RI203
Uvod u računarske algoritme

dr.sc. Edin Pjanić

Pregled predavanja

- Optimizacija rekurzivnih problema
  - Memoizacija
  - Dinamičko programiranje

Optimizacija rekurzivnih problema

- Kod nekih rekurzivnih problema vrijeme rješavanja se može značajno smanjiti korištenjem tehnika:
    - memoizacije
    - dinamičkog programiranja

- Da bi se to moglo postići ti problemi moraju biti takvi da:
    - Rješavanje nekog koraka zahtijeva rješenje iz nekog drugog (prethodnog) koraka.
    - Rješenje iz prethodnog koraka možemo iskazati.

Memoizacija

- Memoizacija podrazumijeva pamćenje (čuvanje) već nađenih rješenja **tokom rekurzije** i njihovu upotrebu u rješavanju trenutnog koraka (pristup rješavanju problema je *top-down*).
- Čuvanje prethodnih vrijednosti se vrši u odgovarajućim strukturama podataka, u skladu sa problemom.

- Primjer: Kod računanja Fibonaccijevih brojeva možemo zapisivati izračunate brojeve iz prethodnog koraka i upotrijebiti ih tako da smanjimo njihovo ponovno računanje pri računanju u trenutnom koraku.

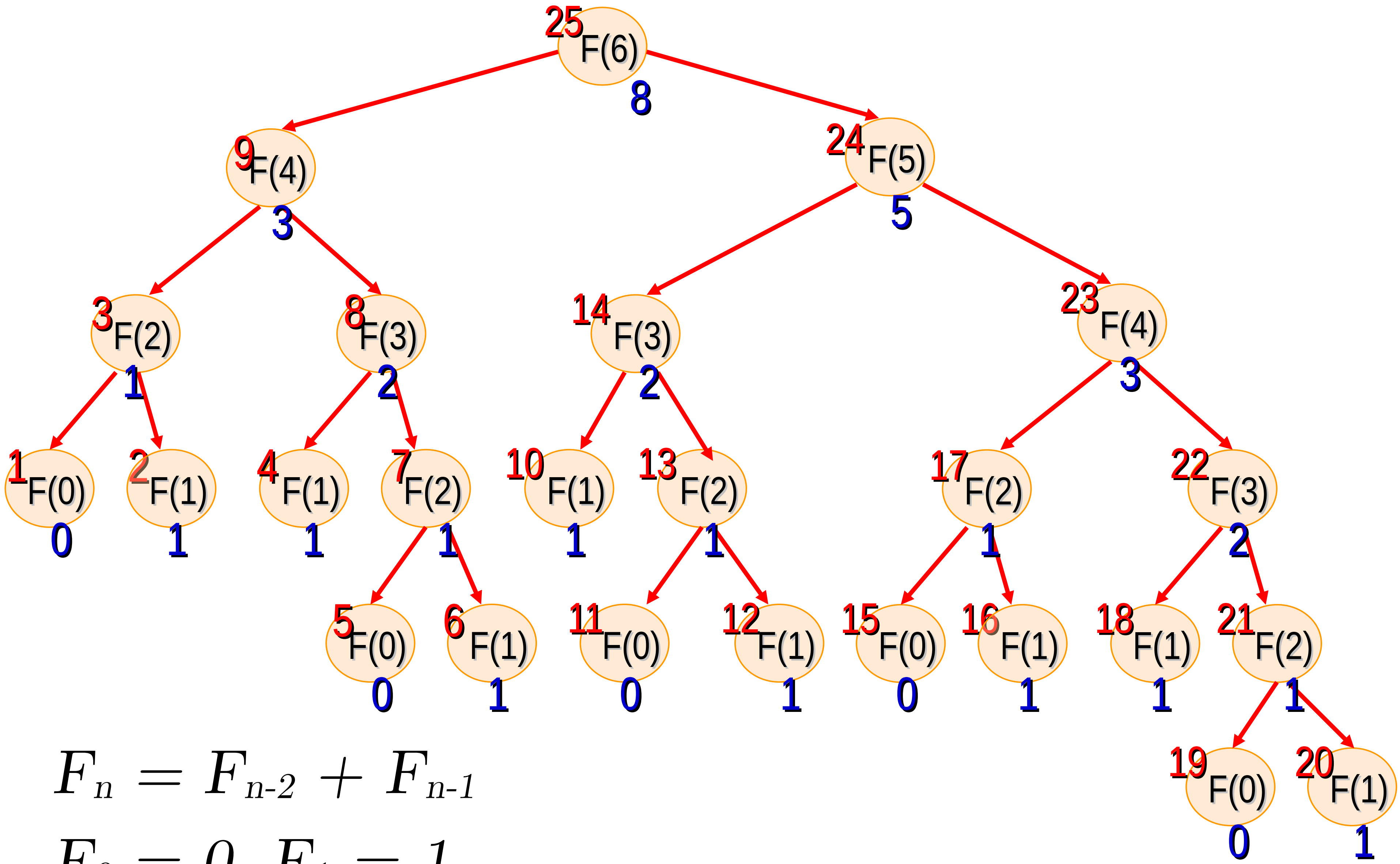
Fibonaccijevi brojevi - rekurzivno

Definicija:

- $F_n = F_{n-2} + F_{n-1}$, za $n>1$
- $F_0 = 0$, $F_1 = 1$

```
int F(int n)
{
  if (n < 2)
    return n;
  else
    return F(n-2) + F(n-1);
}
```

Fibonaccijevi brojevi – stablo izvršavanja



$$F_n = F_{n-2} + F_{n-1}$$

$$F_0 = 0,\ F_1 = 1$$

Fibonacci - memoizacija

```
memoFib(n):
if (n < 2)
   return n
else
   if F[n] nije definisan
      F[n] ← memoFib(n − 1) + memoFib(n − 2)
return F[n]
```

- Niz F mora biti vidljiv pri svakom pozivu ove funkcije (globalan ili static).
- Pri implementaciji ovakvog algoritma za vrijednost "nije definisan" bismo mogli postaviti neku vrijednost koja se ne može pojaviti kao validna, npr. -1.
- Složenost ovakvog rješenja je O(n) – puno efikasnije od običnog rekurzivnog. Ovo zahtijeva O(n) dodatne memorije.

Dinamičko programiranje

- Dinamičko programiranje je pojam koji je uveo Richard Bellman 50. godina prošlog vijeka.
- Ovaj princip rješavanja problema podrazumijeva rješavanje složenih problema na način da se razbiju na jednostavnije probleme ali malo drugačije nego d-a-q.
- Koraci:
    - Definisati problem rekurzivno: Napisati rekurzivnu formulu ili algoritam cijelog problema kao rješenje manjih podproblema.
    - Napisati rješenje rekurzije na *bottom-up* principu (odozdo nagore), tj. napisati algoritam koji **kreće od osnovnog slučaja** i ide ka cjelokupnom problemu.
- U svojoj osnovi, DP rekurziju pretvara u **iterativno rješenje**. Obično su DP algoritmi takvi da se međurezultati smještaju u neku vrstu tabele (ili nekoliko objekata) tako da se mogu ponovo koristiti.

Dinamičko programiranje – bottom-up pristup

- **Identifikovati podprobleme**; na koji način se sve može pozvati rekurzivni algoritam; početni uslovi
- **Identifikovati zavisnost među podproblemima**; nacrtati dijagram ovisnosti podproblema o drugim podproblemima
- **Analizirati potreban prostor i vrijeme**; broj podproblema određuje potreban prostor i kompleksnost
- **Odabrati strukturu podataka za pamćenje međurezultata**; uglavnom je dovoljno nekoliko objekata
- **Pronaći ispravan redoslijed rješavanja podproblema**; podprobleme rješavati tako da se riješi prvo podproblem od kojeg ovisi drugi podproblem
- **Zapisati algoritam**; kad znamo sve ovo gore možemo zapisati algoritam, obično **iterativno** (pomoću petlji)

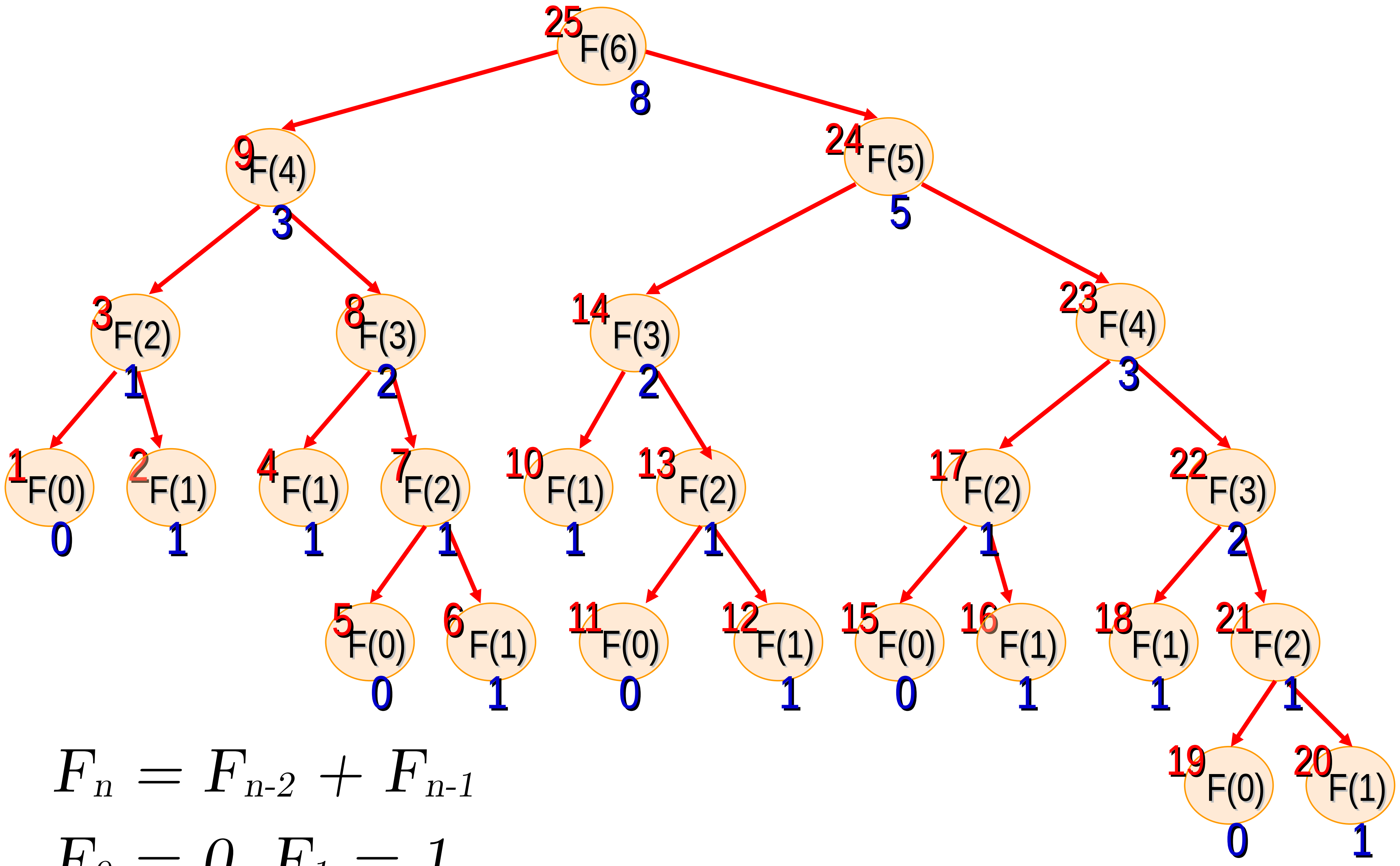
Koje probleme rješavati dinamičkim programiranjem

- Za rješavanje dinamičkim programiranjem pogodni su problemi koji pokazuju naredne osobine:

- **Optimalna podstruktura**: optimalno rješenje problema se nalazi unutar optimalnih rješenja podproblema. Drugim riječima, optimalno rješenje se dobija iz optimalnih rješenja podproblema. Pri tome, rješenje svakog podproblema se treba moći riješiti nezavisno.

- **Preklapanje podproblema**: podproblemi se pojavljuju kao identični podproblemi različitih problema te njihovo rješenje možemo zapamtiti i iskoristiti za rješavanje svih problema ili podproblema koji ih sadrže.

Fibonaccijevi brojevi – stablo izvršavanja

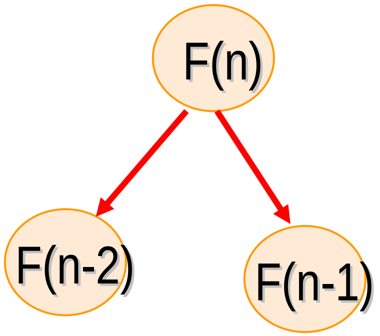


$$F_n = F_{n\text{-}2} + F_{n\text{-}1}$$

$$F_0 = 0,\ F_1 = 1$$

Fibonacci DP

Na prethodnom slajdu se vidi da rješenje u nekom koraku zavisi samo od prethodna dva koraka. To znači da je dovoljno zapamtiti samo prethodne dvije vrijednosti.
U primjeru lijevo, izračunate vrijednosti smještamo u niz (DP sa memoizacijom).
U primjeru desno pamtimo samo prethodne dvije vrijednosti u dvije varijable (optimalno rješenje).
Složenost je O(n) uz manje korištenje memorije i sistemskog steka jer nema rekurzije.




```
DPFibonacci1(n):
F[0] ← 0
F[1] ← 1
for i ← 2 to n
   F[i] ← F[i-1] + F[i-2]
return F[n]
```

```
DPFibonacci2(n):
if n <= 1 return n
F_2 ← 0
F_1 ← 1
for i ← 2 to n
   Fn  ← F_1 + F_2
   F_2 ← F_1
   F_1 ← Fn
return Fn
```

Primjer: najveći podniz (maximum subarray)

- Za dati niz, potrebno je pronaći dio niza za koji je suma elemenata najveća. Taj dio niza nazivamo *najveći podniz*.
- Npr. za niz A[0 … 12]:
    - A: {5, -4, 7, 2, -12, 6, -2, 5, 6, -1, -13, 7, 2}

Najveći podniz je A[5 … 8] a suma iznosi 15.

Rješenje:

- Brute-force: $\Theta(n^2)$
- Divide and conquer:
    - Master teorema => $\Theta(n \log n)$

Master teorema – uprošteno

1. slučaj: ako je $a < b^c$, odnosno $\log_b a < c$, tada je

$$T(n) = \Theta(n^c)$$

2. slučaj: ako je $a = b^c$, odnosno $\log_b a = c$, tada je

$$T(n) = \Theta(n^c \log n)$$

3. slučaj: ako je $a > b^c$, odnosno $\log_b a > c$, tada je

$$T(n) = \Theta(n^{\log_b a})$$

DP: najveći podniz (maximum subarray)

- Zadatak sa slajda 3: Za dati niz, potrebno je pronaći dio niza za koje je suma elemenata najveća. Taj dio niza nazivamo *najveći podniz*.
- Npr. za niz A[0 … 12]:
  - A: {5, -4, 7, 2, -12, 6, -2, 5, 6, -1, -13, 7, 2}

Rješenje:

- Pretpostavimo da znamo max sumu za dio niza koji završava na *i*-tom elementu, tj. S_i.
- Šta je u tom slučaju S_{i+1}?
  - $S_{i+1} = \max(S_i + a_{i+1}, a_{i+1})$
- Složenost: $\Theta(n)$