexität ist O(nm), wie beim naiven Algorithmus
  - wann tritt dieses zum Beispiel ein?