Algorithmen & Datenstrukturen

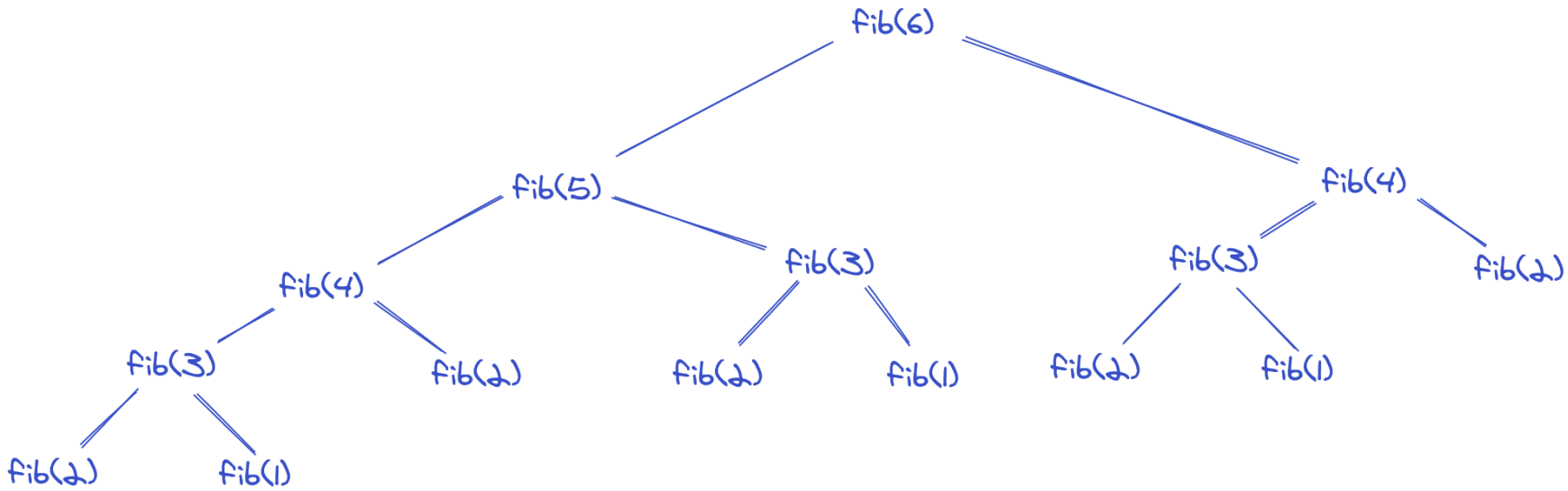
DYNAMISCHES PROGRAMMIEREN

Dynamisches Programmieren

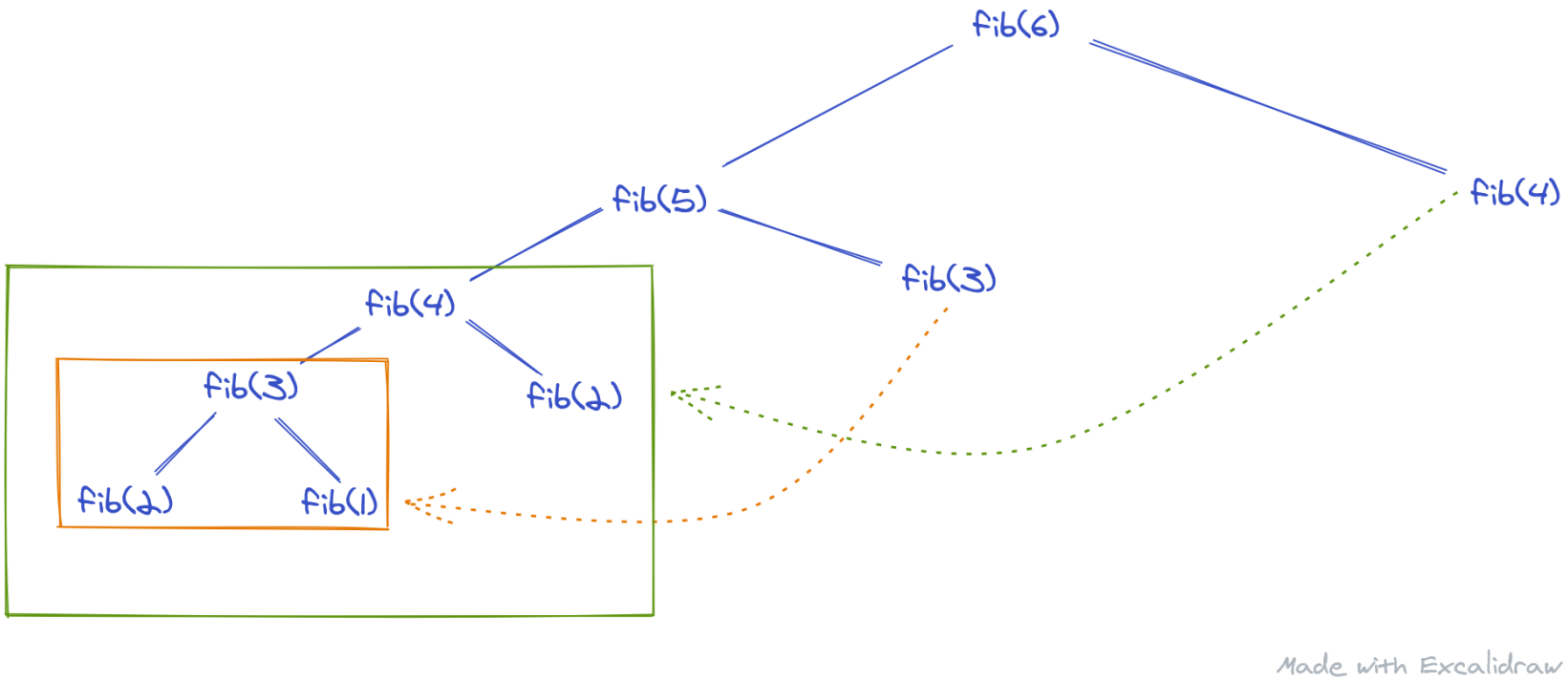
- Reduziert die Laufzeit z. B. bei der rekursiven Fibonacci-Berechnung
  - von $O(n^2)$ auf $O(n)$
  - dafür wird mehr Speicher verwendet
- Idee:
  - Probleme sind nicht unabhängig voneinander
  - Daher die Zwischenergebnisse für andere speichern

- Kleine Anekdote:
  - Der Name kommt aus den 70ern
  - Programmieren: das abarbeiten einer Rechenvorschrift
  - dynamisch: sich ändernde Tabellen

Dynamisches Programmieren



Dynamisches Programmieren



Dynamisches Programmieren

- Reduziert die Laufzeit z. B. bei der rekursiven Fibonacci-Berechnung
  - von $O(n^2)$ auf $O(n)$
  - dafür wird mehr Speicher verwendet
- Idee:
  - Probleme sind nicht unabhängig voneinander
  - Daher die Zwischenergebnisse für andere speichern

- Für jeden rekursiven Algorithmus geeignet
  - Voraussetzung: bei gleicher Eingabe, das gleiche Ergebnis (referenziell transparent)
  - Einfach: *top-down dynamic programming*
  - Letzter Schritt in einer Methode: Ergebnis zur Eingabe speichern
  - Erster Schritt in einer Methode: Prüfen, ob Ergebnis vorhanden
  - Dazwischen: wie vorher

Dynamisches Programmieren

- Fibonacci-Zahlen

Rekursiv

```
int fib(int n) {

  if (n < 2)
      return n;

  return fib(n - 1) +
         fib(n - 2);
}
```

Rekursiv + dyn. Programmierung

```
final int maxN = 47;
int[] knownF = new int[maxN];

int fib(int i) {
  if (knownF[i] != 0)
    return knownF[i];

  int t = i;

  if (i > 1)
    t = fib(i - 1) + fib(i - 2);

  return knownF[i] = t;
}
```

Dynamisches Programmieren

- Ergebnis
  - Rekursiv für n = 45

```
Result: 1.134.903.170 Calls: 3.672.623.805 Duration: 4.786ms
Process finished with exit code 0
```

≈ $1{,}618^N = 1{,}618^{45} = 2.535.322.890$

  - Rekursiv + dyn. Programmierung für n = 45

```
Result: 1.134.903.170 Calls: 89 Duration: 0ms
Process finished with exit code 0
```

≈ 2N – 1 (Zufall?)

Dynamic programming reduces the running time of a recursive method to be at most the time required to evaluate the function for all arguments less than or equal to the given argument, treating the cost of recursive invocation as constant. [Sedgewick2003]

Übung

- Rucksackproblem (knapsack problem)
  - Items mit Größe und Wert
  - Rucksack der Größe x
  - Was nimmt man mit, um möglichst viel Wert zu transportieren?

```
class Item { int size; int val; }

static Item[] items = …

static int knap(int cap) {
  int i, space, max, t;
  for (i = 0, max = 0; i < N; i++)
    if ((space = cap - items[i].size) >= 0)
      if ((t = knap(space) + items[i].val) > max)
        max = t;

  return max;
}
```