2.2 Entwurfsparadigmen

- Top-down
- Bottom-up
- Divide & Conquer
- Dynamisches Programmieren
- Caching (Memoization)
- Branch-and-Bound
- Greedy

Top-Down

- Zerlege das gegebene Problem in Teilschritte
- Zerlege Teilschritte in Sub-Teilschritte
- Zerlege Sub-Teilschritte in Sub-Sub-Teilschritte
- usw.

Top-Down

- Bei der Implementierung eines Teilschrittes werden die Sub-Teilschritte als Black-Boxes verwendet (Spezifikation!)
- Betrachte die Teilschritte in der Reihenfolge der geringsten Querbezüge
- Vermeide Seiteneffekte

2.2 Entwurfsparadigmen

- Top-down
- Bottom-up
- Divide & Conquer
- Dynamisches Programmieren
- Caching (Memoization)
- Branch-and-Bound
- Greedy

Bottom-up

- Löse Teilprobleme und kombiniere diese zu einer Gesamtlösung
- Möglich selbst bei unvollständiger Spezifikation
- Bessere Test-Suites parallel zur Entwicklung (Entwurf & Implementierung)
- Code reuse !?

2.2 Entwurfsparadigmen

- Top-down
- Bottom-up
- Divide & Conquer
- Dynamisches Programmieren
- Caching (Memoization)
- Branch-and-Bound
- Greedy

Divide & Conquer

- Geg.: Problem der Größe n
- Divide: Zerlege das Problem rekursiv in k Teilprobleme der Größe n/k
- Conquer: Löse kleine Probleme direkt
- Kombiniere die Teillösungen zur Gesamtlösung

Divide & Conquer

- Meistens: k = 2
- Rekursive Implementierung
- Intuitiv leicht verständlich (eigentlicher Algorithmus in den Prozeduraufrufen versteckt)
- Aufwand durch Rekursionsgleichungen beschrieben (Master-Theorem)

Find MinMax iterativ

- MinMax(A[1..n])

```
min ← 1; max ← 1; i ← 2
while i ⩽ n do
    if A[ i ] < A[min] then
        min ← i
    if A[ i ] > A[max] then
        max ← i
    i ← i+1
return (A[min],A[max])
```

- T(n) = 2×n−2 … (= 3×n−3)

Find MinMax D&C

- MinMax(A[a..b])

```
if b−a ⩽ 1 then
    return (Min(A[a],A[b]),Max(A[a],A[b]))
else
    (u1,v1) ← MinMax(A[a...(a+b)/2])
    (u2,v2) ← MinMax(A[(a+b)/2+1...b])
    return (Min(u1,u2),Max(v1,v2))
```

Find MinMax D&C

- $T(n) = \begin{cases} 1 & n = 2 \\ 2 \times T(n/2) + 2 & n > 2 \end{cases}$

- $T(n) = 2 \times T(n/2) + 2$

  $= 4 \times T(n/4) + 4 + 2$

  $= \ldots$

  $= n/2 + n/2 + \ldots + 4 + 2 = 3 \times n/2 - 2$

2.2 Entwurfsparadigmen

- Top-down
- Bottom-up
- Divide & Conquer
- Dynamisches Programmieren
- Caching (Memoization)
- Branch-and-Bound
- Greedy

Dynamisches Programmieren

- Reduziere gegebenes Problem auf kleinere Teilprobleme
- Löse Teilprobleme
- Speichere Teillösungen in Tabelle
- Kombiniere Teillösungen zur Gesamtlösung

Dynamisches Programmieren

- Typisches Muster …
  - Gesucht: beste Lösung $L_{1,n}$ von 1 bis n
  - Bestimme: optimale Teillösungen $L_{1,k}$ und $L_{k+1,n}$ für $k=1,\ldots,n-1$
  - Finde: beste Kombination $L_{1,k}$, $L_{k+1,n}$

Dynamisches Programmieren

- Im Unterschied zu Divide & Conquer wird keine top-down Zerlegung, sondern eine bottom-up Kombination durchgeführt.
  - Iterative Implementierung
  - Speichere Zwischenergebnisse in einer Tabelle und vermeide dadurch doppelte Berechnungen

Beispiel

- Matrizenmultiplikation
- $A = [a_{ij}] \in R^{p \times q}, \quad B = [b_{ik}] \in R^{q \times r}$
- $C = [c_{ik}] = A \times B \in R^{p \times r}$
- $c_{ik} = \sum_j a_{ij} \times b_{jk}$
- Zur Berechnung von C sind $p \times q \times r$ skalare Multiplikationen notwendig

Beispiel

- Matrizenmultiplikation
- $A \in R^{p \times q}$, $B \in R^{q \times r}$, $C \in R^{r \times s}$
- $D = (A \times B) \times C = A \times (B \times C)$
- $(A \times B) \times C$ ... $p \times q \times r + p \times r \times s$ Multiplikationen
- $A \times (B \times C)$ ... $q \times r \times s + p \times q \times s$ Multiplikationen

Beispiel

- Matrizenmultiplikation
- $A \in R^{10\times1}$, $B \in R^{1\times10}$, $C \in R^{10\times1}$
- $D = (A\times B)\times C = A\times(B\times C)$
- $(A\times B)\times C$ ... $10\times1\times10 + 10\times10\times1 = 200$
- $A\times(B\times C)$ ... $1\times10\times1 + 10\times1\times1 = 20$

Beispiel

- Matrizenmultiplikation
- $A_i \in R^{r[i-1] \times r[i]}, \quad i=1,\ldots,n$
- $M_{1,n}$: minimale Anzahl von skalaren Multiplikationen zur Berechnung von $B = A_1 \times A_2 \times \ldots \times A_n$
- $M_{i,j}$: Zahl der Multiplikationen zur Berechnung von $A_i \times \ldots \times A_j$

Beispiel

- Matrizenmultiplikation
- $M_{1,n} = \min_{1 \leqslant k < n}(M_{1,k} + M_{k+1,n} + r_0 \times r_k \times r_n)$
- $M_{i,j} = \min_{i \leqslant k < j} (M_{i,k} + M_{k+1,j} + r_{i-1} \times r_k \times r_j)$
- Speichere die $M_{i,j}$ in einer Tabelle
- Basis-Fall: $M_{i,i} = 0$
- Berechne sukzessive $M_{i,i+k}$ für $k=1, 2, \ldots$

MatrixMultiply()

- MatrixMultiply(r[0..n])

```
for i←1 to n do
    M[i,i] ← 0
for k←1 to n−1 do
    for i←1 to n−k do
        min ← i
        val ← M[i+1,i+k] + r[i−1]×r[i]×r[i+k]
        for j←i+1 to i+k−1 do
            if M[i,j] + M[j+1,i+k]+r[i−1]×r[j]×r[i+k] < val then
                min ← j
                val  ← M[i,j] + M[j+1,i+k]+ r[i−1]×r[j]×r[i+k]
        S[i,i+k] ← min
        M[i,i+k] ← val
```

MatrixMultiply()

- MatrixMultiply(r[0..n])

```
for i←1 to n do
    M[i,i] ← 0
for k←1 to n−1 do
    for i←1 to n−k do
        min ← i
        val ← M[i+1,i+k] + r[i−1]×r[i]×r[i+k]
        for j←i+1 to i+k−1 do
            if M[i,j] + M[j+1,i+k]+r[i−1]×r[j]×r[i+k] < val then
                min ← j
                val  ← M[i,j] + M[j+1,i+k]+ r[i−1]×r[j]×r[i+k]
        S[i,i+k] ← min
        M[i,i+k] ← val
```

Initialisierung

MatrixMultiply()

- MatrixMultiply(r[0..n])

```
for i←1 to n do
    M[i,i] ← 0
for k←1 to n−1 do        Längenindex
    for i←1 to n−k do
        min ← i
        val ← M[i+1,i+k] + r[i−1]×r[i]×r[i+k]
        for j←i+1 to i+k−1 do
            if M[i,j] + M[j+1,i+k]+r[i−1]×r[j]×r[i+k] < val then
                min ← j
                val  ← M[i,j] + M[j+1,i+k]+ r[i−1]×r[j]×r[i+k]
        S[i,i+k] ← min
        M[i,i+k] ← val
```

MatrixMultiply()

- MatrixMultiply(r[0..n])

```
for i←1 to n do
    M[i,i] ← 0
for k←1 to n−1 do
    for i←1 to n−k do        Berechne M[i,i+k]
        min ← i
        val ← M[i+1,i+k] + r[i−1]×r[i]×r[i+k]
        for j←i+1 to i+k−1 do
            if M[i,j] + M[j+1,i+k]+r[i−1]×r[j]×r[i+k] < val then
                min ← j
                val  ← M[i,j] + M[j+1,i+k]+ r[i−1]×r[j]×r[i+k]
        S[i,i+k] ← min
        M[i,i+k] ← val
```

Rechenaufwand

- Innerste Schleife:

  for j←i+1 to i+k−1 do

  → $O(i+k-1-i-1) = O(k-2) = O(k)$

- Mittlere Schleife:

  for i←1 to n−k do

  → $O((n-k)\times O(k)) = O(nk-k^2) = O(nk)$

- Äußere Schleife:

  for k←1 to n−1 do

  → $O(n\times O(nk)) = O(n^3)$

2.2 Entwurfsparadigmen

- Top-down
- Bottom-up
- Divide & Conquer
- Dynamisches Programmieren
- Caching (Memoization)
- Branch-and-Bound
- Greedy

Memoization

- Effizienzsteigerung beim dynamischen Programmieren durch Speichern der Zwischenergebnisse in einer Tabelle
  → vermeide doppelte Berechnungen
- Allgemeines Konzept
  (lohnt sich, wenn der Tabellenzugriff schneller als die Neuberechnung ist)

Fibonacci (zum letzten Mal)

- MemoFibonacci(n)

```
new F[0..n]
for i←0 to n do
    F[ i ] ← −1
Fibo(F,n)
```

- Fibo(F,n)

```
if F[n] < 0 then
    if n > 1 then
        F[n] ← Fibo(F,n−1) + Fibo(F,n−2)
    else
        F[n] ← n
return F[n]
```

2.2 Entwurfsparadigmen

- Top-down
- Bottom-up
- Divide & Conquer
- Dynamisches Programmieren
- Caching (Memoization)
- Branch-and-Bound
- Greedy

Branch-and-Bound

- Anwendung bei Suche einer optimalen Lösung in einer unendlichen oder sehr grossen diskreten Menge
- Teile Problemmenge in zwei oder mehr Teile auf (Branching), es ergibt sich der Branching-Baum
  - Last In – First Out ⇔ Tiefensuche
  - First In – First Out ⇔ Breitensuche

Branch-and-Bound

- Erkenne suboptimale Lösungen frühzeitig durch untere/obere Schranken (Bounds)
  - z.B. kann nach dem Finden einer zulässigen Lösung diese eine Schranke darstellen
- Beispiel: Finden der nächsten Nachbarn

Erfüllbarkeit Boolescher Formeln

- Gegeben eine Boolesche Formel mit Variablen X1, X2, ... sowie ∧, ∨ und ¬
- Gibt es eine Belegung der Xi mit wahr oder falsch, so dass die Formel wahr ist?
- Beispiel: X1 ∧ (¬X2 ∨ ¬X1)
- Branching anhand des Wertes einer Variable (also Aufteilung in zwei Mengen)
- Nicht-zulässige Kombinationen beschränken den Suchraum
- Wichtig: Geschickte Wahl der Variablen!

2.2 Entwurfsparadigmen

- Top-down
- Bottom-up
- Divide & Conquer
- Dynamisches Programmieren
- Caching (Memoization)
- Branch-and-Bound
- Greedy

Greedy

- Wähle heuristisch den aus aktueller Sicht “besten” Lösungsweg
- Führt im Allgemeinen nur zu einer lokal optimalen Lösng
- In Spezialfällen kann Optimalität garantiert werden
- Beispiel: Rucksackproblem
  - optimal für “Schüttgut” (fractional knapsack)
  - nur lokal optimal für “Stückgut” (0-1 knapsack)

Fractional Knapsack

- Greedy kann Bruchteile auswählen
- Gegeben sind drei Teile mit folgenden Größen: A (3 units), B (5 units), C (7 units)
- Die Kapazität des Rucksacks beträgt 8 units
- Greedy wählt zuerst Teil C aus.
- Die Restkapazität beträgt 1 unit.
- Greedy wählt 1/5 von B aus.
- Der Rucksack ist optimal gefüllt.

0-1 Knapsack

- Greedy kann nur ganze Teile auswählen
- Gegeben sind drei Teile mit folgenden Größen: A (3 units), B (5 units), C (7 units)
- Die Kapazität des Rucksacks beträgt 8 units
- Greedy wählt zuerst Teil C aus.
- Die Restkapazität beträgt 1 unit.
- Greedy kann kein Teil mehr auswählen.
- Der Rucksack wird nicht optimal gefüllt.

2.2 Entwurfsparadigmen

- Top-down
- Bottom-up
- Divide & Conquer
- Dynamisches Programmieren
- Caching (Memoization)
- Branch-and-Bound
- Greedy