UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

CURS SDA 9: TEHNICI DE DEZVOLTARE A ALGORITMILOR(II)

PROGRAMARE DINAMICA

1. NOTIUNI DE BAZA IN PROGRAMARE DINAMICA
2. EXEMPLE:
    - A) SIRUL LUI FIBBONACCI
    - B) TAIEREA TIJEI
    - C) SUBSECVENTA COMUNA DE LUNGIME MAXIMA
    - D) PROBLEMA RUCSACULUI
    - E) SELECTIA ACTIVITATILOR
    - F) PROBLEMA NUMARARII RESTULUI
3. CONCLUZII

PROGRAMAREA DINAMICA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- "Those who forget the past are condemned to repeat it" (George Santayana)
- Definitie: Tehnica eficienta de implementare a algoritmilor recursivi, prin stocarea rezultatelor partiale
- Cand
  - algoritmul recursiv reface aceleasi calcule de mai multe ori
- Cum?
  - *stocare rezultate* in tabel
  - trebuie sa avem o *relatie de recurenta* corecta
- Potrivita in cazurile in care obiectele combinatoriale manifesta o ordonare stanga-dreapta a componentelor
  - siruri de caractere, arbori, poligoane, secvente de intregi

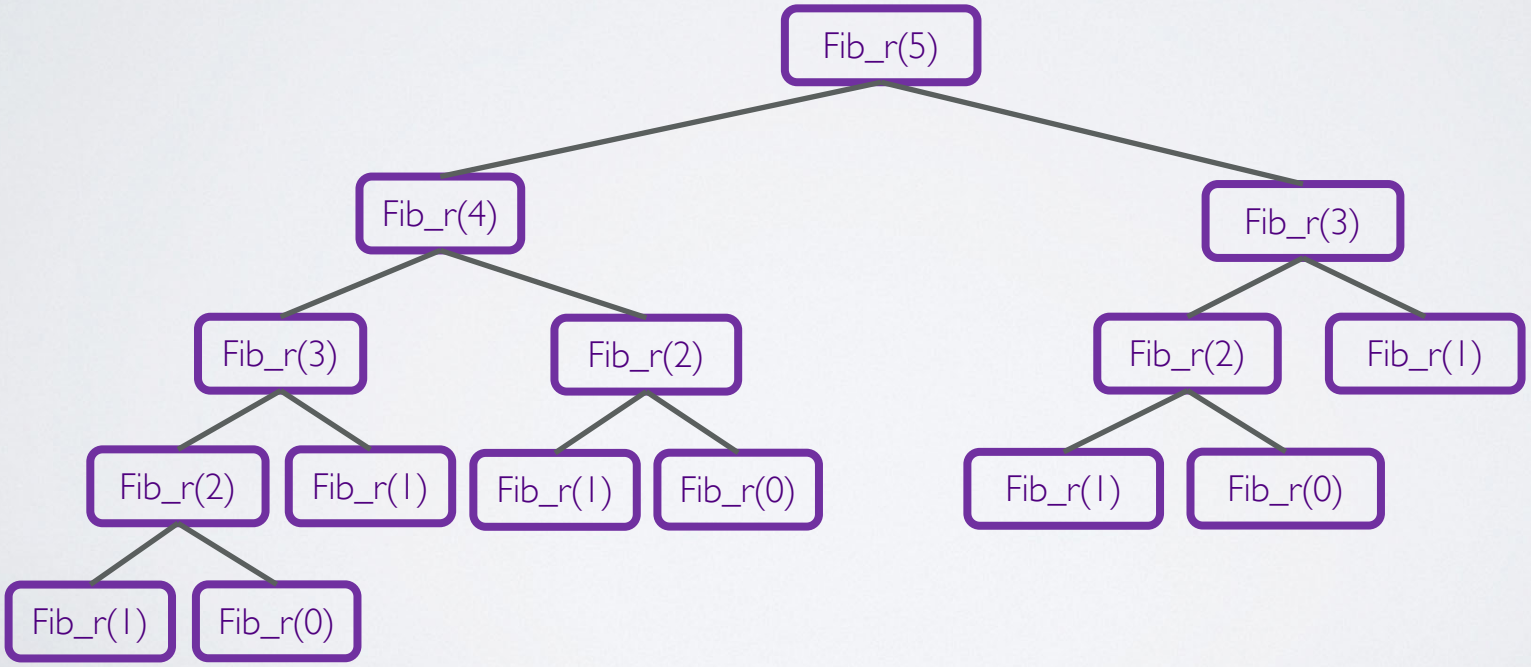
SIRUL LUI FIBONACCI – VARIANTA RECURSIVA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

$F_n = F_{n-1} + F_{n-2}, \; F_0 = 1, \; F_1 = 1$

$F_n = \{1,1,2,3,5,8,13,21,\ldots\}$

```
long Fib_r(int n)
{
  if (n == 0) return(1);
  if (n == 1) return(1);
  return(Fib_r(n-1) + Fib_r(n-2));
}
```



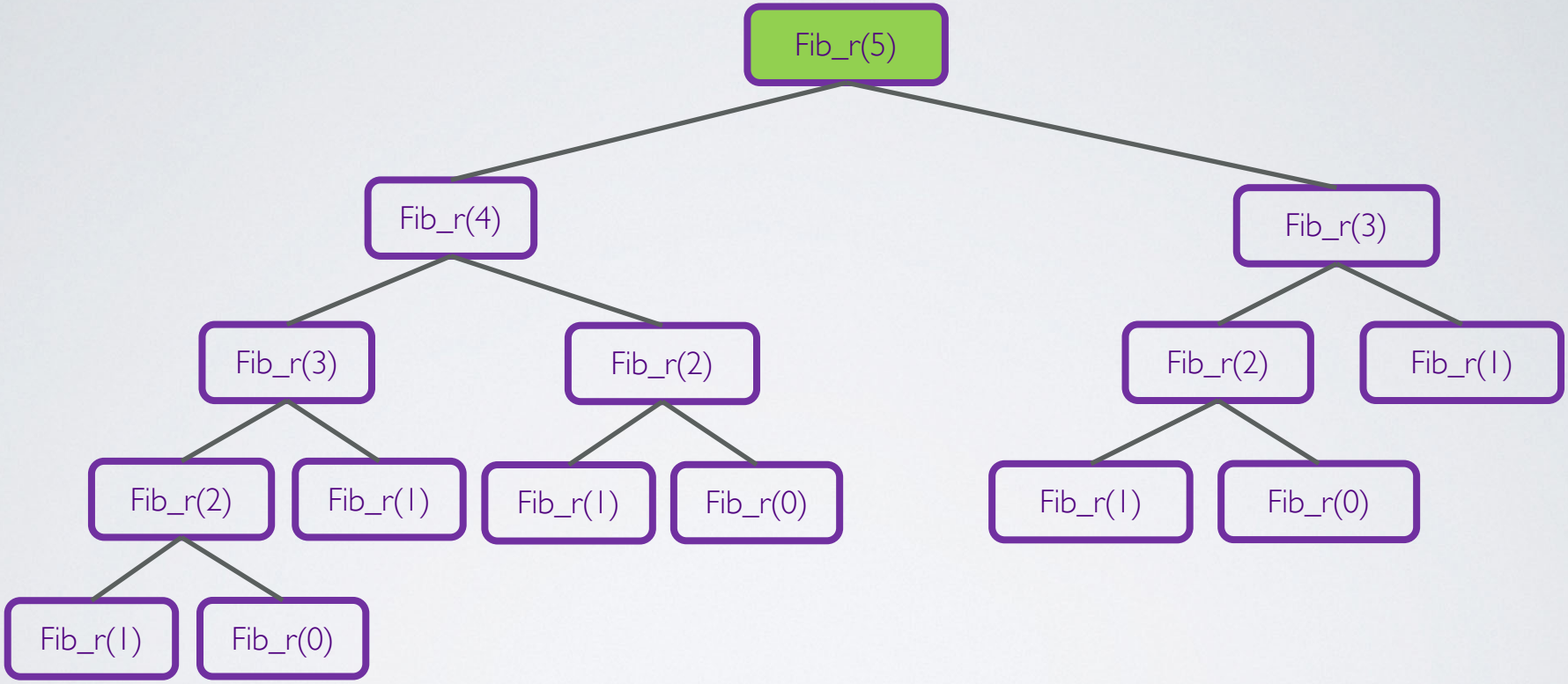
SIRUL LUI FIBONACCI – VARIANTA RECURSIVA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Numarul de apeluri ale functiei Fib_r este dat de urmatoarea recurenta:
  - T(0) = 1
  - T(1) = 1
  - T(n) = T(n-1)+T(n-2)+1
  - Se poate arata ca T(n) = $2F_{n+1}$-1, unde F_{n+1} este elementul n+1 din sirul lui Fibonacci
  - F(n)=$\left\lfloor \frac{\varphi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$=Θ($\varphi^n$), unde φ = 1.61803, timp de rulare exponential !

SIRUL LUI FIBONACCI – VARIANTA RECURSIVA

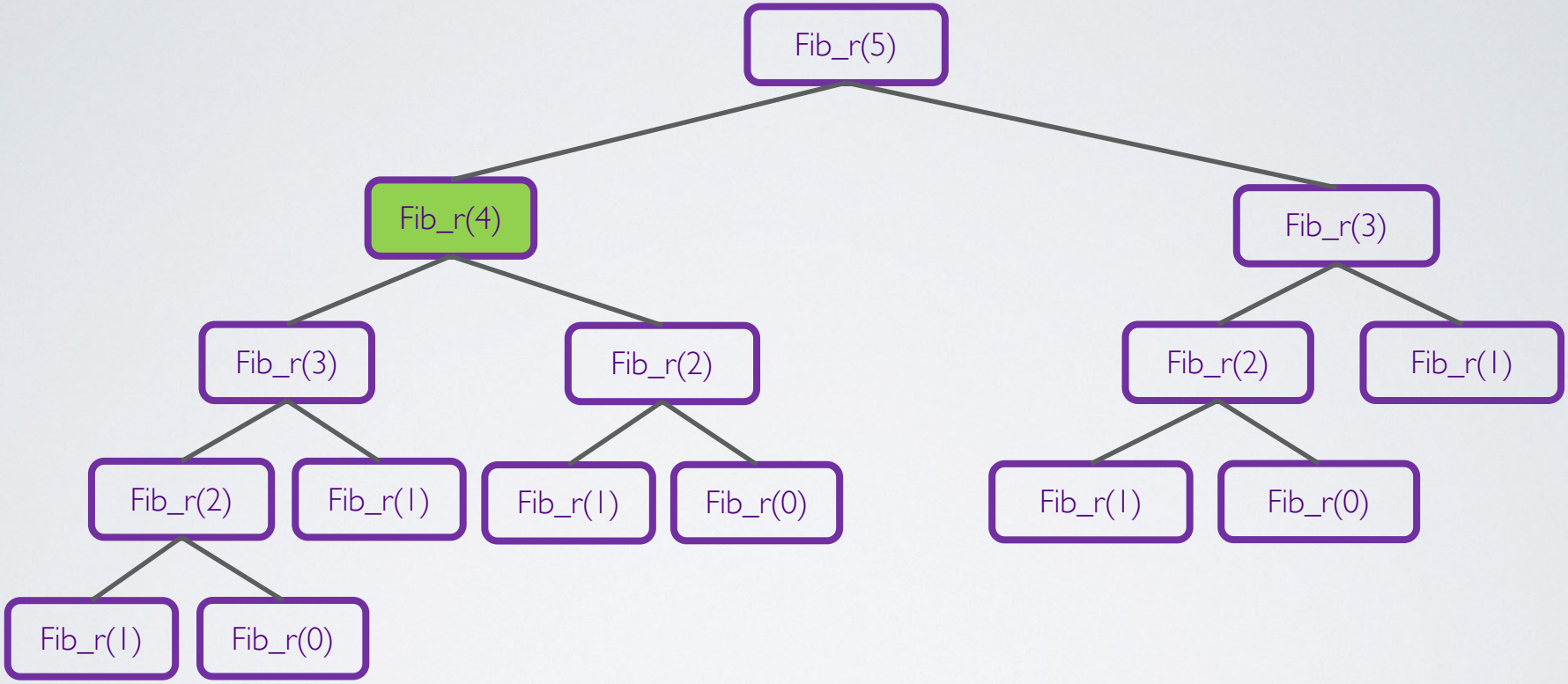
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



Arborele de recursivitate are Fib_r(0) si Fib_r(1) frunze !

SIRUL LUI FIBONACCI – VARIANTA RECURSIVA

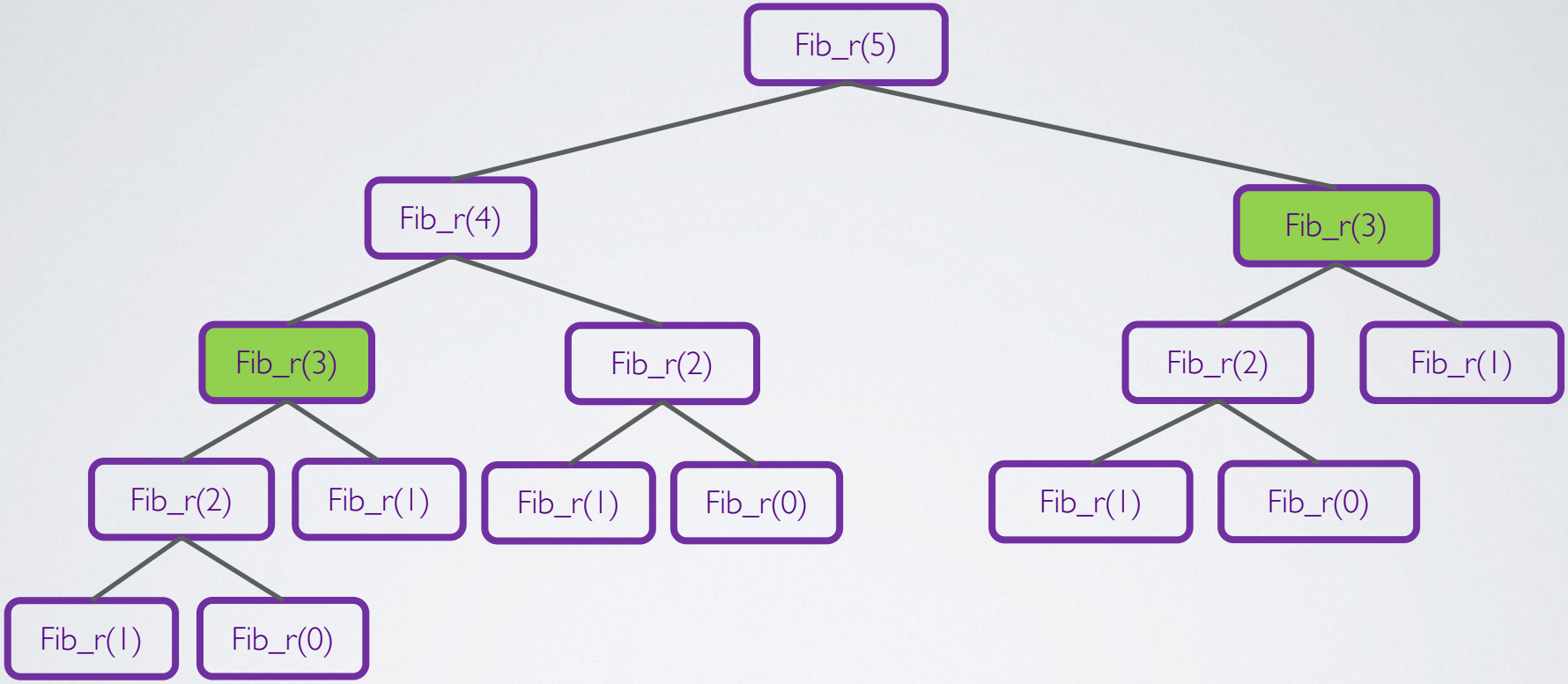
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



Arborele de recursivitate are Fib_r(0) si Fib_r(1) frunze !

SIRUL LUI FIBONACCI – VARIANTA RECURSIVA

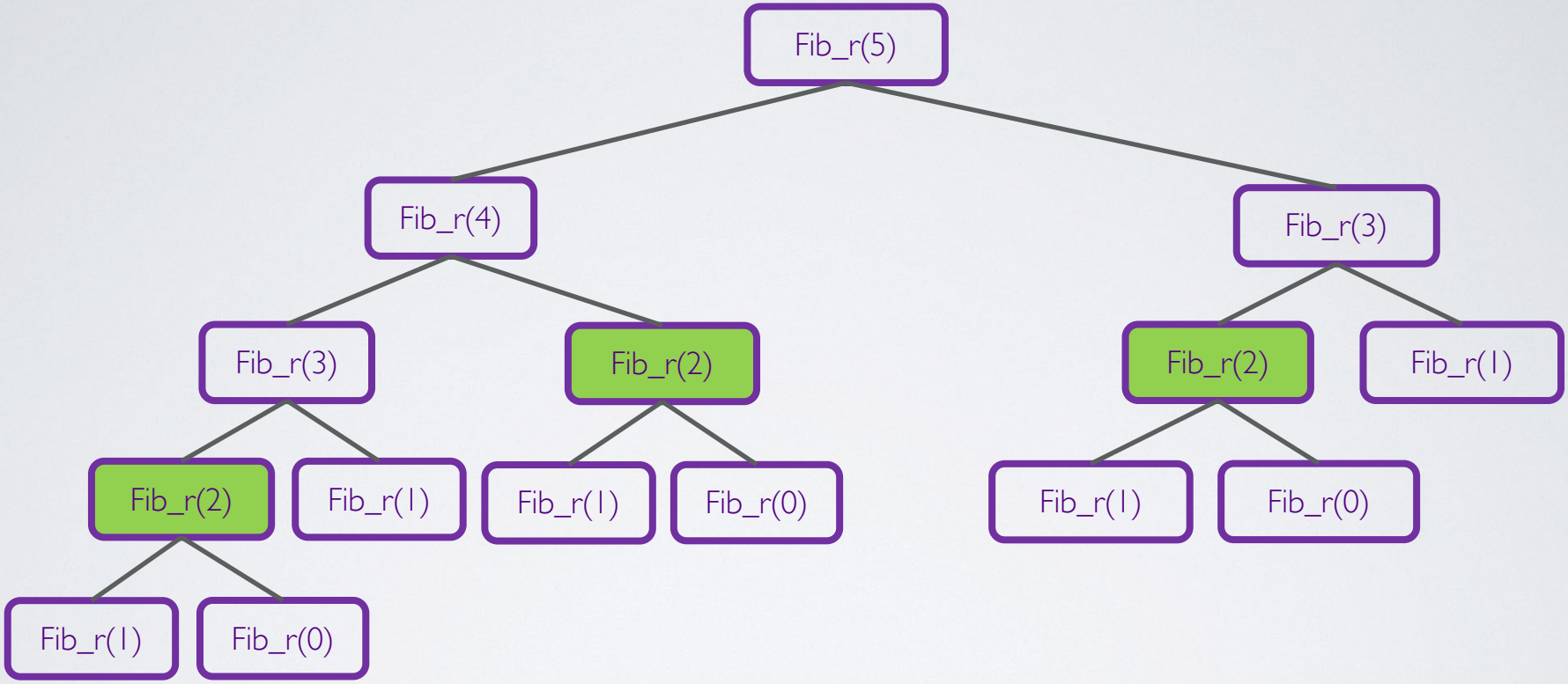
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



Arborele de recursivitate are Fib_r(0) si Fib_r(1) frunze !

SIRUL LUI FIBONACCI – VARIANTA RECURSIVA

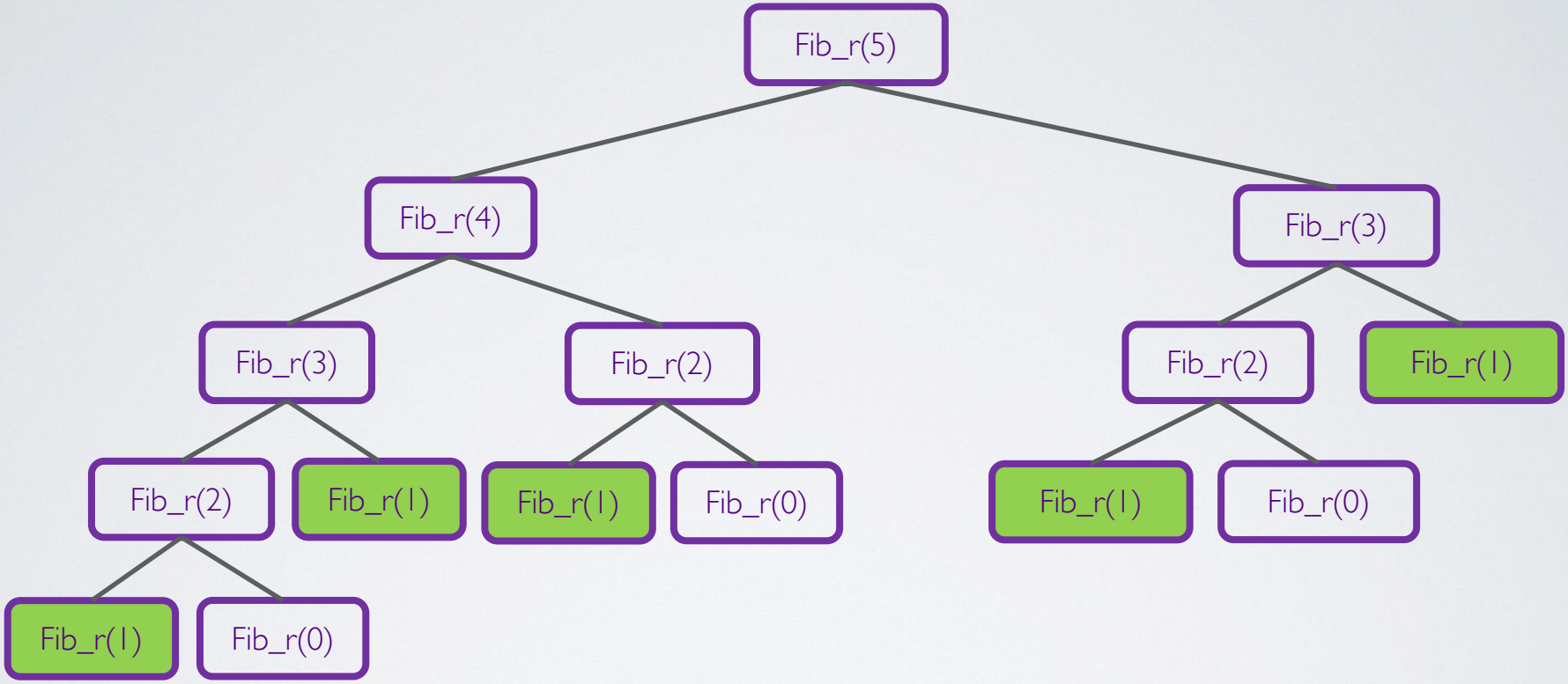
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



Arborele de recursivitate are Fib_r(0) si Fib_r(1) frunze !

SIRUL LUI FIBONACCI – VARIANTA RECURSIVA

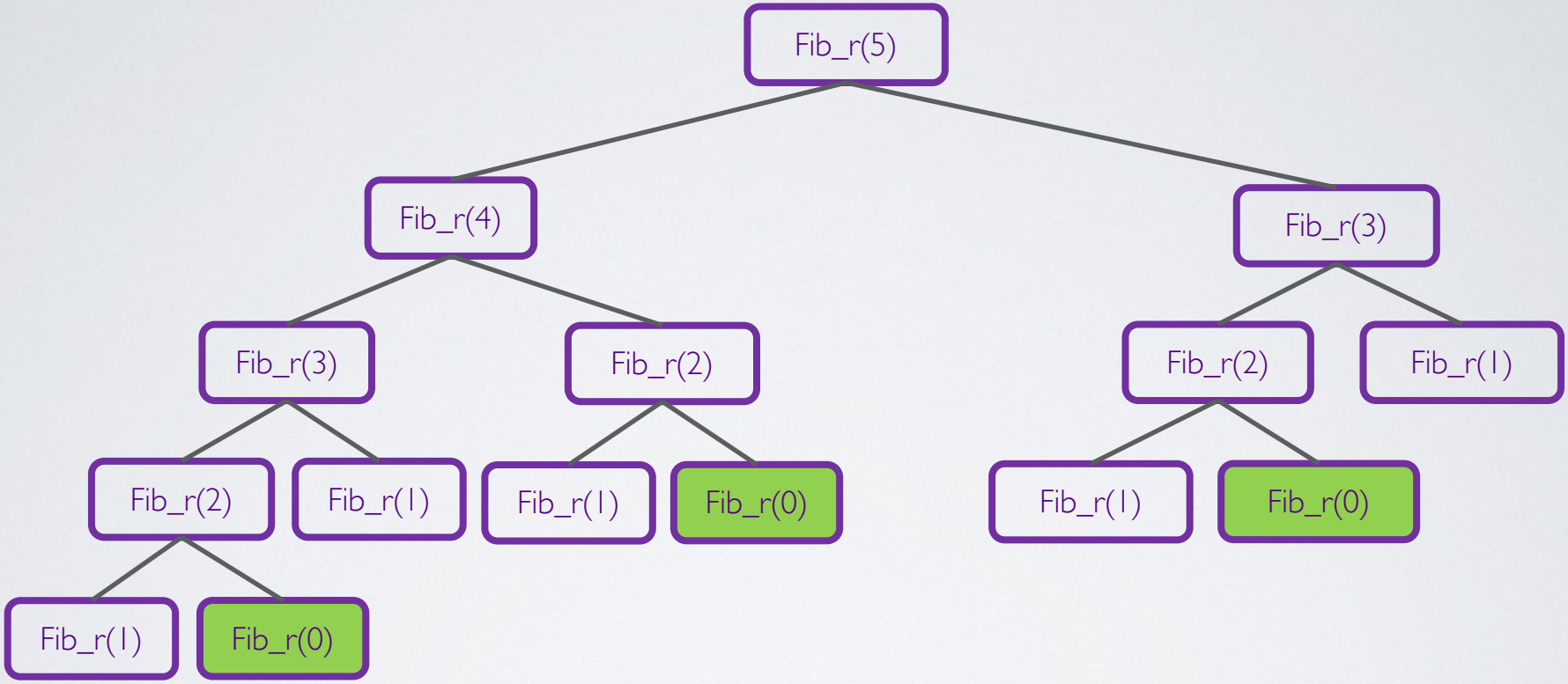
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



Arborele de recursivitate are Fib_r(0) si Fib_r(1) frunze !

SIRUL LUI FIBONACCI – VARIANTA RECURSIVA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



Arborele de recursivitate are Fib_r(0) si Fib_r(1) frunze !

SIRUL LUI FIBONACCI – VARIANTA RECURSIVA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Se refac aceleasi calcule de mai multe ori !

- Algoritm ineficient deoarece ramuri diferite ale recursivitatii fac acelasi lucru.
- Numarul total de **apeluri recursive unice este mic,** chiar daca numarul total de apeluri recursive este mare.
- Idee: evitam munca – calculele redundante
- Cum ?

SIRUL LUI FIBONACCI - PD CU "MEMOIZATION" (CACHING, MEMORARE)

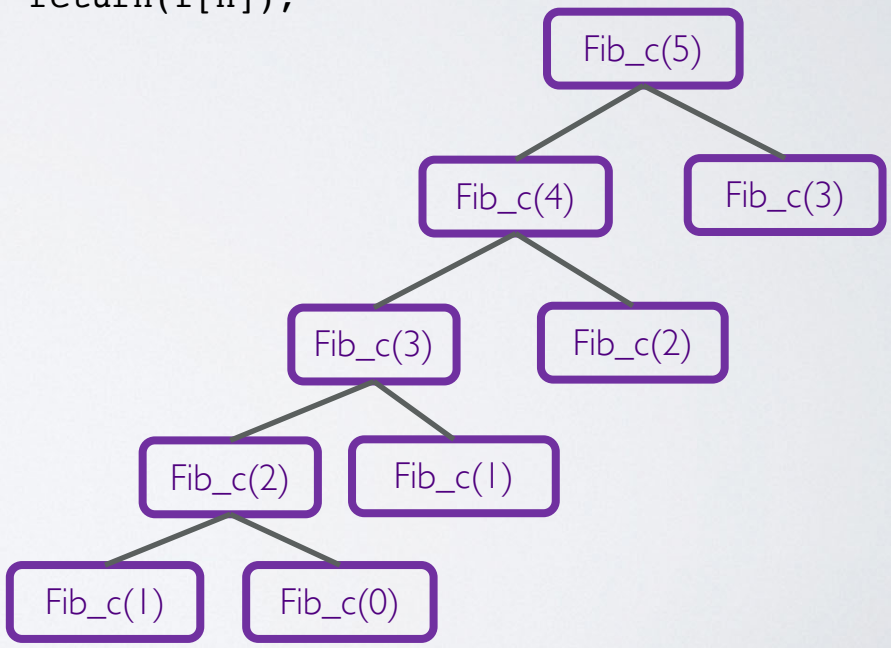
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

```
#define MAXN    45  /* largest interesting n */
#define UNKNOWN -1  /* contents denote an empty cell */
long f[MAXN+1];     /* array for caching computed fib values */

long fib_c_driver(int n)
{
   int i; /* counter */
   f[0] = 1;
   f[1] = 1;
   for (i=2; i<=n; i++)
      f[i] = UNKNOWN;
   return(Fib_c(n));
}
```

```
long Fib_c(int n)
{

   if (f[n] == UNKNOWN)
         f[n] = Fib_c(n-1) + Fib_c(n-2);
   return(f[n]);
}
```



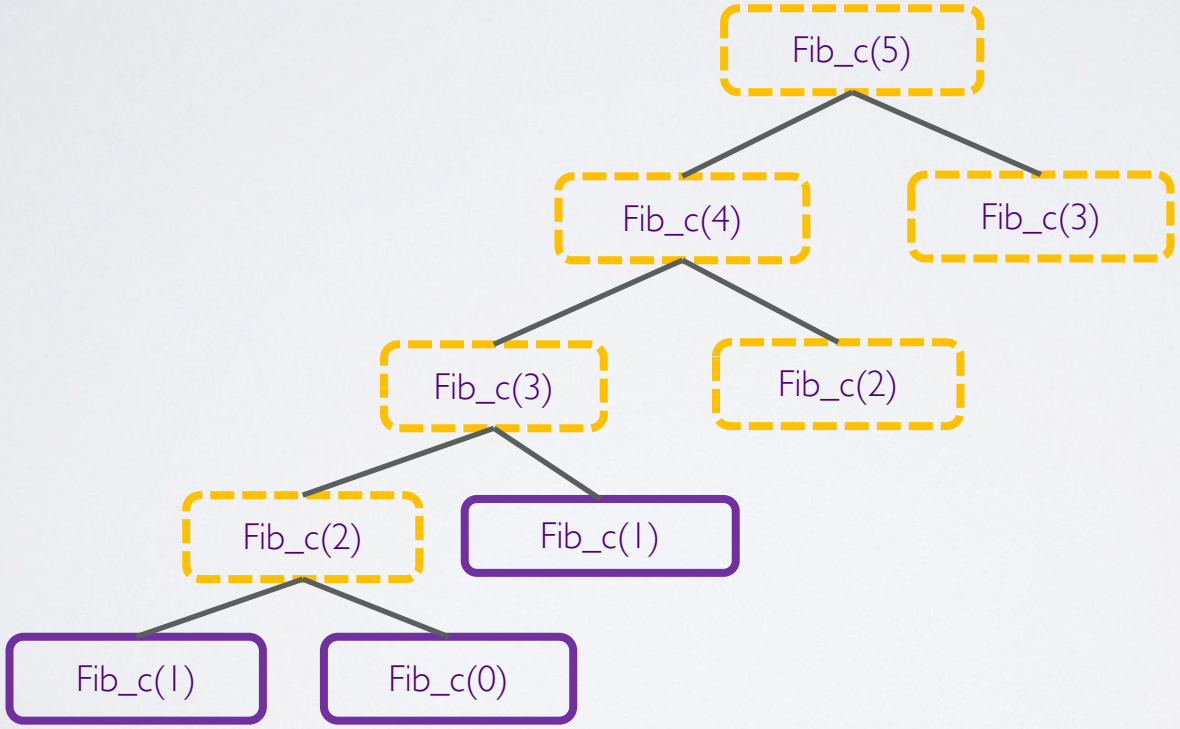
Nu avem ramificare de fapt in arborele de recursivitate

- Complexitate
  - Timp ?
  - Memorie ?

!!! Memorarea are sens cand spatiul parametrilor diferiti are dimensiune rezonabil!a (e.g. nu e utila la quicksort, DFS..)

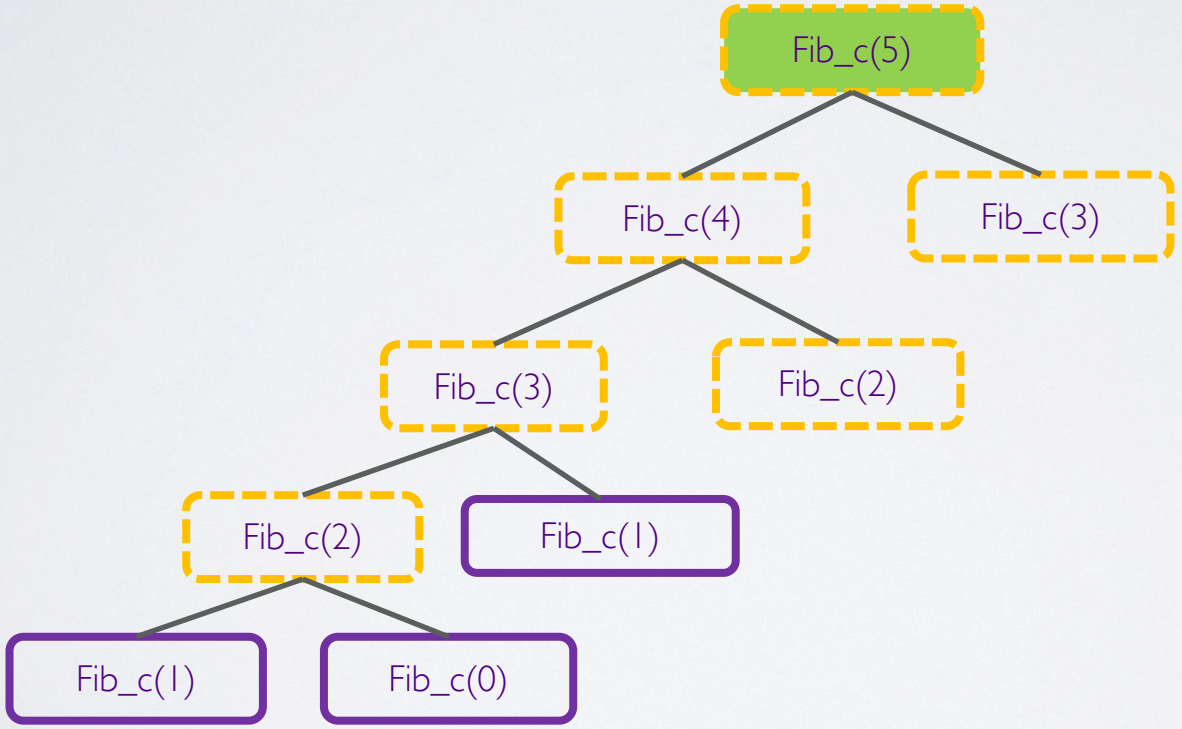
SIRUL LUI FIBONACCI - PD CU "MEMOIZATION" (CACHING, MEMORARE)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



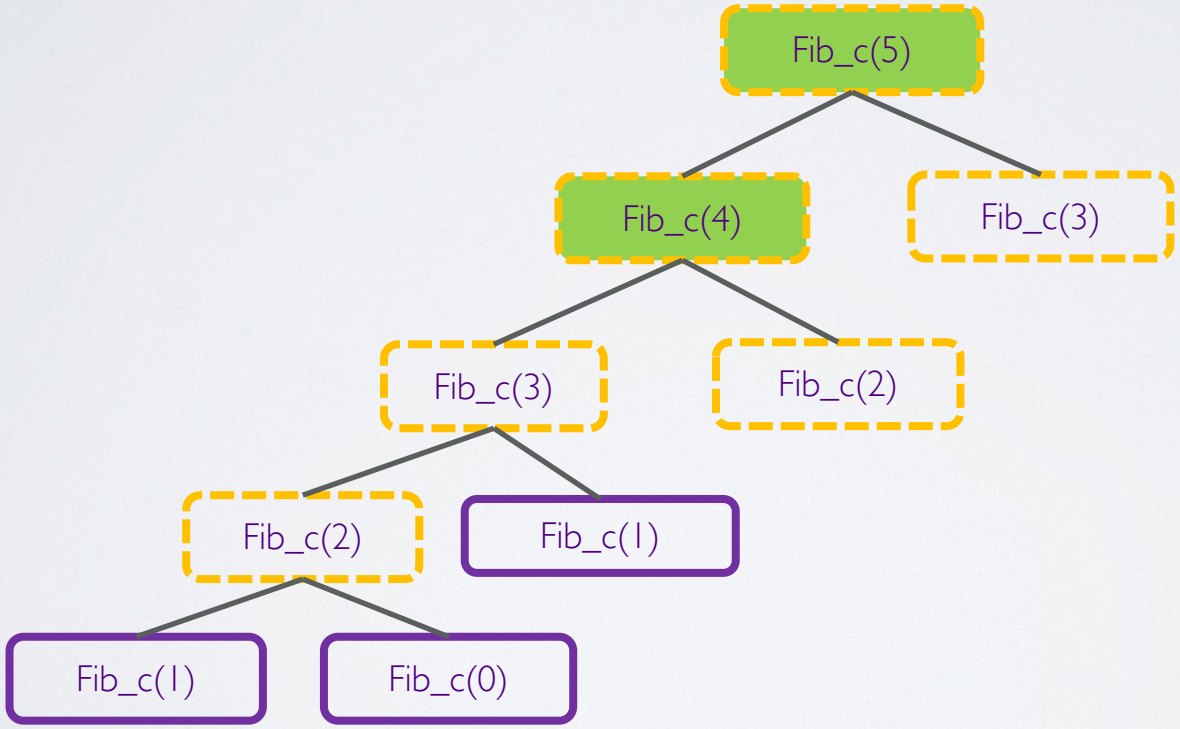
SIRUL LUI FIBONACCI - PD CU "MEMOIZATION" (CACHING, MEMORARE)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



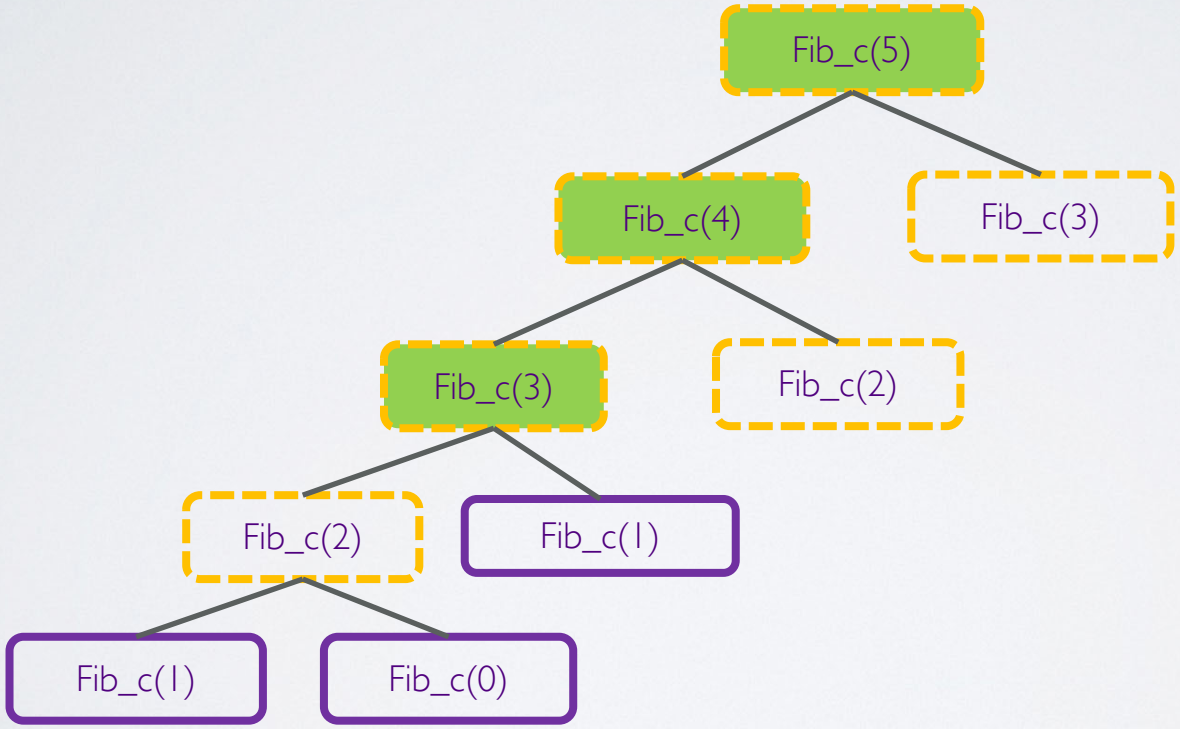
SIRUL LUI FIBONACCI - PD CU "MEMOIZATION" (CACHING, MEMORARE)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



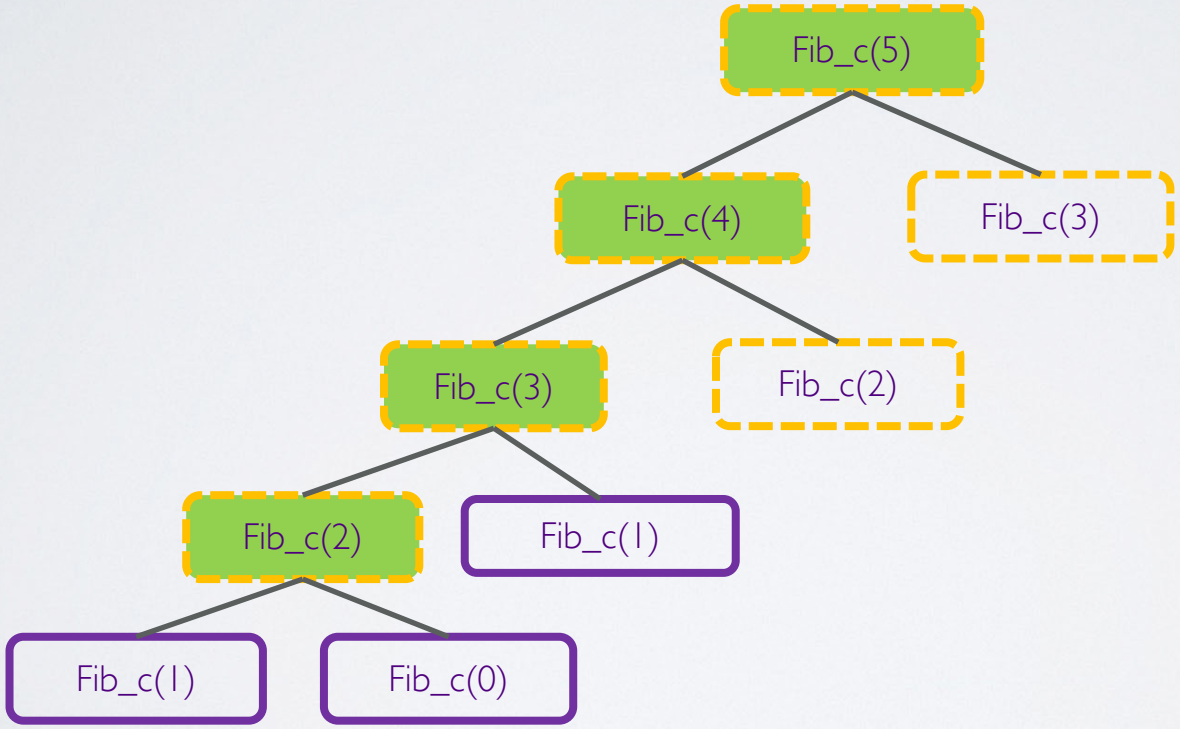
SIRUL LUI FIBONACCI - PD CU "MEMOIZATION" (CACHING, MEMORARE)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



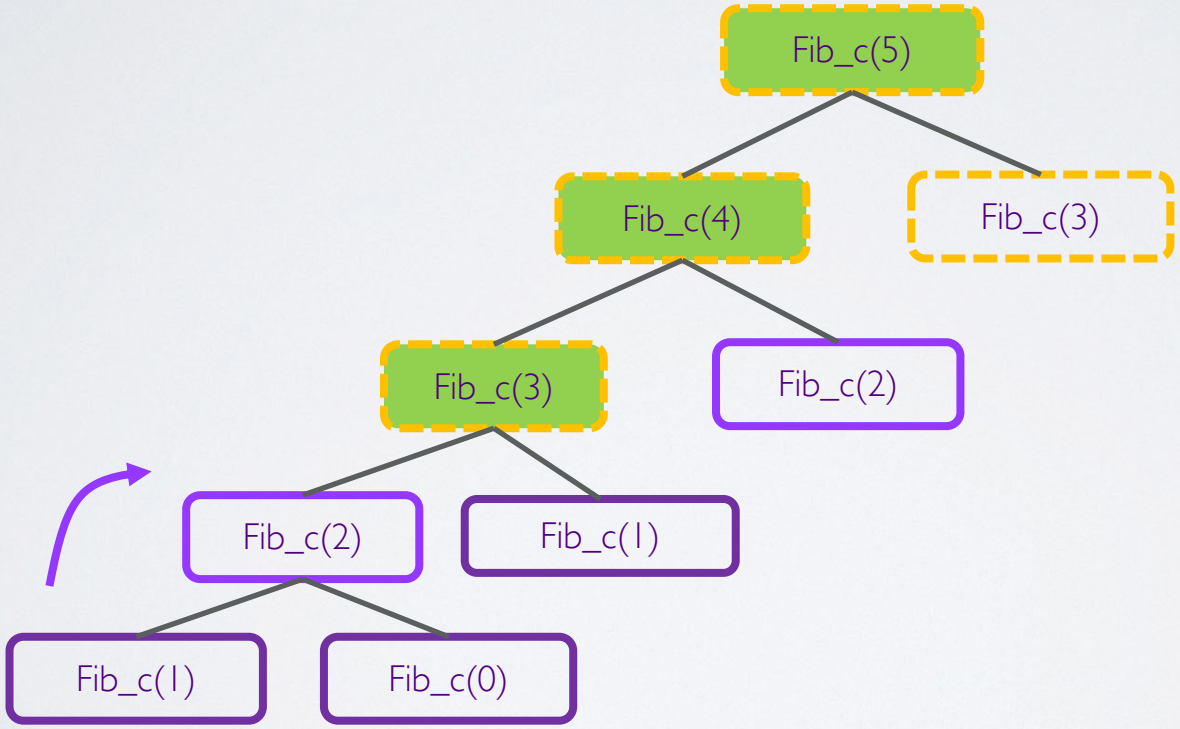
SIRUL LUI FIBONACCI - PD CU "MEMOIZATION" (CACHING, MEMORARE)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



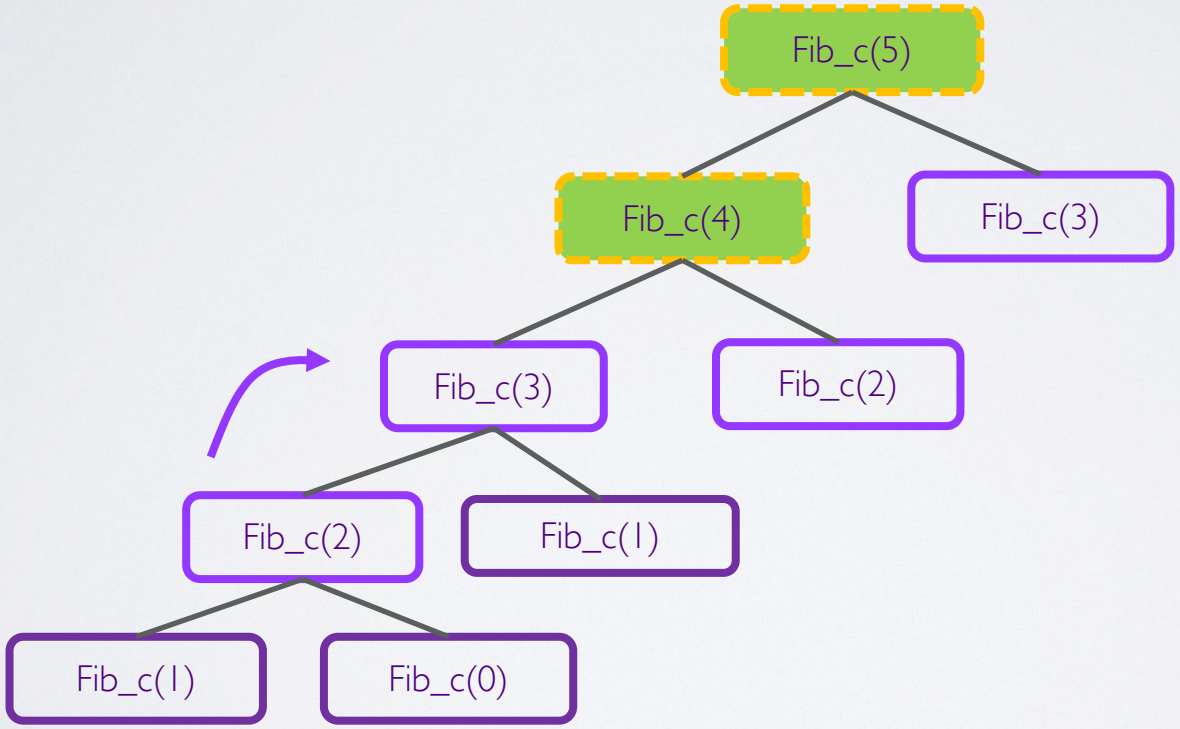
SIRUL LUI FIBONACCI - PD CU "MEMOIZATION" (CACHING, MEMORARE)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



La revenirea din recursivitate se schimba f[i]

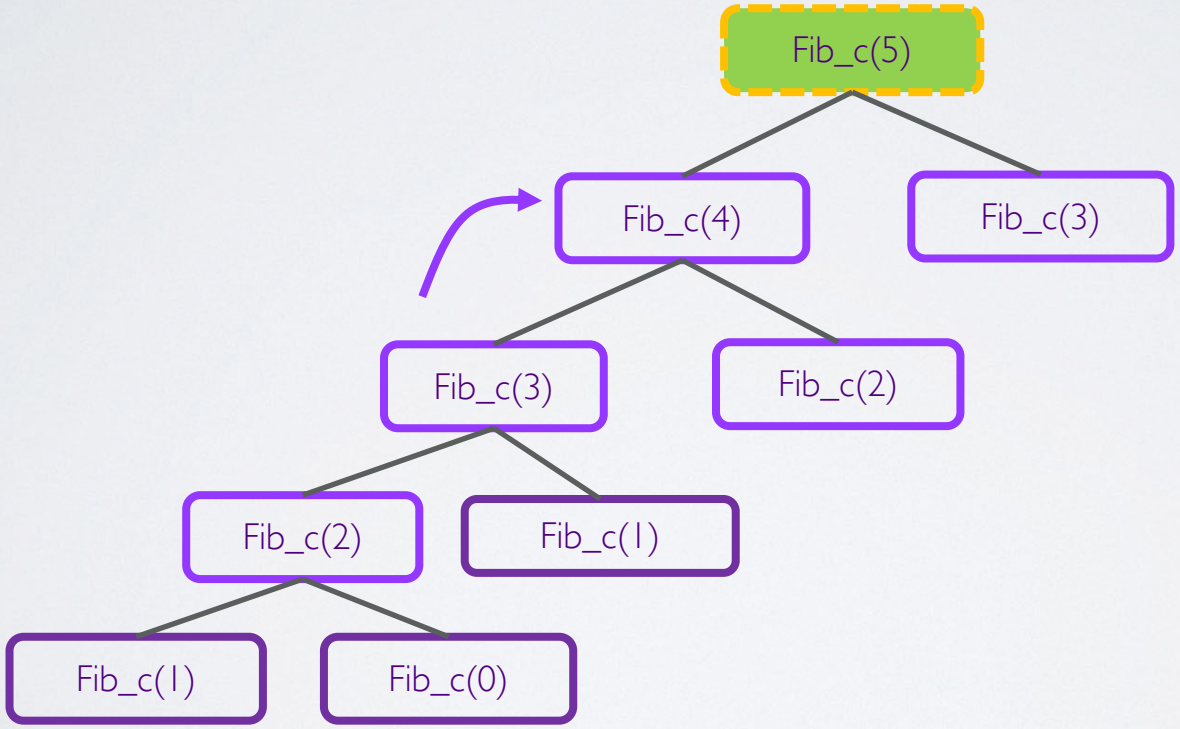
SIRUL LUI FIBONACCI - PD CU "MEMOIZATION" (CACHING, MEMORARE)



La revenirea din recursivitate se schimba f[i]

SIRUL LUI FIBONACCI - PD CU "MEMOIZATION" (CACHING, MEMORARE)

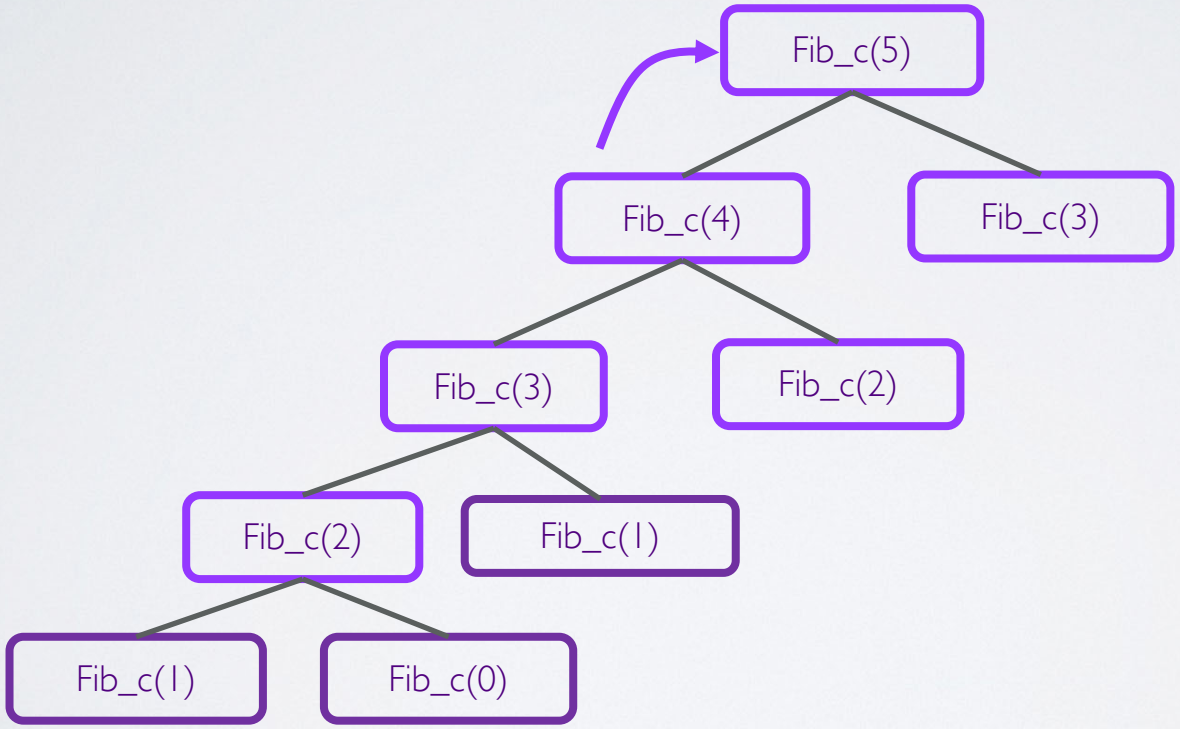
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



La revenirea din recursivitate se schimba f[i]

SIRUL LUI FIBONACCI - PD CU "MEMOIZATION" (CACHING, MEMORARE)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



La revenirea din recursivitate se schimba f[i]

SIRUL LUI FIBONACCI – PD BOTTOM UP

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

```
long fib_dp(int n)
{
  int i;  /* counter */
  long F[MAXN+1]; /* array to cache computed fib values */
  F[0] = 1;
  F[1] = 1;
  for (i=2; i<=n; i++)
    F[i] = F[i-1]+F[i-2];
  return(F[n]);
}
```

- Inversand ordinea de evaluare, nu mai avem deloc apeluri recursive
- Dar tot memorie *O(n)*

SIRUL LUI FIBONACCI – PD BOTTOM UP

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

```
long fib_dp(int n)
{
  int i;  /* counter */
  long F[MAXN+1]; /* array to cache computed fib values */
  F[0] = 1;
  F[1] = 1;
  for (i=2; i<=n; i++)
    F[i] = F[i-1]+F[i-2];
  return(F[n]);
}
```

- Inversand ordinea de evaluare, nu mai avem deloc apeluri recursive
- Dar tot memorie *O(n)*

SIRUL LUI FIBONACCI – PD BOTTOM UP

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

```
long fib_dp(int n)
{
  int i;  /* counter */
  long F[MAXN+1]; /* array to cache computed fib values */
  F[0] = 1;
  F[1] = 1;
  for (i=2; i<=n; i++)
    F[i] = F[i-1]+F[i-2];
  return(F[n]);
}
```

- Inversand ordinea de evaluare, nu mai avem deloc apeluri recursive
- Dar tot memorie *O(n)*

SIRUL LUI FIBONACCI – PD BOTTOM UP

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

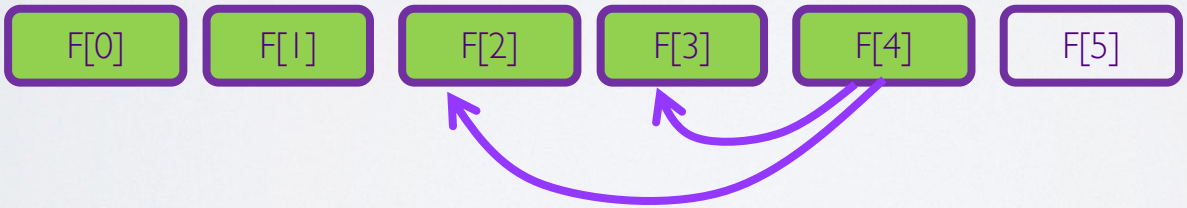
```
long fib_dp(int n)
{
  int i;  /* counter */
  long F[MAXN+1]; /* array to cache computed fib values */
  F[0] = 1;
  F[1] = 1;
  for (i=2; i<=n; i++)
    F[i] = F[i-1]+F[i-2];
  return(F[n]);
}
```

- Inversand ordinea de evaluare, nu mai avem deloc apeluri recursive
- Dar tot memorie *O(n)*

SIRUL LUI FIBONACCI – PD BOTTOM UP

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

```
long fib_dp(int n)
{
  int i;  /* counter */
  long F[MAXN+1]; /* array to cache computed fib values */
  F[0] = 1;
  F[1] = 1;
  for (i=2; i<=n; i++)
    F[i] = F[i-1]+F[i-2];
  return(F[n]);
}
```

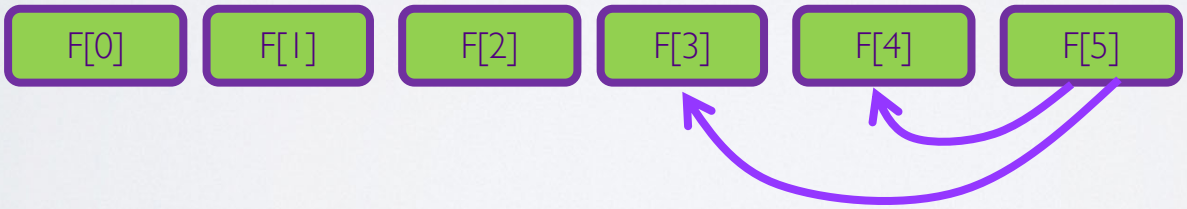


- Inversand ordinea de evaluare, nu mai avem deloc apeluri recursive
- Dar tot memorie *O(n)*

SIRUL LUI FIBONACCI – PD BOTTOM UP

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

```
long fib_dp(int n)
{
  int i;  /* counter */
  long F[MAXN+1]; /* array to cache computed fib values */
  F[0] = 1;
  F[1] = 1;
  for (i=2; i<=n; i++)
    F[i] = F[i-1]+F[i-2];
  return(F[n]);
}
```



- Inversand ordinea de evaluare, nu mai avem deloc apeluri recursive
- Dar tot memorie *O(n)*

SIRUL LUI FIBONACCI – PD BOTTOM-UP "FORGETTING"

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

```
long fib_ultimate(int n)
{
  int i;                    /* counter */
  long back2=1, back1=1;    /* last two values of f[n] */
  long next;                /* placeholder for sum */
  if (n == 0) return (0);
  for (i=2; i<n; i++) {
        next = back1+back2;
        back2 = back1;
        back1 = next;
  }
  return(back1+back2);
}
```

- Obs: avem nevoie doar de ultimele doua valori calculate
- Memorie *O(1)*
- *Ordinea de evaluare*

PROGRAMARE DINAMICA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Programarea dinamica:
  - Partitioneaza problema in subprobleme care nu sunt independente
  - Fiecare *subproblema este rezolvata doar o* data, rezultatul este stocat intr-o tabela pentru a evita re-calcularea
    - Consecinta: nr. relativ redus de sub-probleme pentru ca tabela sa fie calculabila
  - Permite transformarea unui algoritm exponential intr-unul polinomial
  - *Numele 'Programare Dinamica' se refera la calcularea tabelei; trebuie gasita totusi si forma ei!*

PROGRAMARE DINAMICA IN PROBLEME DE OPTIMIZARE

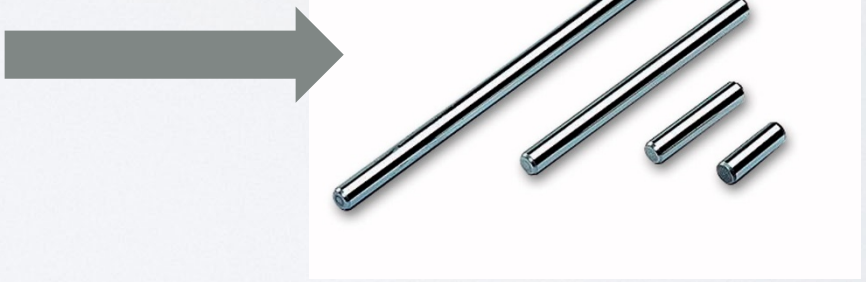
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Problemele de optimizare:
  - mai multe solutii posibile
  - fiecare solutie are o "calitate"; sarcina este gasirea solutiei de cea mai buna calitate - optima - minimizarea/maximizarea unei valori ce cuantifica calitatea
- Dezvoltarea unui algoritm PD - 4 pasi:
  1. Caracterizam **structura** solutiei optime
  2. Definim recursiv **valoarea** solutiei optime
  3. Calculam valoarea solutiei optime intr-o maniera de jos in sus *(bottom-up)*
  4. Construim solutia optima

TAIEREA TIJEI (ROD CUTTING)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- **Problema:**
- O companie cumpara tije de otel. Fiecare tija are lungimea n.
- Tijele sunt taiete in bucati de diferite lungimi.
- O taietura nu costa nimic.
- O tija cu o lungime *i* are un pret p_i
- Sa se determine cum se poate taia optim o tija de lungime *n* astfel incat compania sa maximizeze venitul.

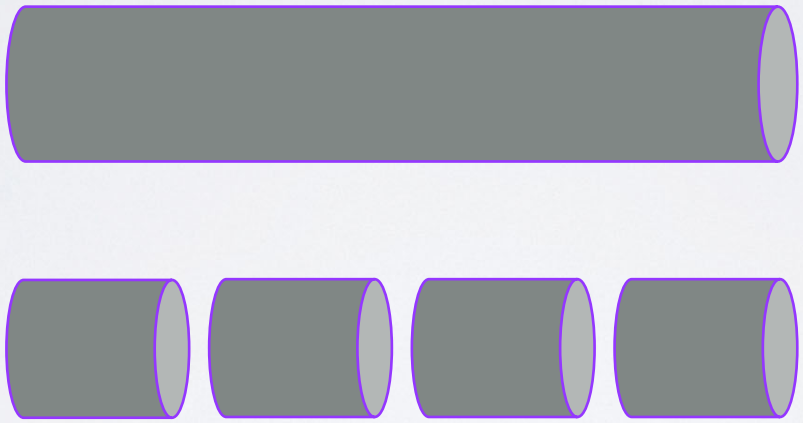


http://cf.fourgenerationsoneroof.com/wp-content/uploads/2013/06/how-to-make-a-curtain-rod-2.jpg

http://www.indiantradebird.com/admin/members/12739/images/0_6a8dc798.jpg

TAIEREA TIJEI

- Exemplu pentru tija de dimensiune n=4.
- Unde facem taieturile ?



TAIEREA TIJEI

- Exemplu pentru tija de dimensiune n=4.
- Unde facem taieturile ?

TAIEREA TIJEI

- Exemplu pentru tija de dimensiune n=4.
- Unde facem taieturile ?

TAIEREA TIJEI

- Exemplu pentru tija de dimensiune n=4.
- Unde facem taieturile ?

TAIEREA TIJEI

- Exemplu pentru tija de dimensiune n=4.
- Unde facem taieturile ?

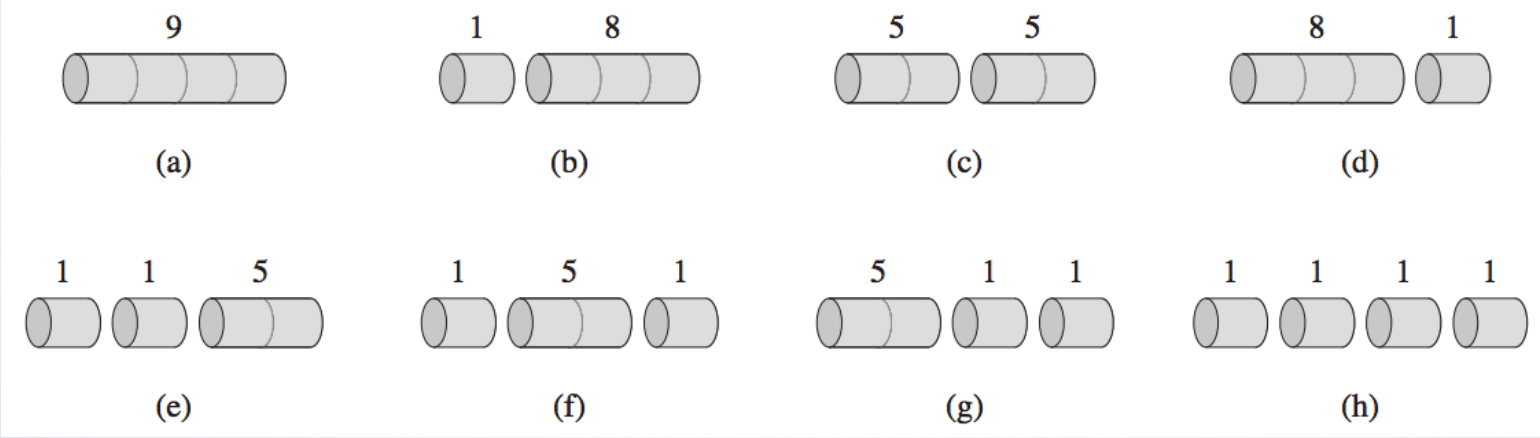
TAIEREA TIJEI (ROD CUTTING)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Pt. fiecare lungime *i* se cunoaste pretul p_i; o taiere nu costa nimic
- Se cere determinarea venitului maxim obtinut prin taierea tijei

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Ex: n=4; Cate solutii avem?



TAIEREA TIJEI (ROD CUTTING)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

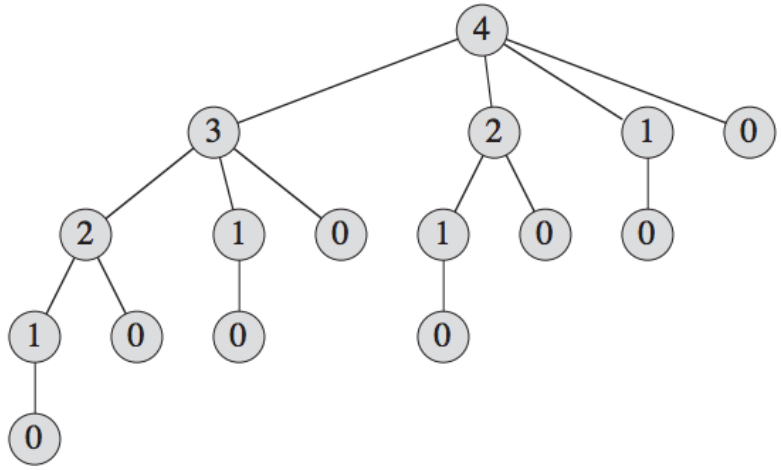
- 2^{n-1} solutii – la fiecare distanta i fata de marginea din stanga, avem 2 optiuni: taie / nu taia => $t(n) = 2*t(n-1) = 2^2*t(n-2) = \ldots = 2^{n-1}*t(1)$
- Fie r_i profitul maxim pentru o tija de lungime i
- Daca solutia optima taie tija in k bucati, fiecare avand lungimea i_j, avem: $n=i_1 + i_2 + \ldots + i_k$, respectiv $r_n = p_{i1} + p_{i2} + \ldots + p_{ik}$
- Formulam problema recursiv astfel (sub-structura optima):
  - Taie o bucata de un anumit profit de la capatul din stanga al tijei si vinde bucata (p_i)
  - Gaseste o metoda optima de a taia restul tijei (r_{n-i})
  - Recurenta pt. valoarea solutiei optime:

$$r_n = \max(p_n, p_1 + r_{n-1}, p_2 + r_{n-2}, \ldots, p_{n-1} + r_1)$$

TAIEREA TIJEI – ALGORITM NAIV

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```



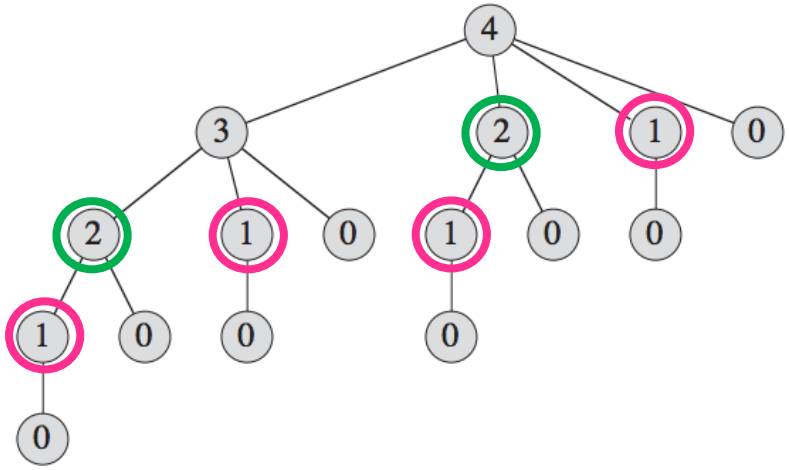
- Arborele cu apeluri recursive:
- Fiecare nod are ca eticheta dimensiunea subproblemei.

- O muchie de la nodul parinte s la un copil t inseamna ca se face o taietura de lungime s-t.
- Un drum de la radacina la frunza corespunde unei modalitati (din cele 2^{n-1}) de a taia tija.

TAIEREA TIJEI – ALGORITM NAIV

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```



- Se fac aceleasi calcule de mai multe ori !
- De exemplu pentru o lungime initiala a tijei n=4 calculam solutia optima pt n=2 de 2 ori !
- Pentru n=1 calculam solutia optima de 4 ori !

TAIEREA TIJEI – ALGORITM PD TOP DOWN

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Versiunea PD cu "memoizare" top-down:

- Stocam rezultatul apelurilor recursive si daca e nevoie in apelurile recursive folosim ce s-a calculat anterior.
  Rezultatul este stocal in r[n].

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

r[0]=-∞ | r[1] =-∞ | r[2] =-∞ | r[3] =-∞ | r[4] =-∞

TAIEREA TIJEI – ALGORITM PD TOP DOWN

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

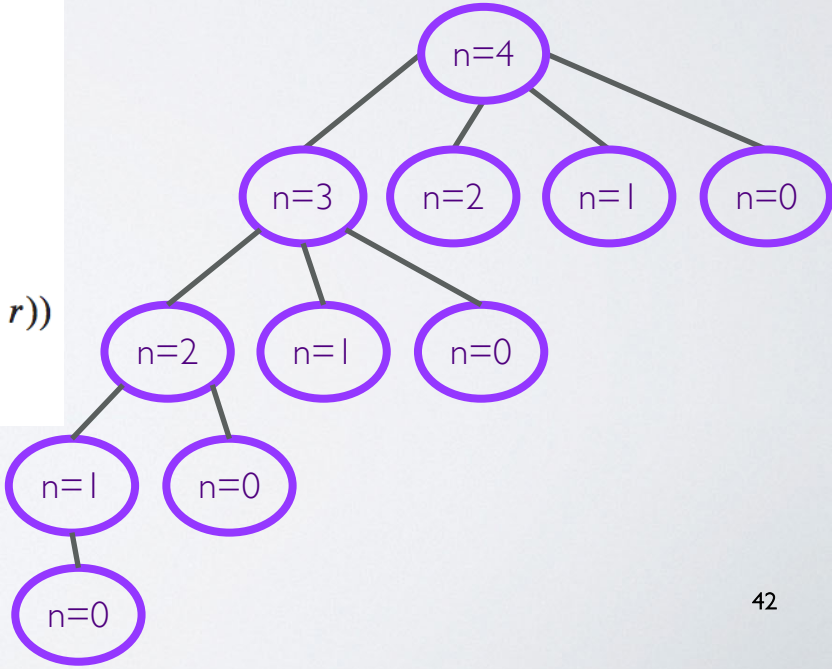
Versiunea PD cu "memoizare" top-down:

- Stocam rezultatul apelurilor recursive si daca e nevoie in apelurile recursive folosim ce s-a calculat anterior. Rezultatul este stocal in r[n].

length i	1	2	3	4
price p_i	1	5	8	9

r[0]=-∞ | r[1] =-∞ | r[2] =-∞ | r[3] =-∞ | r[4] =-∞

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2       return r[n]
3  if n == 0
4       q = 0
5  else q = −∞
6       for i = 1 to n
7            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```



TAIEREA TIJEI – ALGORITM PD TOP DOWN

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

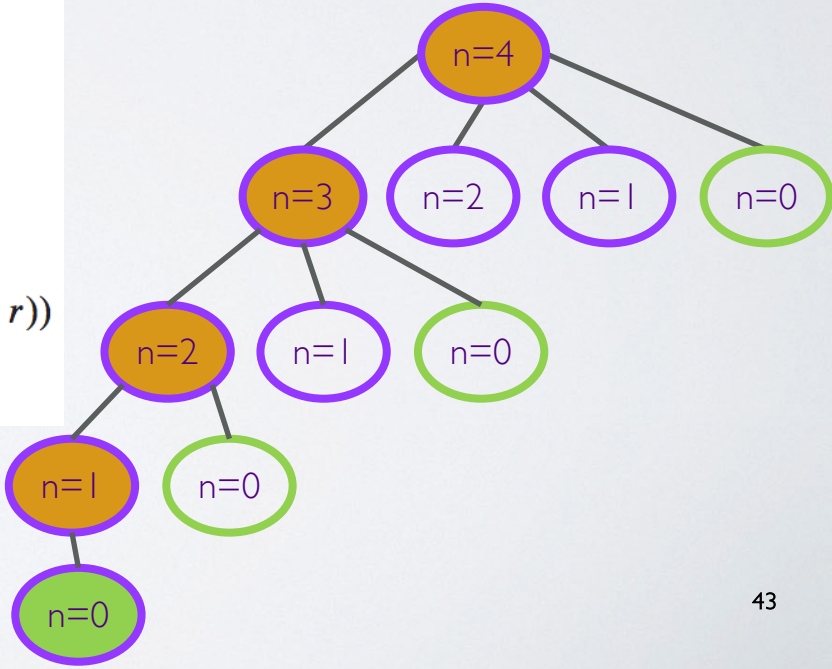
Versiunea PD cu "memoizare" top-down:

- Stocam rezultatul apelurilor recursive si daca e nevoie in apelurile recursive folosim ce s-a calculat anterior. Rezultatul este stocal in r[n].

length i	1	2	3	4
price p_i	1	5	8	9

r[0]=0 | r[1] =-∞ | r[2] =-∞ | r[3] =-∞ | r[4] =-∞

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2       return r[n]
3  if n == 0
4       q = 0
5  else q = −∞
6       for i = 1 to n
7            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```



TAIEREA TIJEI – ALGORITM PD TOP DOWN

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

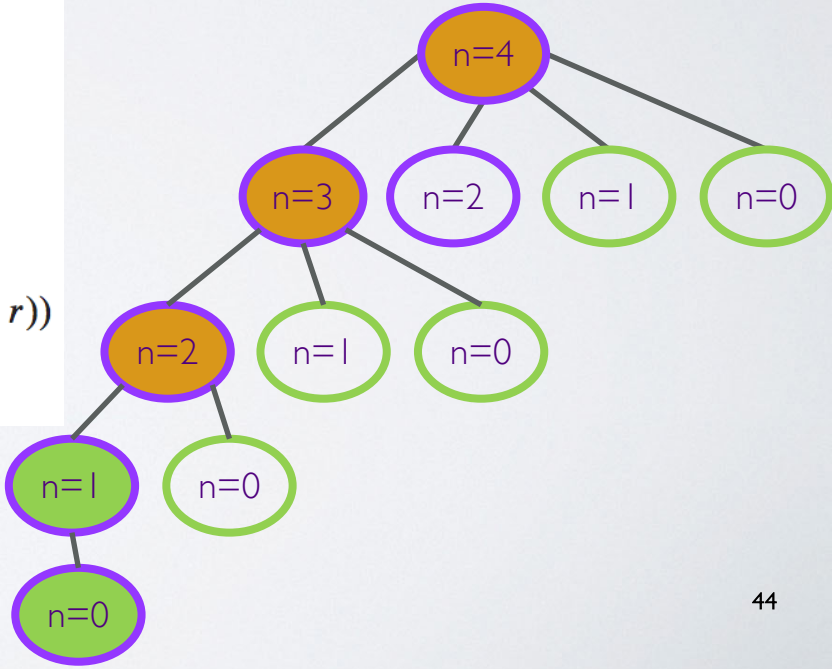
Versiunea PD cu "memoizare" top-down:

- Stocam rezultatul apelurilor recursive si daca e nevoie in apelurile recursive folosim ce s-a calculat anterior. Rezultatul este stocal in r[n].

length i	1	2	3	4
price p_i	1	5	8	9

r[0]=0 | r[1] =1 | r[2] =-∞ | r[3] =-∞ | r[4] =-∞

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```



r[1]=max(-∞, 1+0)

TAIEREA TIJEI – ALGORITM PD TOP DOWN

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

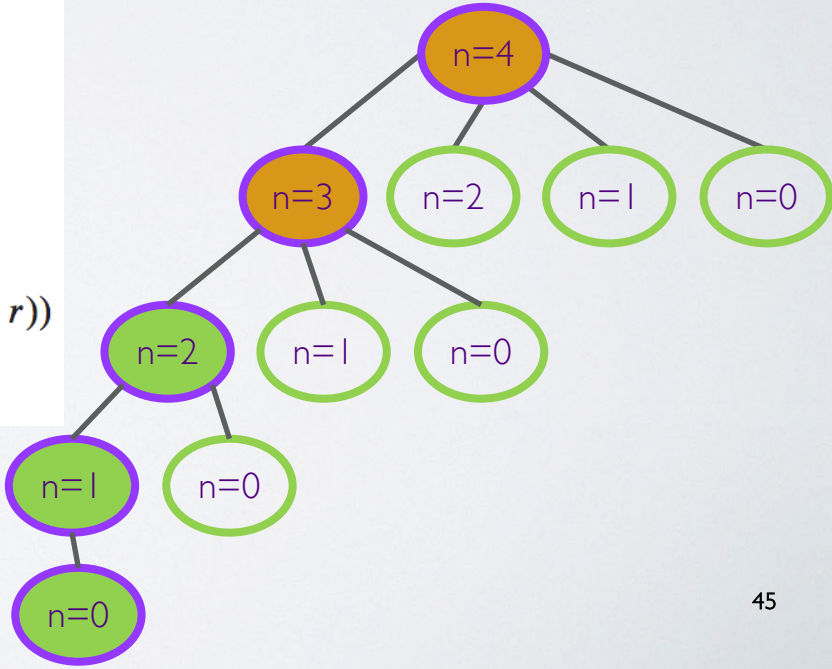
Versiunea PD cu "memoizare" top-down:

- Stocam rezultatul apelurilor recursive si daca e nevoie in apelurile recursive folosim ce s-a calculat anterior. Rezultatul este stocal in r[n].

length i	1	2	3	4
price p_i	1	5	8	9

r[0]=0 | r[1] =1 | r[2] = 5 | r[3] =-∞ | r[4] =-∞

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```



r[2]=max(-∞, 1+1, 5+0) = 5

TAIEREA TIJEI – ALGORITM PD TOP DOWN

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

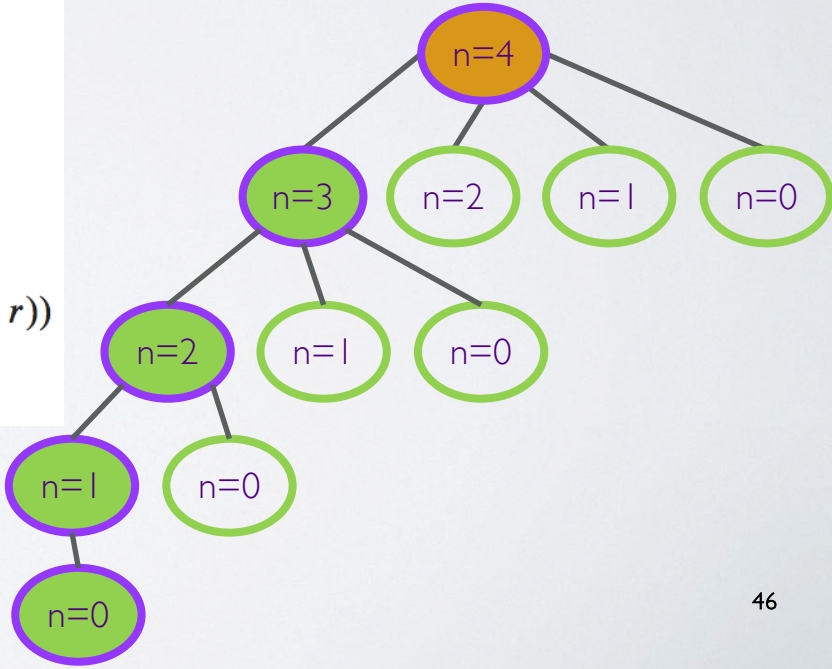
Versiunea PD cu "memoizare" top-down:

- Stocam rezultatul apelurilor recursive si daca e nevoie in apelurile recursive folosim ce s-a calculat anterior. Rezultatul este stocal in r[n].

length i	1	2	3	4
price p_i	1	5	8	9

r[0]=0 | r[1] =1 | r[2] = 5 | r[3] = 8 | r[4] =-∞

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2       return r[n]
3  if n == 0
4       q = 0
5  else q = −∞
6       for i = 1 to n
7            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```



r[3]=max(-∞, 1+5, 5+1,8) = 8

TAIEREA TIJEI – ALGORITM PD TOP DOWN

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

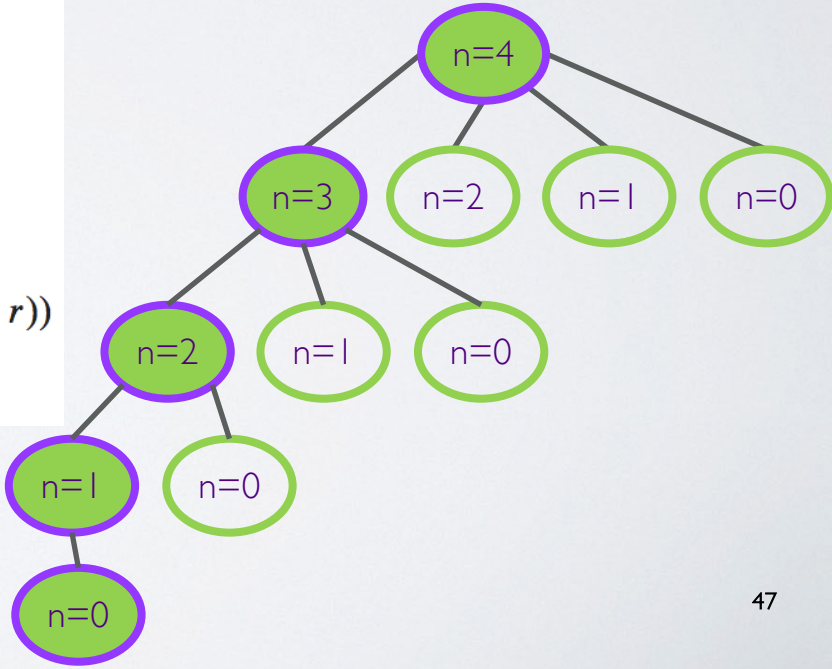
Versiunea PD cu "memoizare" top-down:

- Stocam rezultatul apelurilor recursive si daca e nevoie in apelurile recursive folosim ce s-a calculat anterior. Rezultatul este stocal in r[n].

length i	1	2	3	4
price p_i	1	5	8	9

r[0]=0 | r[1] =1 | r[2] = 5 | r[3] = 8 | r[4] =10

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2       return r[n]
3  if n == 0
4       q = 0
5  else q = −∞
6       for i = 1 to n
7            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```



r[4]=max(-∞, 1+8, 5+5, 8+1, 9) = 10

TAIEREA TIJEI – ALGORITM PD BOTTOM-UP

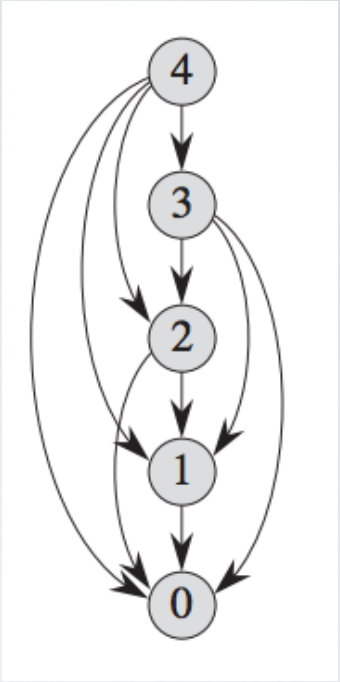
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

- Se calculeaza solutiile pentru tije de lungime mai mica stiind ca acestea vor fi folosite mai tarziu pentru a calcula solutiile pentru tije de lungime mai mare. Rezultatul este stocat in r[n].
- Ordonarea naturala a problemelor! Rezolvam problemele in ordinea "dimensiunii" lor
- Atat versiunea TD, cat si cea BU, au timp $O(n^2)$

TAIEREA TIJEI - GRAFUL SUB-PROBLEMELOR

UNIVERSITATEA TEHNICĂ



- Multimea de sub-probleme implicate
- Interactiunea dintre ele
- Abordarea BU considera probl. in *ordinea invers topologica* fata de cum apar ele in graf
- Abordarea TD cu "memoizare" considera probl. in ordinea data de *DFS*
- Dimensiunea lui ne poate ajuta la estimarea complexitatii in timp (de regula |V+E|)

TAIEREA TIJEI – RECONSTRUCTIA SOLUTIEI

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- salvarea *deciziei* care a dus la valoarea optima, pt fiecare sub-problema
  - dimensiune primei partitii a tijei, s_j

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n - s[n]
```

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

LCS - LONGEST COMMON SUB-SEQUENCE (SUB-SECVENTA COMUNA DE LUNGIME MAXIMA)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Aplicatii:
  - spell-checking
  - bio-informatica: secvente de ADN
- Ce inseamna sub-secventa:
  - Fie sirul de caractere:

A	C	C	G	G	T	C
1	2	3	4	5	6	7

  - Exemple sub-secvente ale acestui sir:

A	C	C	G
1	2	3	4

A	C	T	C
1	2	6	7

C	C	C
2	3	7

  - Diferenta fata de subsir?

Se defineste ca **sub-secventa** a sirului $X = <x_1, x_2, \ldots, x_m>$ sirul $Z = <z_1, z_2, \ldots, z_k>$, unde exista o secventa crescatoare de indici $<i_1, i_2, \ldots, i_k>$ *in X a.i. pentru orice* $j=1,2,\ldots,k$ *avem* $x_{i_j} = z_j$

LCS - LONGEST COMMON SUB-SEQUENCE (SUB-SECVENTA COMUNA DE LUNGIME MAXIMA)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Problema de gasire a sub-secventei comune de lungime maxima
- Se dau doua siruri X si Y si se cere cea mai lunga sub-secventa care are proprietatea ca este sub-secventa si pentru X si pentru Y.
- Exemplu:

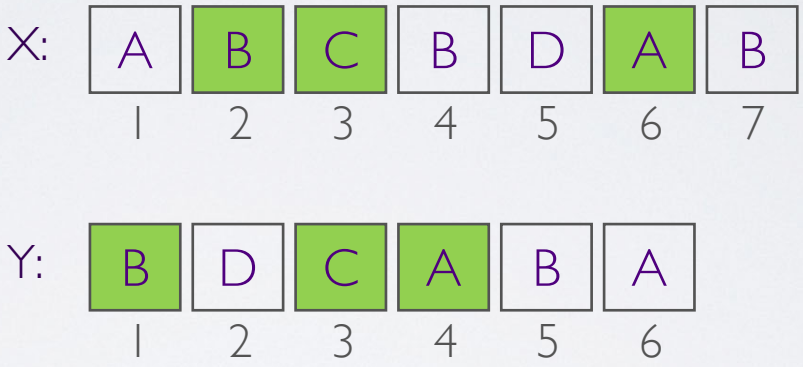
X:	A	B	C	B	D	A	B
	1	2	3	4	5	6	7

Y:	B	D	C	A	B	A
	1	2	3	4	5	6

LCS - LONGEST COMMON SUB-SEQUENCE (SUB-SECVENTA COMUNA DE LUNGIME MAXIMA)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Problema de gasire a sub-secventei comune de lungime maxima
- Se dau doua siruri X si Y si se cere cea mai lunga sub-secventa care are proprietatea ca este sub-secventa si pentru X si pentru Y.
- Exemplu:

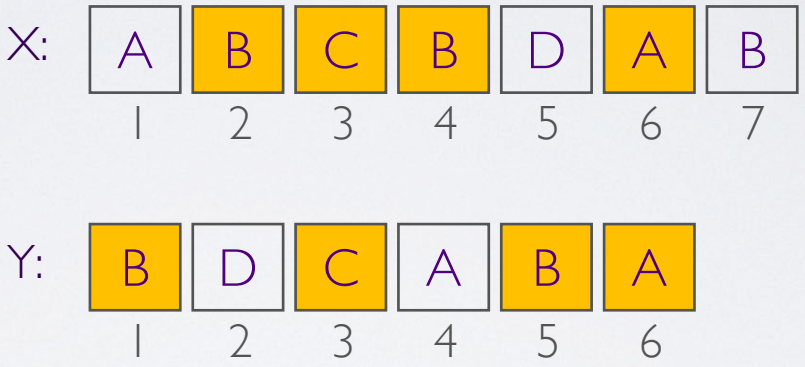


sub-secvente commune:

LCS - LONGEST COMMON SUB-SEQUENCE (SUB-SECVENTA COMUNA DE LUNGIME MAXIMA)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Problema de gasire a sub-secventei comune de lungime maxima
- Se dau doua siruri X si Y si se cere cea mai lunga sub-secventa care are proprietatea ca este sub-secventa si pentru X si pentru Y.
- Exemplu:

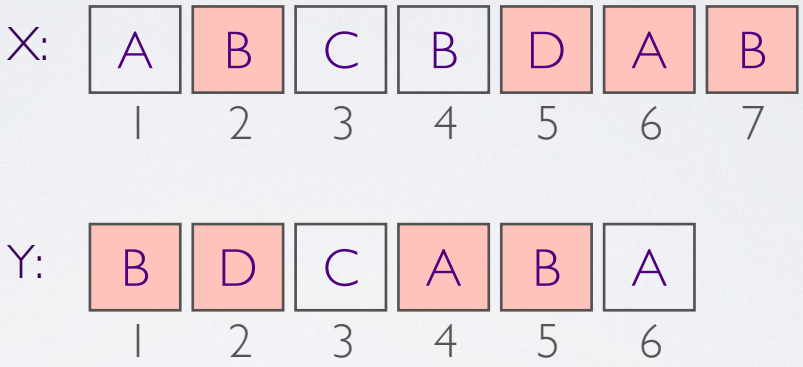


sub-secvente commune:

LCS - LONGEST COMMON SUB-SEQUENCE (SUB-SECVENTA COMUNA DE LUNGIME MAXIMA)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Problema de gasire a sub-secventei comune de lungime maxima
- Se dau doua siruri X si Y si se cere cea mai lunga sub-secventa care are proprietatea ca este sub-secventa si pentru X si pentru Y.
- Exemplu:



sub-secvente commune:

LCS - LONGEST COMMON SUB-SEQUENCE

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

(1) Proprietatea de **sub-structura optima**

- Definim *prefixul i al unui sir* $X=<x1, x2, \ldots, xm>$: $Xi = <x1, x2, \ldots, xi>$
- **Teorema: Sub-structura optima a unui LCS**

Fie $X=<x1, x2, \ldots, xm>$ *si* $Y=<y1, y2, \ldots, yn>$ - siruri, si $Z = <z1, z2, \ldots, zk>$ - orice LCS a lui X si Y:

(1) Daca $x_m=y_n$, atunci $z_k = x_m = y_n$ si Z_{k-1} este LCS a lui X_{m-1} si Y_{n-1}

(2) Daca $x_m!=y_n$, atunci daca $z_k!= x_m$ => Z este LCS a lui X_{m-1} si Y

(3) Daca $x_m!=y_n$, atunci daca $z_k != y_n$ => Z este LCS a lui X si Y_{n-1}

LCS - LONGEST COMMON SUB-SEQUENCE

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Teorema - demonstratie
  - (1) daca z_k != x_m, putem adauga x_m=y_n la Z si sa obtinem o secventa comuna de lungime k+1; demonstratia ca Z_{k-1} e LCS de lungime k-1 a lui X_{m-1} si Y_{n-1} se face prin red. la absurd
  - (2) daca z_k != x_m, atunci Z este sub-secventa comuna al lui X_{m-1} si Y. Daca ar exista W – subsecventa comuna a lui X_{m-1} si Y de lungime > k, W ar fi si sub-secventa comuna al lui X_m si Y (contradictie)
  - (3) simetric cu (2)

LCS - LONGEST COMMON SUB-SEQUENCE

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

(2) Definim recursiv valoarea solutiei optime

- avem de examinat 1 sau 2 sub-probleme pt a gasi LCS pt X si Y
  - $x_i = y_j$ - o problema; $x_i != y_j$ - 2 probleme

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

  - Conditiile din problema restrang sub-problemele (cate sub-probleme avem in total?)

LCS - LONGEST COMMON SUB-SEQUENCE

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

(3) Calculam lungimea LCS (valoarea optima) *O(mn)*

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

(4) Construirea LCS

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

O(m+n)

Nu avem de fapt nevoie de b pt a calcula LCS

Putem calcula eficient bazandu-ne doar pe c

LCS – EXEMPLU

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

m=7, n = 6

c[0..m, 0..n]

i		j	0	1	2	3	4	5	6
			Y_j	B	D	C	A	B	A
0	x_i								
1	A								
2	B								
3	C								
4	B								
5	D								
6	A								
7	B								

b[0..m, 0..n]

i		j	1	2	3	4	5	6
			Y_j	B	D	C	A	B	A
	x_i							
1	A							
2	B							
3	C							
4	B							
5	D							
6	A							
7	B							

LCS – EXEMPLU

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

m=7, n = 6

c[0..m, 0..n]

i		j	0	1	2	3	4	5	6
			Y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0						
2	B		0						
3	C		0						
4	B		0						
5	D		0						
6	A		0						
7	B		0						

b[0..m, 0..n]

i		j	1	2	3	4	5	6
			B	D	C	A	B	A
1	A							
2	B							
3	C							
4	B							
5	D							
6	A							
7	B							

LCS – EXEMPLU

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

m=7, n = 6

c[0..m, 0..n]

i		j	0	1	2	3	4	5	6
			Y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	0					
2	B		0						
3	C		0						
4	B		0						
5	D		0						
6	A		0						
7	B		0						

b[0..m, 0..n]

	j	1	2	3	4	5	6
i	y_i / x_j	B	D	C	A	B	A
1	A						
2	B						
3	C						
4	B						
5	D						
6	A						
7	B						

```
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

LCS – COMPLETATI – MATRICILE C SI B

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

m=7, n = 6

c[0..m, 0..n]

i		j	0	1	2	3	4	5	6
			Y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	0					
2	B		0						
3	C		0						
4	B		0						
5	D		0						
6	A		0						
7	B		0						

b[0..m, 0..n]

	j	1	2	3	4	5	6
i	y_i / x_j	B	D	C	A	B	A
1	A						
2	B						
3	C						
4	B						
5	D						
6	A						
7	B						

```
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

LCS – EXEMPLU

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

i		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

PROBLEMA RUCSACULUI

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Se dau n obiecte si un rucsac.
- Obiectul i are greutatea $w_i > 0$ si profit $v_i > 0$.
- Rucsacul are o capacitate W .
- Obiectiv: umplerea rucsacului astfel incat sa maximizam profitul total.

- Exemplu:
  - {1, 2, 5} are valoarea 35
  - {3,4} are valoarea 40
  - {3,5} are valoarea 46 dar depaseste greutatea rucsacului !

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance (weight limit W = 11)

PROBLEMA RUCSACULUI

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Definim *OPT*(*i*) = valoarea maximala a profitului pentru sub-problema ce contine obiectele 1, …, *i*.

- Cazul 1: *OPT* nu selecteaza obiectul *i*.
  - *OPT* selecteaza ce e mai bun din { 1, 2, …, *i* – 1 }.

- Cazul 2. *OPT* selecteaza obiectul *i*.
  - Selectia obiectului *i* nu inseamna ca vom respinge imediat celelate obiecte.
  - Fara a sti ce alte obiecte au fost selectate inaintea lui *i* nu stim nici macar daca avem loc suficient pentru *i*.

- Concluzie: Avem nevoie de mai multe sub-probleme !

PROBLEMA RUCSACULUI

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Adaugam o noua variabila !
- Definim OPT(i, w) = valoarea maximala a profitului pentru sub-problema ce contine obiectele 1, …, *i*, cu limita de greutate w
- Daca wi > w, nu se poate alege obiectul i, deci OPT nu selecteaza obiectul i
- Altfel
  - Cazul 1. OPT nu selecteaza obiectul i.
    - OPT selecteaza ce e mai bun din obiectele {1, 2, …, i-1} folosind limita de greutate w.
  - Cazul 2. OPT selecteaza obiectul i.
    - Noua limita de greutate este w – wi.
    - OPT selecteaza ce e mai bun din obiectele {1, 2, …, i-1} folosind noua limita de greutate.

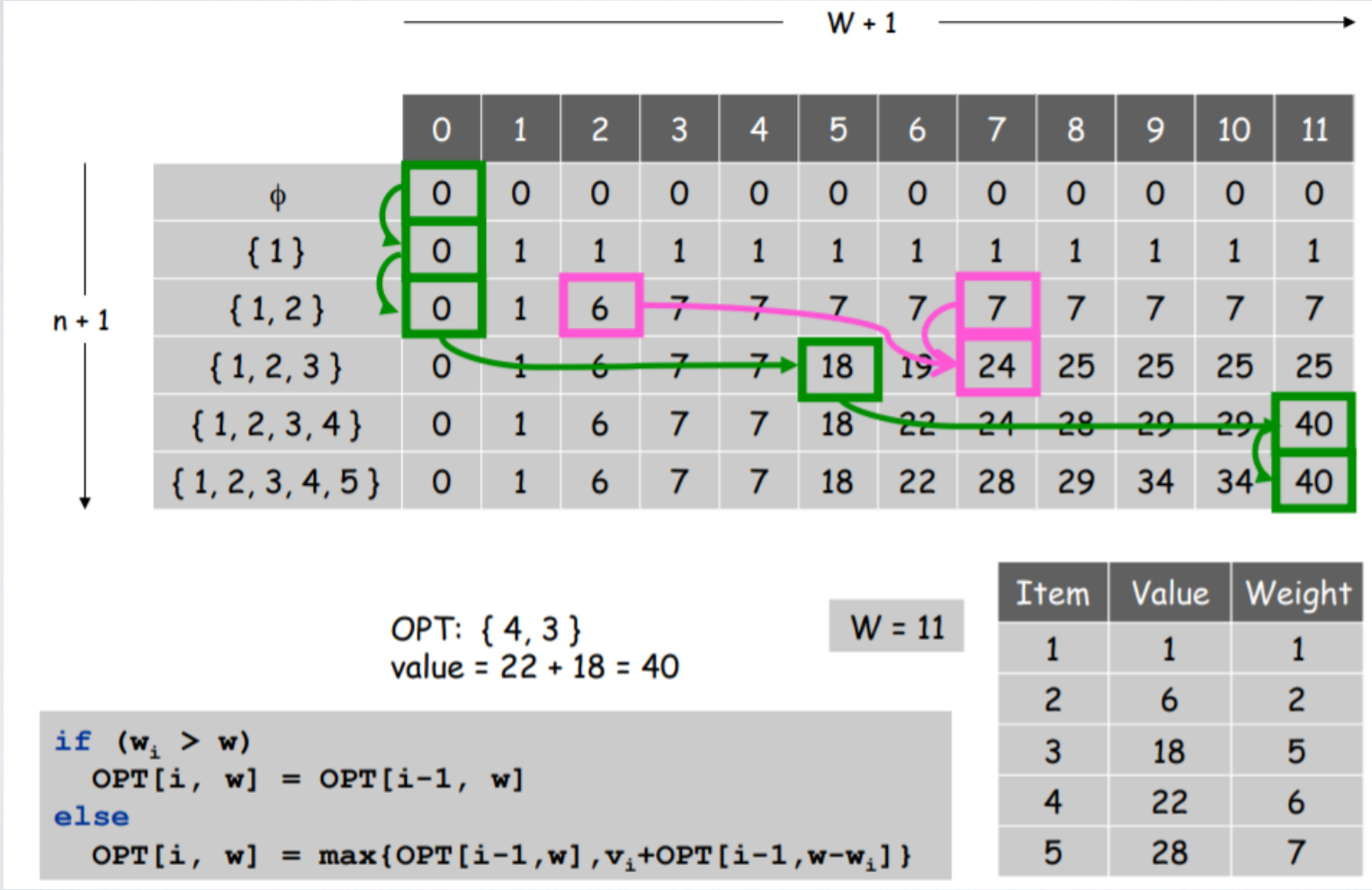
$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

PROBLEMA RUCSACULUI – PD BOTTOM UP

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

```
rucsac(n, W, w_1, … w_n; v_1, … v_n)
  for w = 0 to W
    OPT[0,w] ← 0
  for i = 1 to n
    for w = 0 to W
      if w_i > w
         OPT[i,w] ← OPT[i-1, w]
      else
         OPT[i,w] ← max{OPT[i-1,w], v_i+OPT[i-1, w-w_i]}
  return OPT[n,W]
```

PROBLEMA RUCSACULUI – PD BOTTOM UP DEMO



	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1, 2}	0	1	6	7	7	7	7	7	7	7	7	7
{1, 2, 3}	0	1	6	7	7	18	19	24	25	25	25	25
{1, 2, 3, 4}	0	1	6	7	7	18	22	24	28	29	29	40
{1, 2, 3, 4, 5}	0	1	6	7	7	18	22	28	29	34	34	40

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

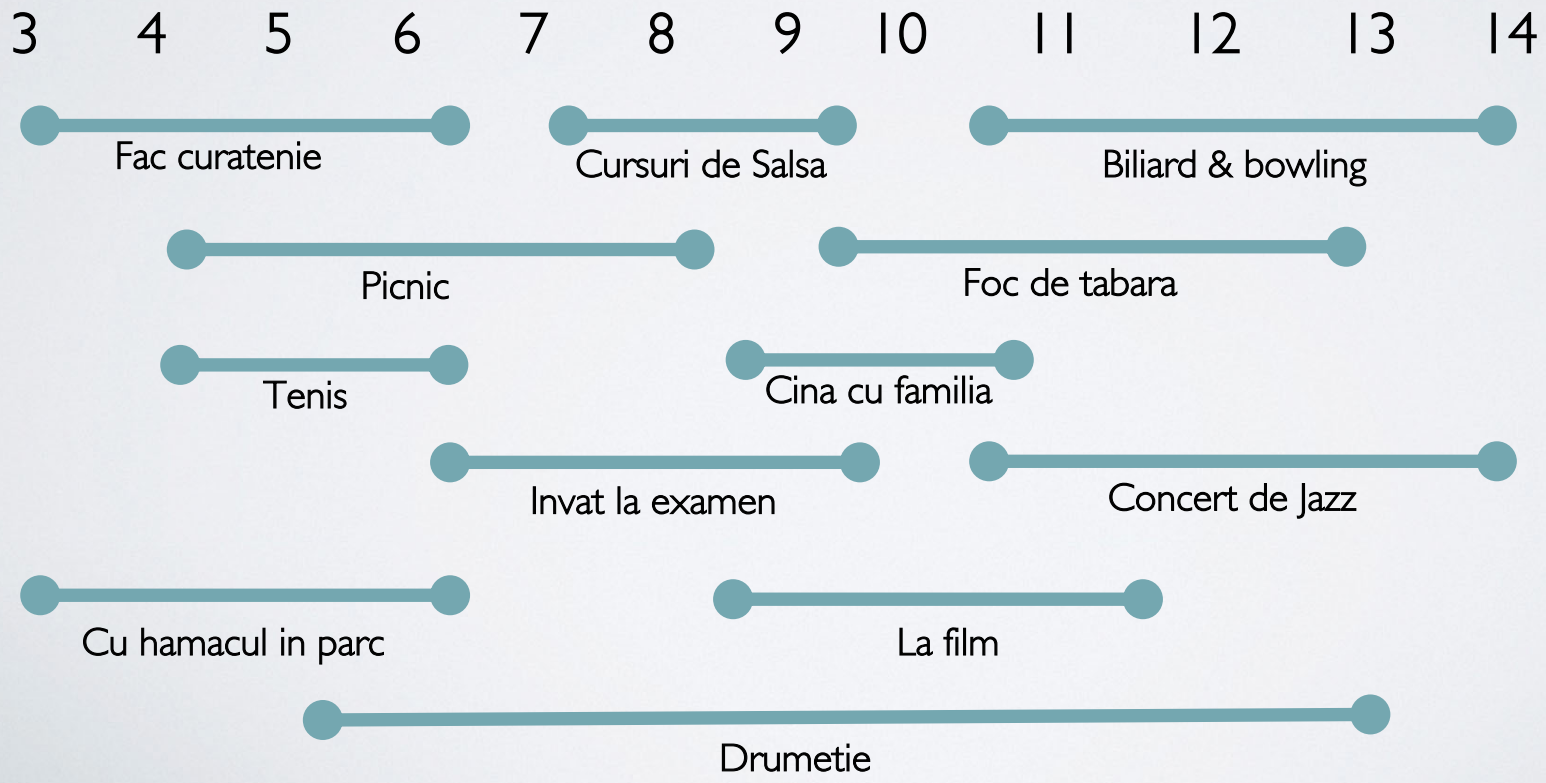
Sursa: https://courses.cs.washington.edu/courses/cse521/13wi/slides/06dp-sched.pdf

SELECTIA ACTIVITATILOR –

Se da o lista de activitati $S=\{a_1, a_2, \ldots a_n\}$, fiecare avand un timp de inceput si de sfarsit $a_i = (si, fi)$, $0 \leq s_i < f_i < \infty$

Activitatile pot fi la fel de atractive, sau nu (2 formulari alternative).

Dorim sa maximizam valoarea totala a activitatilor realizate intr-o zi



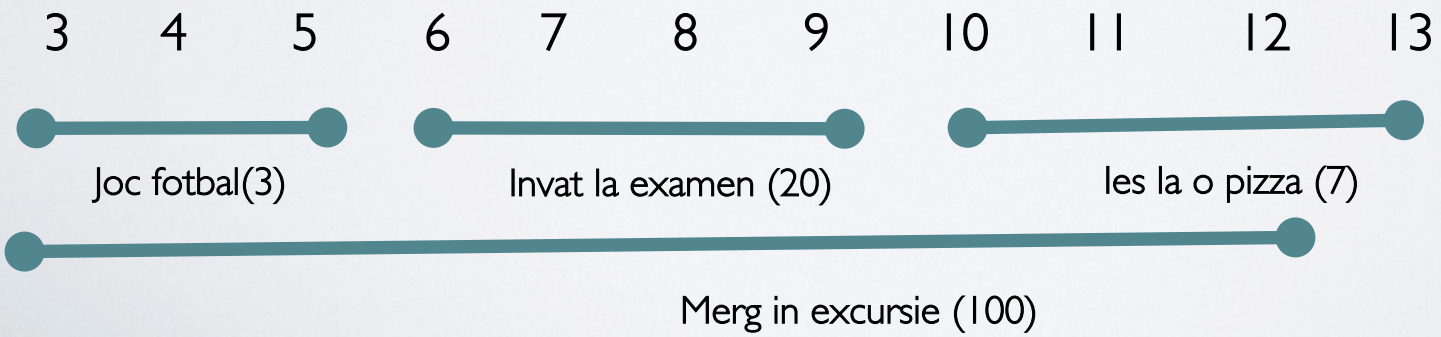
SELECTIA ACTIVITATILOR

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Scop: alegem o submultime de activitati care nu se suprapun, a.i. sa maximizam functia obiectiv (numar sau valoare totala)!
O idee (greedy – cursul urmator) – ordinea crescatoare a timpului de finalizare.
(Greedy ofera solutia optima daca fiecare activitate e la fel de atractiva, i.e. are aceeasi pondere).

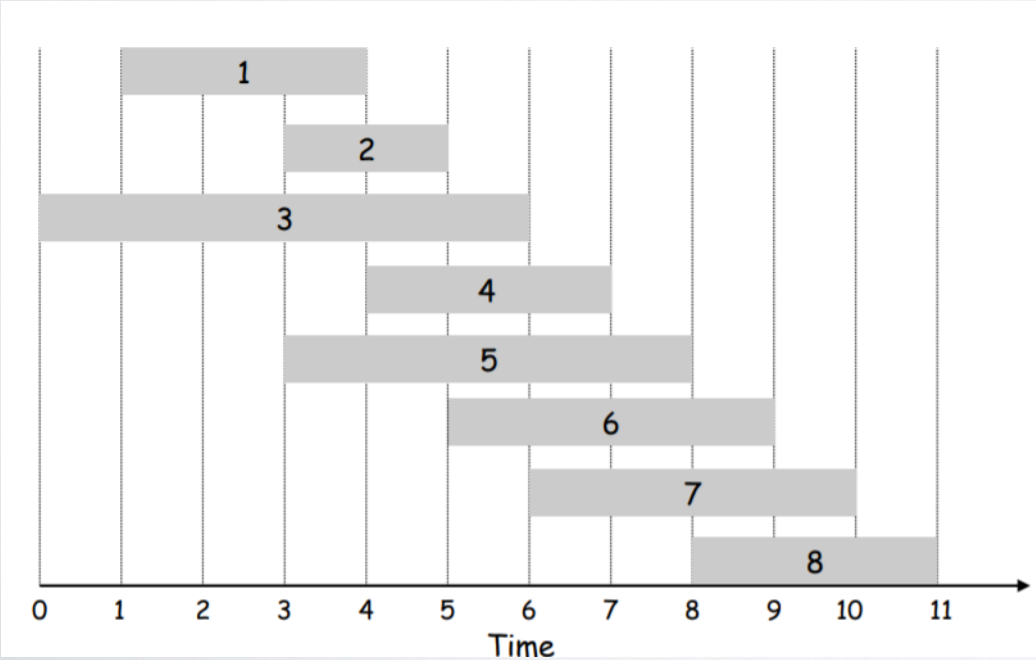
Dar daca activitatile nu sunt la fel de atractive – fiecare are o pondere diferita ?!

Ideea cu ordinea crescatoare a timpului de finalizare (greedy) – esueaza in acest caz !



SELECTIA ACTIVITATILOR

- Notatie: etichetam activitatile cu timpii lor de finalizare: $f_1 \leq f_2 \leq \ldots \leq f_n$
- Definim p(j) = cel mai mare indice k < j astfel incat activitatea k este compatibila cu activitatea j
- Ex. p(8) = 5, p(7) = 3, p(2) = 0.



j	vj	pj	optj
0	-	-	0
1		0	
2		0	
3		0	
4		1	
5		0	
6		2	
7		3	
8		5	

SELECTIA ACTIVITATILOR

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Notam *OPT(j)* = valoarea solutiei optime pentru problema care consta in activitatile 1, 2, … , j
- Cazul 1. *OPT* selecteaza activitatea *j*.
  - Colectam profitul *vj*.
  - Nu se pot folosi activitatile incompatibile {*p*(*j*)+1, *p*(*j*)+2, ..., *j* –1}.
  - Trebuie inclusa solutia optima la problema care consta din activitatile compatibile ramase din cele1, 2, ..., *p*(*j*).
- Cazul 2. *OPT* nu selecteaza activitatea *j*.
  - Trebuie inclusa solutia optima la problema care alege activitatile compatibile din cele ramase 1, 2, ..., *j* – 1.

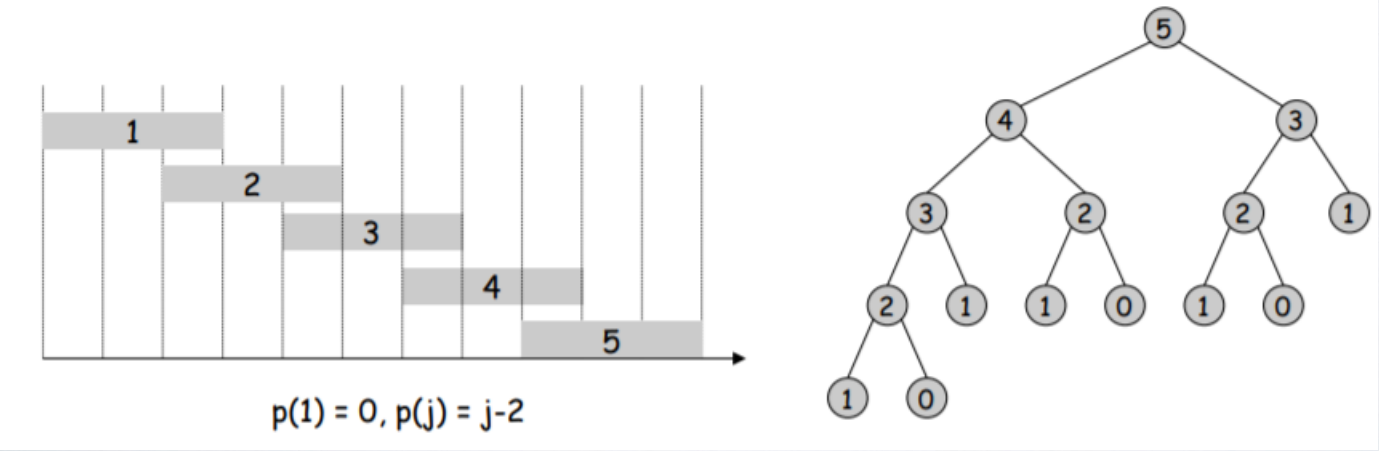
$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

SELECTIA ACTIVITATILOR

- Cautare exhaustiva – algoritm exponential :

Input: n, s[1..n], f[1..n], v[1..n]
Ordoneaza activitatile in functie de timpul de terminare astfel incat f[1] <=f[2]<=f[3] <= … f[n]
Calculeaza p[1], p[2], … , p[n]

Procedure compute_opt(j)
if j=0 then
    return 0
else
    return max(v[j] + compute_opt(p[j]), compute_opt(j-1))



SELECTIA ACTIVITATILOR

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Programare dinamica cu "memoizare":

```
Procedure main_activity_selection
  Input: n, s[1..n], f[1..n], v[1..n]
  Ordoneaza activitatile in functie de timpul de terminare astfel incat f[1] <=f[2]<=f[3] <= … f[n]
  Calculeaza p[1], p[2], … , p[n]
  for j = 1 to n
        M[j] = -1 //empty
  M[0] = 0
  memo_compute_opt(n)

Procedure memo_compute_opt(j)
  if  M[j] =-1
        M[j] = max(v[j] + memo_compute_opt(p[j]), compute_opt(j-1))
  return M[j]
```

- Complexitate : O(nlogn)

SELECTIA ACTIVITATILOR

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- PD gaseste valoarea maxima, dar cum gasim activitatile selectate ?
- Facem o traversare a lui M

```
Find-Solution(j)
if j = 0
   return ∅.
else if (v[j] + M[p[j]] > M[j-1])
   return {j} ∪ Find-Solution(p[j]).
else
   return Find-Solution(j-1).
```

SELECTIA ACTIVITATILOR

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- PD Bottom up

```
Procedure bottomUP_activity_selection
  Input: n, s[1..n], f[1..n], v[1..n]
  Ordoneaza activitatile in functie de timpul de terminare astfel incat f[1] <=f[2]<=f[3] <= … f[n]
  Calculeaza p[1], p[2], … , p[n]
  for j = 1 to n
        M[j] = -1 //empty
  M[0] = 0
  for j = 1 to n
    M[j] = max(v[j] + M[p[j]], M[j-1])
```

PROBLEMA NUMARARII RESTULUI

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- *Problema*: Un vanzator are la dispozitie o colectie de monede si bancnote de diferite valori. Se cere sa formeze o suma specificata folosind numarul minim de bancnote.
- Formulare matematica:
  - Se dau *n* bancnote si monede: $P=\{p_1, p_2, \ldots, p_n\}$
  - putem avea repetitii (2 bancnote de 1 leu, etc)
  - fie d_i valoarea lui p_i
  - Gasiti cea mai mica submultime S a lui P, $S \subseteq P$, astfel incat

$$\sum_{p_i \in S} d_i = A$$

unde A este suma de returnat.

PROBLEMA NUMARARII RESTULUI

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Tema: implementati rezolvarea acestei probleme folosind programarea dinamica.
- Relatia de recurenta pentru valoarea problemei:

$$C[p] = \begin{cases} 0 & if\ p = 0 \\ \min_{i:d_i \leq p}\{1 + C[p - d_i]\} & if\ p > 0 \end{cases}$$

Unde: C[p] este numarul minim de monede sau bancnote necesare pentru a plati suma p folosind valorile d_i.

Moneda selectata are valoarea mai mica decat suma p.

ELEMENTELE PROGRAMARII DINAMICE (RECAP)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- (1) Demonstram ca avem **substructura optima** – sol. optima contine in ea solutiile optime la sub-probleme
  - Gasirea solutiei unei probleme:
    - Efectuarea unei *alegeri* dintr-o serie de variante posibile (trebuie sa investigam care sunt acelea)
    - Rezolvarea *uneia sau mai multor* sub-probleme care sunt rezultatul unei alegeri (caracterizarea spatiului sub-problemelor)
  - Demonstram ca *solutiile* sub-problemelor trebuie sa fie *optime* pentru ca intreaga solutie sa fie optima (reducere la absurd - "cut-and-paste"; e.g. see LCS proof)

ELEMENTELE PROGRAMARII DINAMICE (RECAP)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- (2) Scrierea unei *recurente* pt. val. solutiei optime
  - $M_{opt} = \min_{\text{over all choices } k} \{(\sum M_{opt}$ toate sub-pb, rezultand din sel. *k*) + (costul asociat pentru a face selectia *k*)}
  - Demonstratie ca numarul de sub-probleme diferite este marginit superior de o functie polinomiala
- (3) Calcularea valorii solutiei optime intr-o maniera *bottom-up*, pt. a avea rezultatele necesare pre-calculate (sau *top-down* cu "*memoizare*")
  - Verificare daca se pot *reduce cerintele de memorie*, prin "uitarea" solutiilor la sub-probleme care nu mai sunt utilizate
- (4) Construirea solutiei optime din informatie pre-calculata (inregistram pe parcurs secventa de alegeri care ne duc la solutia optima)

PROGRAMARE DINAMICA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Concluzii:
  - Selectia la fiecare pas depinde de solutiile la sub-probleme
  - Versiune TD sau BU - cea BU e de regula mai eficienta
  - multe probleme sunt repetate in rezolvarea problemelor mai mari (e.g. taierea tijei)
    - aceasta repetitie faciliteaza eficientizarea la varianta BU

PROGRAMARE DINAMICA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- SCHEMA

//T - table of values of best sol. of problems of sizes smallest to P

```
for i in smallest sub-problem to P loop
  T(i) := MAX of:
    T(j) + cost of choice that changes sub-problem j into problem i
    T(k) + cost of choice that changes sub-problem k into problem i
     ... as many sub-problems as needed
end loop
Result is T(P)
```

BIBLIOGRAFIE + EXTRA READ

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Videos:
  - https://www.youtube.com/watch?v=OQ5jsbhAv_M
  - https://www.youtube.com/watch?v=ENyox7kNKeY
- Suplimentar
  - S. Skiena: The Algorithm Design Manual, cap 8.
    - Coeficienti binomiali, *edit distance* (generalizare la LCS), cateva *war stories* interesante
  - Th. Cormen et al.: Introduction to Algorithms, cap. 15
    - Matrix chain multiplication, arbori binari de cautare optimi
  - https://www.cs.princeton.edu/~wayne/kleinberg-tardos/
  - https://web.stanford.edu/class/archive/cs/cs161/cs161.1138/

PROBLEME PROPUSE

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

1. Numere urate (ugly numbers) sunt acele numere care au ca si factori primi pe 2, 3 sau 5. Sirul 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, … arata primele 11 astfel de numere. Numarul 1 este inclus in sir. Cititi un numar n de la tastatura si gasiti al n-lea numar urat din sir.

```
Input : n = 7 Output : 8
Input : n = 10 Output : 12
Input : n = 15 Output : 24
Input : n = 150 Output : 5832
```

2. LCS pentru 3 string-uri:

Se dau trei siruri de caractere de lungime < 100. Gasiti cel mai lung subsir comun celor 3 siruri.
Exemplu:

```
Input : str1 = "abcd1e2"
        str2 = "bc12ea"
        str3 = "bd1ea"
Output: 3 LCS este "b1e” cu lungime = 3.
```

PROBLEME PROPUSE

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

3. Se da o suma de numere intregi positive si o valoare sum, determinati daca exista o submultime a multimii date care are suma elementelor egala cu sum.

```
Exemplu: set[] = {3, 34, 4, 12, 5, 2}, sum = 9
Iesire: True //Exista o submultime (4, 5) cu suma 9.
```

4. Suma perfecta: se da un sir de numere intregi si o valoare sum. Afisati toate submultimile sirului care au suma egala cu sum

```
Intrare : set[] = {2, 3, 5, 6, 8, 10} sum = 10;
Output :  5 2 3
          2 8
          10
Intrare: set[] = {1, 2, 3, 4, 5} sum = 10;
Output :  4 3 2 1
          5 3 2
          5 4 1
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

BACKTRACKING
PROGRAMARE DINAMICA
RECAPITULARE

BACKTRACAKING – RECAPITULARE

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Metoda backtracking se aplică algoritmilor pentru rezolvarea următoarelor tipuri de probleme:
  - Fiind date n mulţimi S_1, S_2, ... S_n, fiecare având un număr nrs_i de elemente, se cere găsirea elementelor vectorului $A=(a1, a2, ... an) \in S=S_1aS_2a\ldots S_n$, astfel încât să fie îndeplinită o anumită relaţie $\phi(a1, a2, \ldots ,an)$ între elementele sale. Relaţia $\phi(a1, a2, \ldots ,an)$ se numeşte relaţie internă (condiţie internă), mulţimea $S=S_1aS_2a\ldots S_n$ se numeşte spaţiul soluţiilor posibile, iar vectorul A se numeşte soluţia rezultat. Metoda backtracking determină toate soluţiile rezultat ale problemei. Dintre acestea se poate alege una care îndeplineşte în plus o altă condiţie.
  - Această metodă se foloseşte în rezolvarea problemelor care îndeplinesc simultan următoarele condiţii:
    - mulţimile S_1, S_2, ... S_n sunt mulţimi finite, iar elementele lor se consideră că se află într-o relaţie de ordine bine stabilită (de regulă sunt termenii unei progresii aritmetice);
    - nu se dispune de o altă metodă de rezolvare, mai rapidă;
    - a_1, a_2, …, a_n pot fi la rândul lor vectori;
    - S_1, S_2, ... S_n pot fi identice.

BACKTRACKING – FORMA RECURSIVĂ

```
#define nmaa 100//numărul maaim de mulțimi
/* se consideră declarate global vectorii care mulţimile Si şi numărul lor de elemente nrsi */
int a[nmaa],n,k,nrs[nmaa];
int valid(int k){
            return (φ(a[1], a[2], ..., a[k])==1);
}
int soluție(int k){….}//de eaemplu {return k==n+1}

void backtracking_recursiv(int k)
{
            if(soluție(k)) afisare(a,n);
            else { //generare candidate
                        construct_candidates(a,k,S,c,&nc);
                        for (int i=0;i<nc;i++){
                                    a[k]=c[i]; /* al i-lea element din mullţimea de candidati */
                                    if (valid(k))backtracking_recursiv(k+1);
                        }
            }
}

int main(){
            citire();
            backtracking_recursiv(0);
            return 0;
}
```

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

BACKTRACKING – TIPURI DE PROBLEME LA CARE SE APLICA:

1. Probleme în care vectorul soluţie are lungime fixă şi fiecare element apare o singură dată în soluţie;

2. Probleme în care vectorul soluţie are lungime variabilă şi fiecare element poate să apară de mai multe ori în soluţie;

3. Probleme în plan, atunci când spaţiul în care ne deplasăm este un tablou bidimensional (backtracking generalizat).

BACKTRACKING – TIPURI DE PROBLEME LA CARE SE APLICA:

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

1. Probleme în care vectorul soluţie are lungime fixă şi fiecare element apare o singură dată în soluţie;
    1. Generarea produsului cartezian a n mulțimi
    2. Generarea submulţimilor unei mulţimi
    3. Generarea permutărilor unei mulţimi
    4. Generarea aranjamentelor
    5. Generarea submulţimilor cu m elemente ale unei mulţimi (combinări)
    6. Aranjarea a n regine pe o tablă de şah de dimensiune nxn fără ca ele să se atace
    7. Generarea tuturor secvenţelor de n (numar par) paranteze care se închid corect
    8. Generarea partitiilor unei mulțimi
    9. Colorarea ţărilor de pe o hartă astfel încât oricare două ţări vecine să aibă culori diferite
    10. Problema comisului voiajor

BACKTRACKING – TIPURI DE PROBLEME LA CARE SE APLICA:

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

2. Probleme în care vectorul soluţie are lungime variabilă şi fiecare element poate să apară de mai multe ori în soluţie:

- Partiţiile unui număr natural.
- Plata unei sume cu monede de valori date
- Submultimi de suma data.

BACKTRACKING – TIPURI DE PROBLEME LA CARE SE APLICA:

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

3. Probleme în plan, atunci când spaţiul în care ne deplasăm este un tablou bidimensional (backtracking generalizat):

- Problema labirintului
- Problema Bilei

PROBLEME PROPUSE – BACKTRACKING

Generarea tuturor submultimilor unei multimi cu numere {1,2,…n}

k	*a*				output
4	true	true	true	true	{ 1 2 3 4 }
	true	true	true	false	{ 1 2 3 }
3	true	true	false	true	{ 1 2 4 }
	true	true	false	false	{ 1 2 }
2	true	false	true	true	{ 1 3 4 }
	true	false	true	false	{ 1 3 }
	true	false	false	true	{ 1 4 }
	true	false	false	false	{ 1 }
1	false	true	true	true	{ 2 3 4 }
	false	true	true	false	{ 2 3 }
	false	true	false	true	{ 2 4 }
	false	true	false	false	{ 2 }
	false	false	true	true	{ 3 4 }
	false	false	true	false	{ 3 }
	false	false	false	true	{ 4 }
	false	false	false	false	{ }

Exemplu n = 4

Submultimile sunt: { 2 3 4 } , { 2 3 }, { 2 4 } , { 2 } , { 3 4 }, { 3 }, {4 } , { }, { 1 3 4 }, { 1 3 }, { 1 4 }, { 1 }, { 1 2 4}, { 1 2 }, { 1 2 3 4 }, {1 2 3 }

PROBLEME PROPUSE – BACKTRACKING

- Generarea tuturor submultimilor unei multimi cu numere {1,2,…n} care au maxim r elemente.

Exemplu n = 4

```
S[] = {1, 2, 3, 4}
r = 2
Submultimi : {1 2}; {1 3} {1 4} {2 3} {2 4} {3 4}
```

PROBLEME PROPUSE - BACKTRACKING

- Generarea tuturor submultimilor unei multimi cu numere {1,2,…n} care au suma egala cu K. Presupunem ca multimea contine doar numere positive si nu are duplicate.
  Eaemplu:
  arr[] = {1, 2, 3, 4, 5}

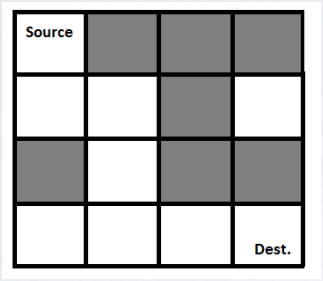
  K = 10

- Iesire : {4 3 2 1 }, {5 3 2 }, {5 4 1 }

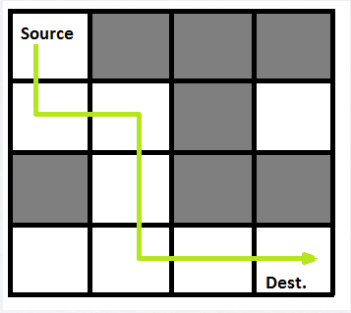
PROBLEME PROPUSE BACKTRACKING
SOARECELE IN LABIRINT

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Se da un labirint in care se gaseste un soricel. Labirintul este sub forma unei matrici. Valorile de 1 inseamna pozitii in care se poate deplasa soricelul si valori de 0 inseamna pozitii ocupate (nu se poate deplasa soricelul). Soarecele se poate
Deplasa in jos sau la dreapta fata de o celula. Gasiti un traseu de iesiere al soricelului din labirint.



Reprezentarea labirintului ca matrice
cu valori 1 (ok) si 0 (nu poate trece)
{1, 0, 0, 0}
{1, 1, 0, 1}
{0, 1, 0, 0}
{1, 1, 1, 1}



Solutie: {1, 0, 0, 0}
{1, 1, 0, 0}
{0, 1, 0, 0}
{0, 1, 1, 1}

PROGRAMARE DINAMICA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Pasii de rezolvare a unei probleme de programare dinamica:
  - Identifica daca problema se poate rezolva prin PD
  - Definiti o functie de stare cu cat mai putini parametrii
  - Formulati matematic relatia dintre stari
  - Realizati memoizarea

PROGRAMARE DINAMICA: IDENTIFICA DACA PROBLEMA SE POATE REZOLVA PRIN PD

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Toate problemele care cer sa se maaimizeze sau minimizeze o anumita cantitate sau probleme de numarare care cer sa se numere aranjamente in anumite conditii, sau animate probleme probabilistice – se pot rezolva cu PD
Toate problemele de PD satisfac doua proprietati:

1. Proprietatea subproblemelor care se suprapun

- PD combina solutii ale unor sub-probleme. PD se foloseste cand solutii ale acelorasi subprobleme sunt necesare iar si iar. In PD aceste solutii sunt stocate intr-un table astfel incat sa nu fie necesara recalcularea lor.
- Deci PD nu este utila cand nu eaista sub-probleme comune deoarece nu este necesar sa se stocheze rezultatele lor intermediare. De ea cautarea binara nu are sub-probleme commune , dar sirul lui Fibonacci are !

2. Proprietatea de substructura optimala:

- O anumita problema are proprietatea de substructura optimala daca Solutia optima a problemei se poate obtine folosind solutii optime ale sub-problemelor in care se descompune problema data.
- De eaemplu problema gasirii drumului cel mai scurt are urmatoarea substructure optimala: Daca un nod a se afla pe drumul cel mai scurt dintre un nod sursa u si un nod destinatie v atunci drumul cel mai scurt de la u la v este o combinatie de drumuri minimale de la u la a si drumuri minime de la a la v.

PROGRAMARE DINAMICA – SUBMULTIME DE SUMA DATA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Fiind data o multime de numere intregi positive si o valoare W, determinati daca exista o submultime a multimii date care sa aiba suma elementelor egala cu W.
- Pasii de PD:
  - Se poate descompune in subprobleme care se suprapun?
  - Putem formula o solutie optimala?
- Descompunem problema data in 2 subprobleme:
  1) Includem ultimul element, aplicam recursiv algoritmul pentru n=n-1 si suma=suma-set[n];
  2) Excludem ultimul element, aplicam recursive algoritmul pentru n=n-1;
- Daca oricare din subproblemele 1) sau 2) returneaza true atunci exista o solutie.

PROGRAMARE DINAMICA – SUBMULTIME DE SUMA DATA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Cum calculam solutia optima: OPT(j, w) folosind solutiile subproblemelor mai mici:

$$OPT(j, W) = \max \begin{cases} OPT(j-1, W) & \text{if } j \notin S^* \\ w_j + OPT(j-1, W - w_j) & \text{if } j \in S^* \end{cases}$$

$$OPT(0, W) = 0 \quad \text{If no items, 0}$$
$$OPT(j, 0) = 0 \quad \text{If no space, 0}$$

Special case: if $w_j > W$ then $OPT(j, W) = OPT(j-1, W)$.

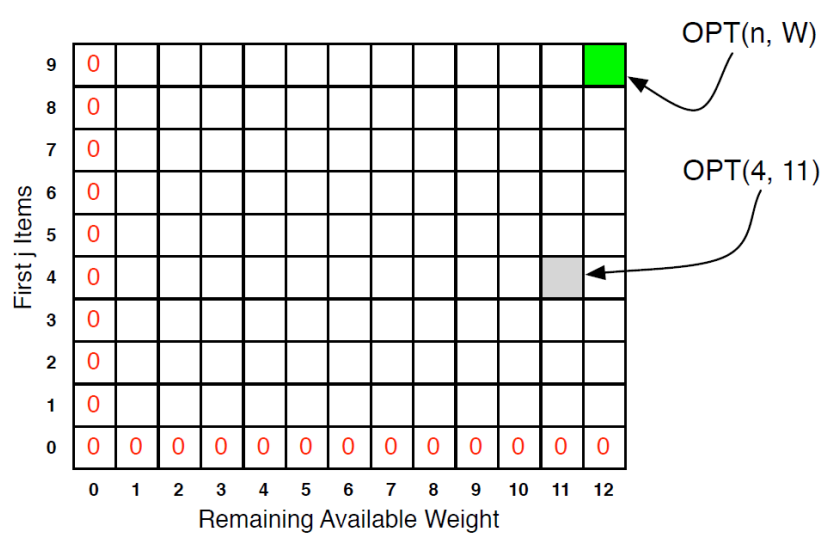
sau

$$OPT(j, W) = \begin{cases} 0 & \text{if } j = 0 \text{ or } W = 0 \\ OPT(j-1, W) & \text{if } w_j > W \\ \max \begin{cases} OPT(j-1, W) & \text{if } j \notin S^* \\ w_j + OPT(j-1, W - w_j) & \text{if } j \in S^* \end{cases} \end{cases}$$

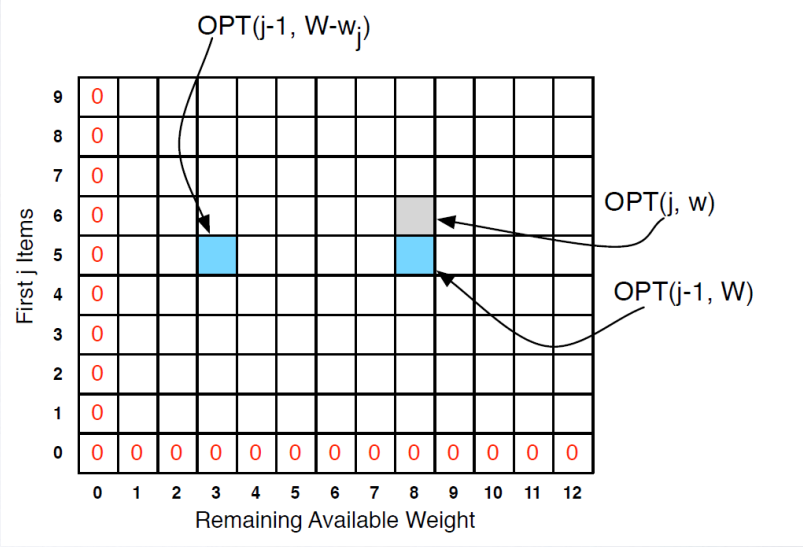
PROGRAMARE DINAMICA – SUBMULTIME DE SUMA DATA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Tabelul solutiilor



Calcularea unei solutii folosind solutiile subproblemelor mai mici



PROGRAMARE DINAMICA – SUBMULTIME DE SUMA DATA - PSEUDOCOD

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

```
SubsetSum(n, W):
   Initialize M[0,r] = 0 for each r = 0,...,W
   Initialize M[j,0] = 0 for each j = 1,...,n

   For j = 1,...,n:                  for every row
      For r = 0,...,W:               for every column
         If w[j] > r:                case where item can't fit
            M[j,r] = M[j-1,r]

         M[j,r] = max(               which is best?
            M[j-1,r],
            w[j] + M[j-1, W-w[j]]
         )
   Return M[n,W]
```

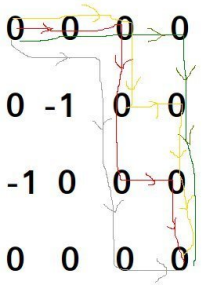
PROGRAMARE DINAMICA: RAT IN A MAZE

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

```
// If current cell is a blockage
 if (maze[i][j] == -1)
     maze[i][j] = -1; //  Do not change

 // If we can reach maze[i][j] from maze[i-1][j]
 // then increment count.
 else if (maze[i-1][j] > 0)
     maze[i][j] = (maze[i][j] + maze[i-1][j]);

 // If we can reach maze[i][j] from maze[i][j-1]
 // then increment count.
 else if (maze[i][j-1] > 0)
     maze[i][j] = (maze[i][j] + maze[i][j-1]);
```



-1 represents blockage.

(0,0) is source.

(3,3) is destination.

There are total four paths from source to destination

PROGRAMARE DINAMICA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Se da o multime formata din p numere sa se determine toate posibilitatile de a forma un numar N folosind suma celor p numere. Numerele care formeaza suma se pot repeat si se pot face diferite aranjamente.
Exemplu: p = 3 si multimea {1, 3, 5}, N = 6
Se va afisa: exista 8 metode de a forma suma si acestea sunt urmatoarele:

- 1+1+1+1+1+1
- 1+1+1+3
- 1+1+3+1
- 1+3+1+1
- 3+1+1+1
- 3+3
- 1+5
- 5+1