4 Entwurf von Algorithmen

Lernziele

- Welche Entwurfsverfahren gibt es (grob)?
- Beispiele zu den einzelnen Verfahren

Verfahren

- Vollständige Enumeration
- Gierige Verfahren (Greedy-Verfahren)
- Teile und Beherrsche (Divide & Conquer, divide et impera)
- Dynamische Programmierung
- Backtracking (im nachfolgenden Kapitel bei Rekursion)
- Andere (in späteren Vorlesungen)

Vollständige Enumeration

Prinzip

- Systematische Erzeugung aller möglichen Lösungen. Danach Auswahl einer optimalen Lösung.
- Anwendbar bei allen Problemen, bei denen es mehrere Lösungsalternativen gibt, von denen eine ausgewählt werden soll.
  Oft: Optimierungsprobleme
- Verfahren haben meist ungünstige Rechenzeitkomplexität (nicht polynomiell).

Vollständige Enumeration

Beispiel: Travelling Salesman Problem (TSP)

Eingabe:

n Städte 1, 2, … , n

Entfernungen d_{ij} zwischen den Städten i und j.

Ausgabe:

Eine Permutation π von (1, 2, …, n) mit

$$\sum_{i=1}^{n} d_{\pi(i)\pi((i+1) \bmod n)} = min$$

Man erhält die günstigste Permutation sicher, wenn man alle Permutationen untersucht.

Vollständige Enumeration

Beispiel: Travelling Salesman Problem (TSP)

Ist das eine effiziente Möglichkeit?

Antwort: Nein!
n! viele Permutationen! Die Rechenzeitkomplexität ist also $T(n) \in O(n!)$

Bei nur 20 Städten müsste man 770 Jahrhunderte rechnen, wenn man pro Sekunde 1 Mio. Permutationen testen könnte!

Vollständige Enumeration

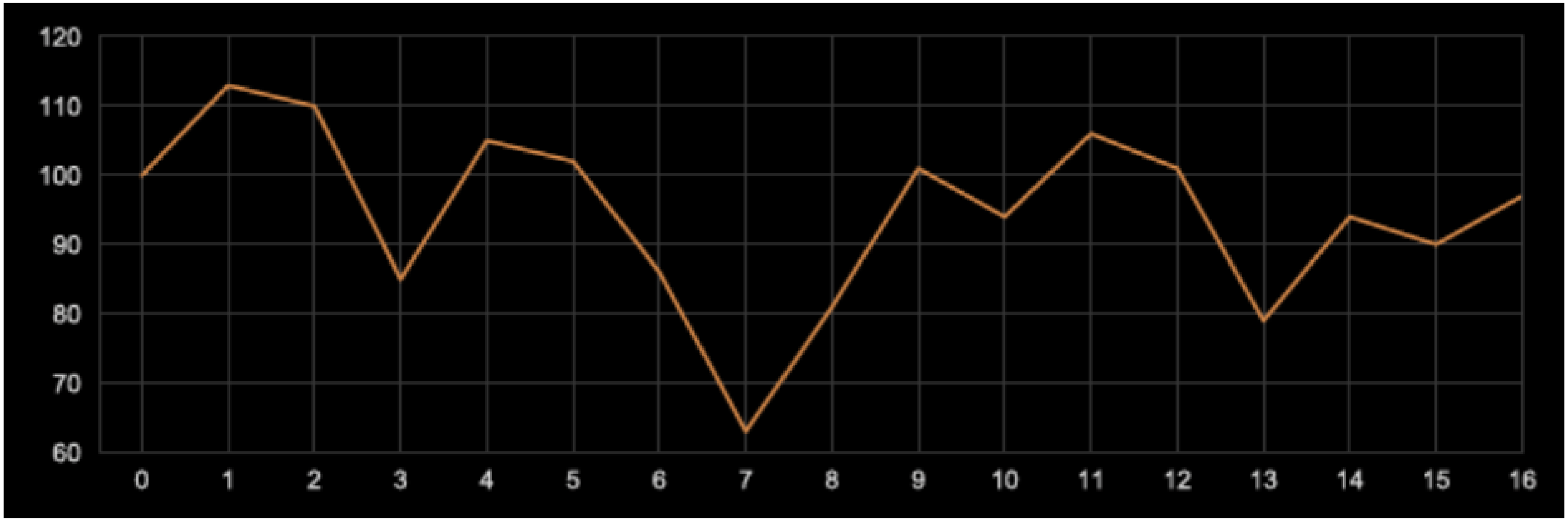
Anwendungsbeispiel: Maximum-Subarray-Problem

Tag	0	1	2	3	4	5	6	7	8	…
Aktienkurs	100	113	110	85	105	102	86	63	81	…

- Sie haben einen Verlauf von Aktienkursen.
- Was wäre der günstigste Kauf- bzw. Verkaufszeitpunkt (leider im Nachhinein!)

Vollständige Enumeration

Anwendungsbeispiel: Maximum-Subarray-Problem



- Sie haben einen Verlauf von Aktienkursen.
- Was wäre der günstigste Kauf- bzw. Verkaufszeitpunkt (leider im Nachhinein!)

Vollständige Enumeration

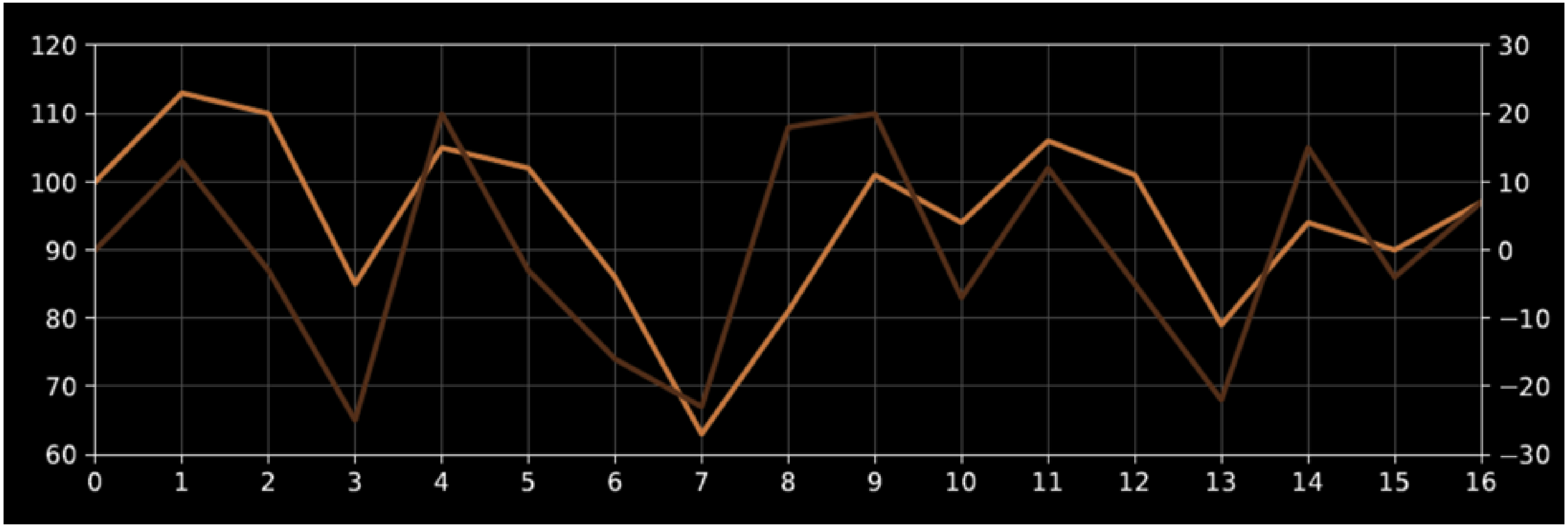
Anwendungsbeispiel: Maximum-Subarray-Problem

Tag	0	1	2	3	4	5	6	7	8	…
Aktienkurs	100	113	110	85	105	102	86	63	81	…
Veränderung		+13	-3	-25	+20	-3	-16	-23	+18	…

- Minimum bis Maximum der Sequenz.
- Transformation: Untersequenz mit der größten positiven Summe.

Vollständige Enumeration

Anwendungsbeispiel: Maximum-Subarray-Problem



- Minimum bis Maximum der Sequenz.
- Transformation: Untersequenz mit der größten positiven Summe.

Vollständige Enumeration

Anwendungsbeispiel: Maximum-Subarray-Problem

- In einer Folge A von n ganzen Zahlen wird eine zusammenhängende Teilfolge gesucht, deren Summe unter allen zusammenhängenden Teilfolgen maximal ist.
- Algorithmus zunächst nach dem Prinzip der vollständigen Enumeration, also der Berechnung sämtlicher Teilfolgen

Vollständige Enumeration

Anwendungsbeispiel: Maximum-Subarray-Problem

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

- Welches ist die maximale Teilfolge?

Hochschule Aalen

Vollständige Enumeration

Anwendungsbeispiel: Maximum-Subarray-Problem

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

- Welches ist die maximale Teilfolge?

Vollständige Enumeration

Allgemein: Wie viele Teilfolgen gibt es?

Vollständige Enumeration

```
function MAXIMUMSUBARRAY_V1(A, left, right)
    for l = left to right do
        for r = l to right do
            sum = 0
            for i = l to r do
                sum = sum + A[i]
            end for
            if sum > max_sum then
                max_sum = sum
                Merke l und r
            end if
        end for
    end for
end function
```

Vollständige Enumeration

Anwendungsbeispiel: Maximum-Subarray-Problem

- Wie ist die Komplexität dieses Algorithmus?
- Welche Schwächen gibt es?

Vollständige Enumeration

```
function MAXIMUMSUBARRAY_V2 (A, left, right)
    max_sum = A[left]
        for l = left to right do
            sum = 0
            for r = l to right do
                sum = sum + A[r]
                if sum > max_sum then
                    max_sum = sum
                    Merke l und r
                end if
            end for
        end for

end function
```

Vollständige Enumeration

Zusammenfassung

Vorteil:

- Günstigste Lösung wird garantiert gefunden.

Nachteil:

- Verfahren haben meist ungünstige Rechenzeitkomplexität (nicht polynomiell)

Greedy-Verfahren

Prinzip

„Gierige“ Algorithmen wählen den zum aktuellen Zeitpunkt besten Folgezustand

- In jedem Schritt wird die im Moment (lokal) optimale Wahl getroffen
- Dadurch ergibt sich oftmals eine sehr gute Laufzeit
- Allerdings ist das Ergebnis global meist nicht optimal
- Dennoch lohnen sich Greedy-Verfahren, wenn die Laufzeit zur Erreichung des globalen Optimismus sehr ungünstig ist (z.B. knapsack, travelling salesman)

Anwendung: Optimierungsprobleme

Greedy-Verfahren

Beispiel: Münzwechseln

Gegeben: Ein Betrag W an Wechselgeld sowie eine Reihe B von Münzwerten.

Gesucht: Eine Folge von Münzwerten möglichst kurzer Länge mit Gesamtwert W.

Beispiel:

W = 98, Münzwerte 50, 20, 10, 5, 2, 1

Eine Lösung ist: 50, 20, 20, 5, 2, 1

Greedy-Verfahren: Beispiel Münzwechsel

```
function MUENZWECHSELN(W, B)
    while W ≠ 0 do
        b = SUCHEGROESSTEMUENZE(B, W)
        ZAHLEAUS(b)
        W = W - b
    end while
end function
```

Greedy-Verfahren: Beispiel Münzwechsel

Problem: Gefundene Lösung ist nicht immer optimal.

Beispiel: W = 60, B = 1, 20, 41

Algorithmus liefert: 41, 19x1

Optimal wäre: 3x20

Greedy-Verfahren

Zusammenfassung

Vorteil:

- Findet schnell eine mögliche Lösung des Problems

Nachteil:

- Lösung des Problems in vielen Fällen nicht optimal

Teile und Beherrsche

Prinzip:

Teile ein Problem so lange in Teilprobleme auf, bis diese trivial, offensichtlich oder zumindest einfach lösbar sind.

Methode A zur Lösung eines Problems P der Größe n:

1. Direkte Lösung: Falls $n < n_0$: Löse das Problem P direkt
2. Divide: Teile P in Teilprobleme $P_1, \ldots P_k$ mit $k \geq 2$
3. Conquer: Löse jedes der Teilprobleme P_i mit der Methode A
4. Merge: Setze die Teillösungen zusammen

Teile und Beherrsche

Beispiel: Austeilen von Übungsblättern

Variante 1

- Jeder nimmt ein Blatt und gibt den Rest des Stapels an seinen Nachbarn weiter
- Aufwand?

Teile und Beherrsche

Beispiel: Austeilen von Übungsblättern

Variante 1

- Jeder nimmt ein Blatt und gibt den Rest des Stapels an seinen Nachbarn weiter
- Aufwand: O(n)

Teile und Beherrsche

Beispiel: Austeilen von Übungsblättern

Variante 1

- Jeder nimmt ein Blatt und gibt den Rest des Stapels an seinen Nachbarn weiter
- Aufwand: O(n)

Variante 2

- Jeder nimmt ein Blatt und gibt jeweils die Hälfte des Restes an zwei Nachbarn weiter
- Aufwand?

Teile und Beherrsche

Beispiel: Austeilen von Übungsblättern

Variante 1

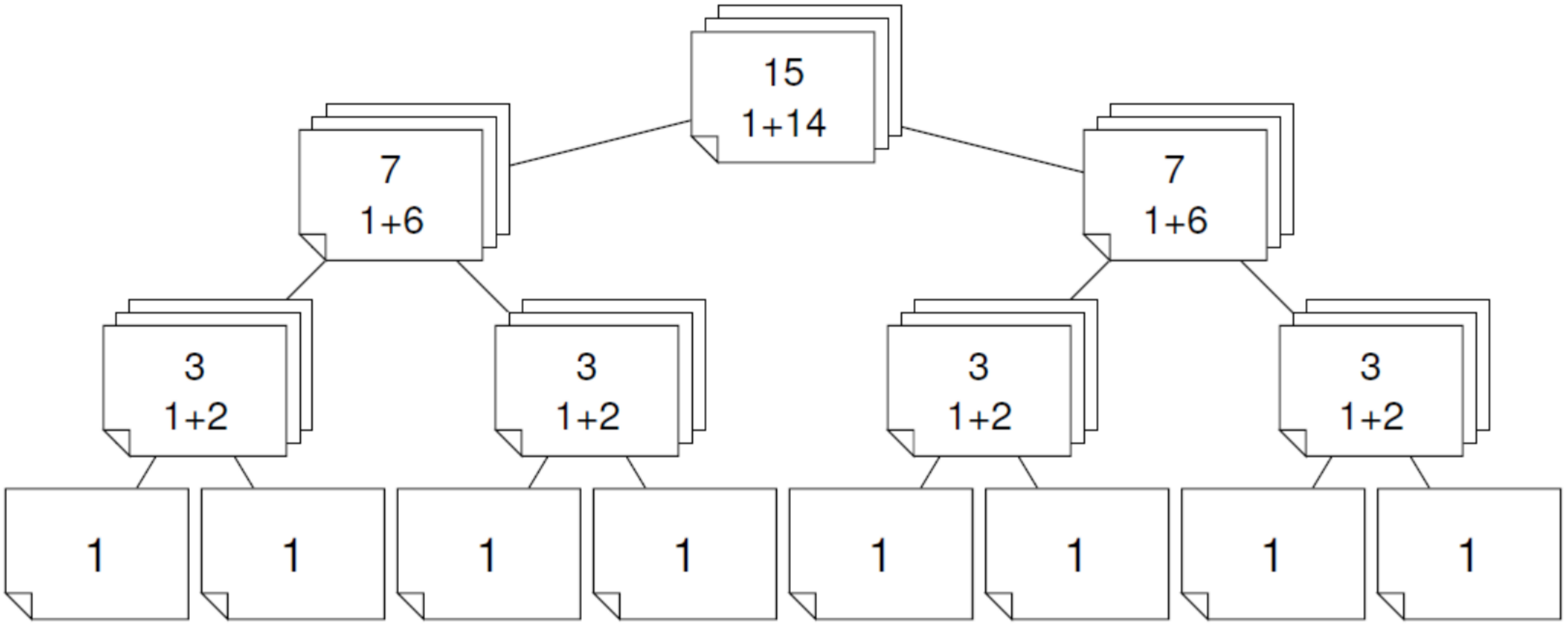
- Jeder nimmt ein Blatt und gibt den Rest des Stapels an seinen Nachbarn weiter
- Aufwand: O(n)

Variante 2

- Jeder nimmt ein Blatt und gibt jeweils die Hälfte des Restes an zwei Nachbarn weiter
- Aufwand: $O(\log_2 n)$

Teile und Beherrsche

Variante 2



Teile und Beherrsche

Beispiel: Maximum-Subarray-Problem

- Annahme: Wir suchen das maximale Teilfeld des Arrays A[links ... rechts]
- Teile und beherrsche: Zerlege das Teilfeld in 2 Teilfelder (falls möglich)

A[links ... mitte] und A[mitte+1 ... rechts]

Teile und Beherrsche

Beispiel: Maximum-Subarray-Problem

Jedes zusammenhängende Teilfeld A[i...j] kann folgendermaßen liegen

Komplett im Teilfeld A[links...mitte]	links ≤ i ≤ j ≤ mitte
Komplett im Teilfeld A[mitte+1...rechts]	mitte < i ≤ j ≤ rechts
Mittig	links ≤ i ≤ mitte < j ≤ rechts

Damit muss auch das maximale Teilfeld in genau einer dieser Lagen liegen

Teile und Beherrsche

Beispiel: Maximum-Subarray-Problem

Damit kann man tatsächlich das Gesamtproblem (rekursiv) in kleinere Teilprobleme zerlegen, da die beiden ersten Fälle kleinere Instanzen des Gesamtproblems sind.

Dann fehlt lediglich die Suche nach einem Teilfeld, das die Mitte überspannt.

Die größere dieser drei Lösungen ist die gesuchte Lösung.

Problem: Die dritte Lage ist keine kleinere Instanz des Gesamtproblems.

Allerdings kann man beobachten, dass für die Lösung des die Mitte überkreuzenden maximalen Teilfeldes die beiden Probleme A[i...mitte] und A[mitte+1...j] zu lösen sind

Teile und Beherrsche

```
function MAXCROSSINGSUBARRAY (A, left, middle, right)
    left_sum = -∞, sum = 0
    for i = middle downto left
        sum = sum + A[i]
        if sum > left_sum then
            left_sum = sum, max_left = i
        end if
    end for
    right_sum = -∞, sum = 0
    for i = middle+1 to right do
        sum = sum + A[i]
        if sum > right_sum then
            right_sum = sum, max_right = i
        end if
    end for
    return max_left, max_right, left_sum+right_sum
end function
```

Teile und Beherrsche

```
function MAXIMUMSUBARRAY_V3(A, left, right)
    if left == right then return (left, right, A[left])
    end if
    middle = ⌊(left+right)/2⌋
    (ll, lr, ls) = MAXIMUMSUBARRAY_V3(A, left, middle)
    (rl, rr, rs) = MAXIMUMSUBARRAY_V3 (A, middle+1, right)
    (ml, mr, ms) = MAXCROSSINGSUBARRAY(A, left, middle, right)
    if (ls ≥ rs) ∧ (ls ≥ ms) then return (ll, lr, ls)
    else if (rs ≥ ls) ∧ (rs ≥ ms) then return (rl, rr, rs)
    else return (ml, mr, ms)
    end if
end function
```

Teile und Beherrsche

Beispiel: Maximum-Subarray-Problem

Wie lange ist die Laufzeit von MAXCROSSINGSUBARRAY()?

Wie ist die Laufzeit von MAXIMUMSUBARRAY_V3() bzw. was ist das Problem bei der Abschätzung der Laufzeit?

Teile und Beherrsche

Zusammenfassung

Vorteil:

Einfachere Lösbarkeit von Problemen durch Verringerung der Komplexität

Nachteil:

Rekursion kann je nach Problem bei großen n die Laufzeit beeinflussen

Dynamische Programmierung

- Rekursive Berechnung und Kombination von Teilproblemen (wie bei Teile-und-Beherrsche)
- Systematische Speicherung von Zwischenergebnissen
- Mehrfachberechnung von Teillösungen vermeiden (Speicherkomplexität vs. Zeitkomplexität)

Später: Verwendung der dynamischen Programmierung für Optimierungsprobleme

Dynamische Programmierung

Fibonacci-Folge

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377

Dynamische Programmierung

Definition

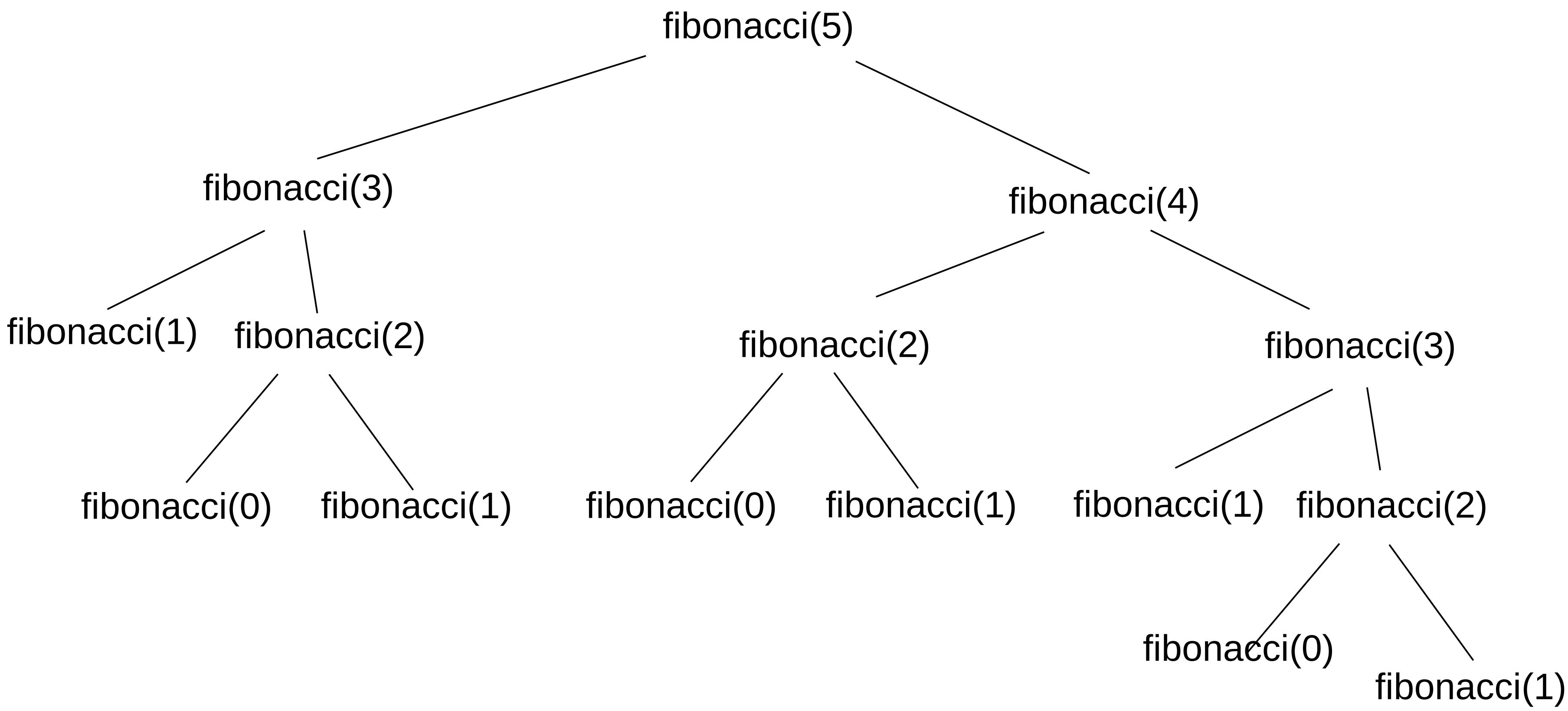
$$F_n = \begin{cases} 0 & für\ n = 0 \\ 1 & für\ n = 1 \\ F_{n-1} + F_{n-2} & für\ n > 1 \end{cases}$$

Dynamische Programmierung: Fibonacci-Folge

```
function FIBONACCI(n)
    if n < 0 then
        Fehlerausgabe
    else if n == 0 then
        return 0
    else if n == 1 then
        return 1
    else
        return FIBONACCI(n-1) + FIBONACCI(n-2)
    end if
end function
```

Dynamische Programmierung

Darstellung der Aufrufe als Baum (n = 5):



Dynamische Programmierung

Vermeiden der mehrfachen Lösung von Teilproblemen durch Tabellierung

Beispiel: Einschalten der Tabellierung in MAPLE durch Schlüsselwort remember

```
F:=proc (n::integer) option
        remember;
        if  n < 0 then NULL
        elif n = 0 then 0
        elif n = 1 then 1
        else F(n-1) + F(n-2) f1
end;
```

Beispiel: Einschalten der Tabellierung in python (ab 3.10) durch Dekorator cache

```
from functools import cache
@cache
def fibonacci (n) :
   if  n <= 1:
        return n
   return fibonacci(n – 1) + fibonacci(n + 1)
```

Dynamische Programmierung

Geht es auch anders (besser) im Fall der Fibonacci-Zahlen?
(nicht dynamisch!)

```
function FIBONACCI(n)
   if n < 0 then
      Fehlerausgabe und Abbruch
   else if n <= 1 then
      f = n
   else
      z1 = 0, z2 = 1, f = 1
      for i = 2 to n do
         f = z1 + z2
         z1 = z2, z2 = f
      end for
   end if
      return f
end function
```

Die Glieder der Fibonacci-Folge f_n lassen sich für alle n über die Formel von Binet berechnen

$$f_n = \frac{1}{\sqrt{5}}(\Phi^n - \overline{\Phi}^n)$$

mit

Φ = Goldener Schnitt

$$\overline{\Phi} = 1 - \Phi = -\frac{1}{\Phi} = \frac{1-\sqrt{5}}{2}$$

Dynamische Programmierung

Zusammenfassung

Vorteil:

- Vermeidung von Rekursionen durch Wiederverwenden bekannter Teilergebnisse

Nachteil:

- Speicherbedarf kann je nach Problem stark mit n wachsen