קורס יסודות התכנות בשפת C

פרק 12

רקורסיה - Recursion

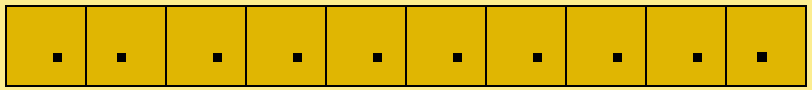
ד"ר שייקה בילו

יועץ ומרצה בכיר למדעי המחשב וטכנולוגית מידע

מומחה למערכות מידע חינוכיות, אקדמיות ומנהליות

תזכורת: מצביעים ומערכים

- הגדרת מערך מקצה בזיכרון מקומות רצופים עבור התאים שלו.
- בנוסף, מוקצה משתנה ששמו הוא שם-המערך, שמכיל את כתובת התא הראשון של המערך (לא ניתן לשנות את ערכו).
- לכן אפשר לגשת לתאי מערך גם ע"י חישוב הכתובת שלהם. למשל (arr+5)* שקול ל- arr [5].
- המחשב יודע לאיזה כתובת לגשת כשעושים את החשבון הזה, כי הוא יודע כמה מקום תופס כל תא במערך לפי סוג ערכים.



arr [0] arr [9]

500
arr
כתובת 500 (למשל)

גישה למערך לפי כתובת – דוגמא נוספת

- עבור הדפסת מערך, 3 האפשרויות הבאות עושות בדיוק אותו דבר:

```
int i,arr[10]={12,36,45,78,95,42,64,31,75,19};
int *ptr;
for (i=0; i<10; i++)
    printf("%d", arr[i]);
for (i=0; i<10; i++)
    printf("%d", *(arr+i));
for (ptr=arr; ptr <= &arr[9]; ptr++)
    printf("%d", *ptr);
```

- הדפסה על-ידי גישה רגילה לתאי המערך
- הדפסה על-ידי גישה לכל תא לפי הכתובת שלו יחסית לתא הראשון
- הדפסה ע"י קידום מצביע מכתובת התא הראשון עד כתובת התא האחרון

גישה למערך לפי כתובת – דוגמא נוספת

- עבור הדפסת מערך, 3 האפשרויות הבאות עושות בדיוק אותו דבר:

```
int i;
char *ptr, arr[12]=“Shalom Taly”;
```

- הדפסה על-ידי גישה רגילה לתאי המערך

```
for (i=0; i<12; i++)
    printf(“%c”, arr[i]);
```

- הדפסה על-ידי גישה לכל תא לפי הכתובת שלו יחסית לתא הראשון

```
for (i=0; i<12; i++)
    printf(“%c”, *(arr+i));
for (ptr=arr; ptr <= &arr[11]; ptr++)
    printf(“%c”, *ptr);
```

- הדפסה ע"י קידום מצביע מכתובת התא הראשון עד כתובת התא האחרון

מערכים ופונקציות - הסבר

- כפי שהוסבר בעבר כאשר מעבירים מערך לפונקציה, השינויים שנעשה בפונקציה ישפיעו על המערך המקורי.
- זה נובע מכך שלפונקציה מועברת הכתובת בזיכרון של המערך המקורי, והיא פועלת ישירות על ערכיו (ולא על העתק שלהם).
- למשל, כפי שאמרנו, הפונקציה הבאה תאפס מערך שמועבר אליה:

```
void zero_arr (int arr[ ], int size)
{
    int i;
    for(i=0 ; i<size; i++)
        arr[i]=0;
}
```

מערכים ופונקציות - הסבר

- אמרנו בעבר שכשמעבירים מערך לפונקציה שינויים שנעשה בפונקציה ישפיעו על המערך המקורי.
- זה נובע מכך שלפונקציה מועברת הכתובת בזיכרון של המערך המקורי, והיא פועלת ישירות על ערכיו (ולא על העתק שלהם).
- כפי שאמרנו, הפונקציה הבאה תאפס מערך מועבר:

```
void zero_arr (int *arr, int size)
{
    int i;
    for(i=0 ; i<size; i++)
        arr[i]=0;
}
```

נציין שאם פונקציה מצפה לקבל מצביע, אפשר להעביר אליה מערך מהסוג הזה, כי בשני המקרים מה שמועבר הוא כתובת בזיכרון.

מערכים ופונקציות - הסבר

- למשל הפונקציות המחרוזתיות שהוזכרו הפועלות על מחרוזות (כמו strcmp, strlen) מוגדרות למעשה תמיד ואך ורק עבור מצביעים מטיפוס * char.
- באחריותנו להפעיל אותן באמת על מחרוזת (כלומר רצף תווים שמסתיים ב- '\0') ולא סתם על מצביע לתו (נקבל תוצאות שגויות אם אין את תו הסיום).

מערכים ופונקציות - הסבר

- כמו-כן, כזכור, אם פונקציה מוגדרת על מערך בגודל מסוים, ניתן להעביר אליה גם מערך בגודל אחר של אותו טיפוס (כי כאמור מה שמועבר זה רק כתובת ההתחלה).
- שוב, זה באחריותנו שתתבצע בפונקציה פעולה הגיונית ושהיא תדע מה גודל המערך ולא תחרוג ממנו.

מערכים ופונקציות – נקודה לתשומת לב

- אם ננסה להחזיר מפונקציה מערך שהגדרנו בתוכה או מצביע על מערך כזה, אז נקבל שגיאה.

```
int * zero_arr ()
{
    int arr [100]={0};
    return arr;
}
```

- זה מכיוון שמוחזרת כתובת של המערך, אבל כל מה שהוגדר בתוך הפונקציה נמחק כשהיא מסתיימת.

שאלות?

חזרה - מבנים - הגדרת מבנה חדש

- נקודה במישור מיוצגת על-ידי שתי קואורדינטות, X ו- Y.
- הגדרת מבנה שייצג נקודה תיראה למשל כך:

```
struct point
{
        double x;
        double y;
};
```

- ההגדרה הזו לא תופיע בתוך פונקציה, אלא בתחילת הקובץ (בד"כ מיד אחרי שורות ה- #include וה- #define).
- כעת ניתן להגדיר בתוכנית משתנים מהסוג הזה, כפי שנבחן בהמשך.

חזרה - מבנים - הגדרת מבנה חדש

- ניתן לגשת לכל אחד משדות המבנה ולכתוב/לקרוא אליו/ממנו, למשל:

```
int main()
{
    struct point P1,P2;
    P1.x = 6.5;
    P1.y = 7.2;
    P2.x = 4.6;
    P2.y = 2.9;
    printf("%.2lf\n", P1.y);
    return 0;
}
```

	P2		P1
x	4.6	x	6.5
y	2.9	y	7.2

(יודפס 7.2)

חזרה - מבנים - הגדרת מבנה חדש


```
#include <stdio.h>
struct point {
   double x;
   double y;
};
int main()
{
  struct point P1,P2;
  P1.x = 6.5;
  P1.y = 7.2;
  P2.x = 4.6;
  P2.y = 2.9;
  printf("%.2lf,%.2lf\n",P1.x,P1.y);
  return 0;
}
```

	P2		P1
x	4.6	x	6.5
y	2.9	y	7.2

חזרה - מבנים – כתיבה מקוצרת

- שמו של הטיפוס החדש שהגדרנו הוא struct point
- אפשר לחסוך את כתיבת המילה struct באמצעות הפקודה typedef, שמאפשרת לתת שם חדש לטיפוס-משתנה:

```
struct point
{
        double x;
        double y;
};
typedef   struct point    point_t;
```

טיפוס קיים שם חדש

חזרה - עוד על typedef


```
struct complex
{
   double real, image;
};
int main()
{
   struct complex c={5,7};
   struct complex *pc;
   pc=&c;
}
```

```
struct complex
{
     double real, image;
};
typedef struct complex
    complex_t;
int main()
{
    complex_t c={5,7};
    complex_t *pc;
    c=&pc;
}
```

חזרה - מצביעים ומבנים

- בדוגמא זו, כדי להגיע לשדות של P דרך ptr שמצביע על P, אפשר להשתמש ב- * כרגיל:

P.x = 3; שקול ל- (*ptr).x = 3;

P.y = 7; שקול ל- (*ptr).y = 7;

- יש צורך בסוגריים כי אחרת לנקודה יש קדימות על פני ה- *.

חזרה -מצביעים ומבנים – גישה לשדות המבנה

- בדוגמא זו, כדי להגיע לשדות של P דרך ptr שמצביע על P, אפשר להשתמש ב- * כרגיל:

P.x = 5; שקול ל- (*ptr).x = 5;

P.y = 8; שקול ל- (*ptr).y = 8;

- אבל בדרך-כלל נשתמש לצורך זה בסימון מקוצר: חץ ->

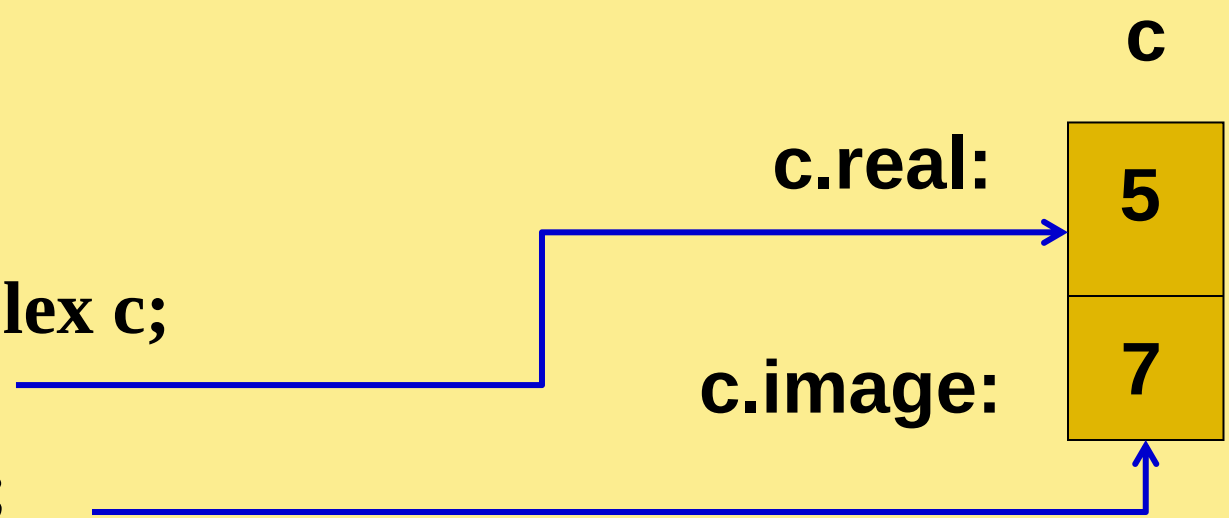
P.x = 5; שקול ל- ptr -> x = 5;

P.y = 8; שקול ל- ptr -> y = 8;

- כלומר משמעות החץ היא גישה לשדה במבנה שמצביעים עליו.

חזרה - מבנים Syntax - שימוש במצביעים


```
struct complex
{
    double real;
    double image;
};
int main()
{
   struct complex c;
   c.real = 5;
   c.image = 7;
}
```



חזרה - מבנים Syntax - כתובות ומצביעים


```
typedef struct {
    double real, image;
}comples;
int main()
{
    complex c;
    complex *pc;
    c.real = 5;      c.image = 7;
    pc = &c;
    pc->real = 3;
    pc->image = 4;
}
```

c → 0x7fff9255c05c

c.real:	5
c.image:	7

pc → 0x7fff9255c05c

pc.real:	3
pc.image:	4

pc → 0x7fff9255c05c

חזרה - מבנה בתוך מבנה

- אפשר להגדיר את זה על-ידי הקואורדינטות של שני הקודקודיים:

```
struct rect
{
        double xl, xh, yl, yh;
};
```

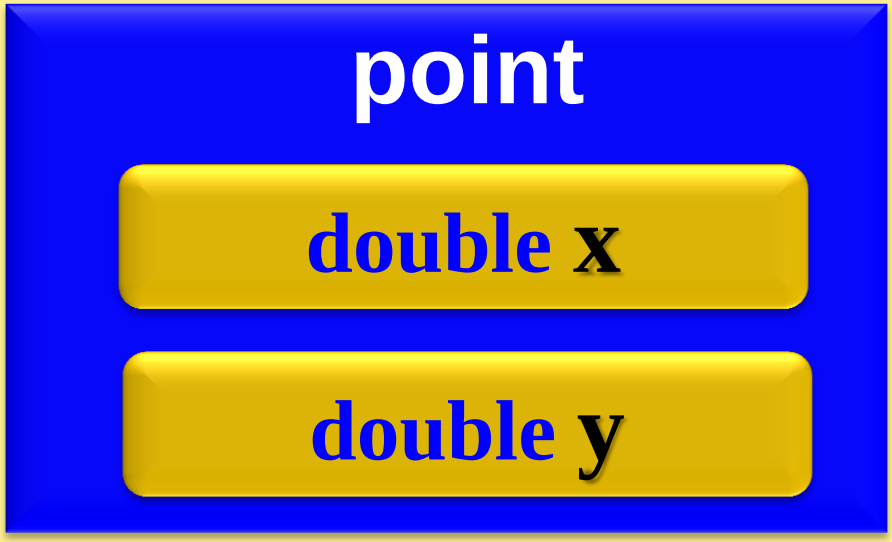
- ברור יותר להגדיר ע"י 2 נקודות.
- נדגים כאן גם את הרישום המקוצר עם typedef:

```
struct rect {
        point_t p;
        point_t q;
};
typedef struct rect rect_t;
```

חזרה - מבנים – כתיבה מקוצרת

- שמו של הטיפוס החדש שהגדרנו היה **struct point**
- אפשר לחסוך את כתיבת המילה **struct** באמצעות הפקודה **typedef**, שמאפשרת לתת שם חדש לטיפוס-משתנה:

```
typedef struct
{
   double x;
   double y;
} point;
```



חזרה - מערך של מבנים

- כמו טיפוסי משתנים אחרים, אפשר להגדיר מערך של מבנים.
- מערך זה יוגדר ויכיל בתוכו שדות שהם מבנים הכוללים תתי שדות.
- למשל, אפשר להגדיר מערך של נקודות:

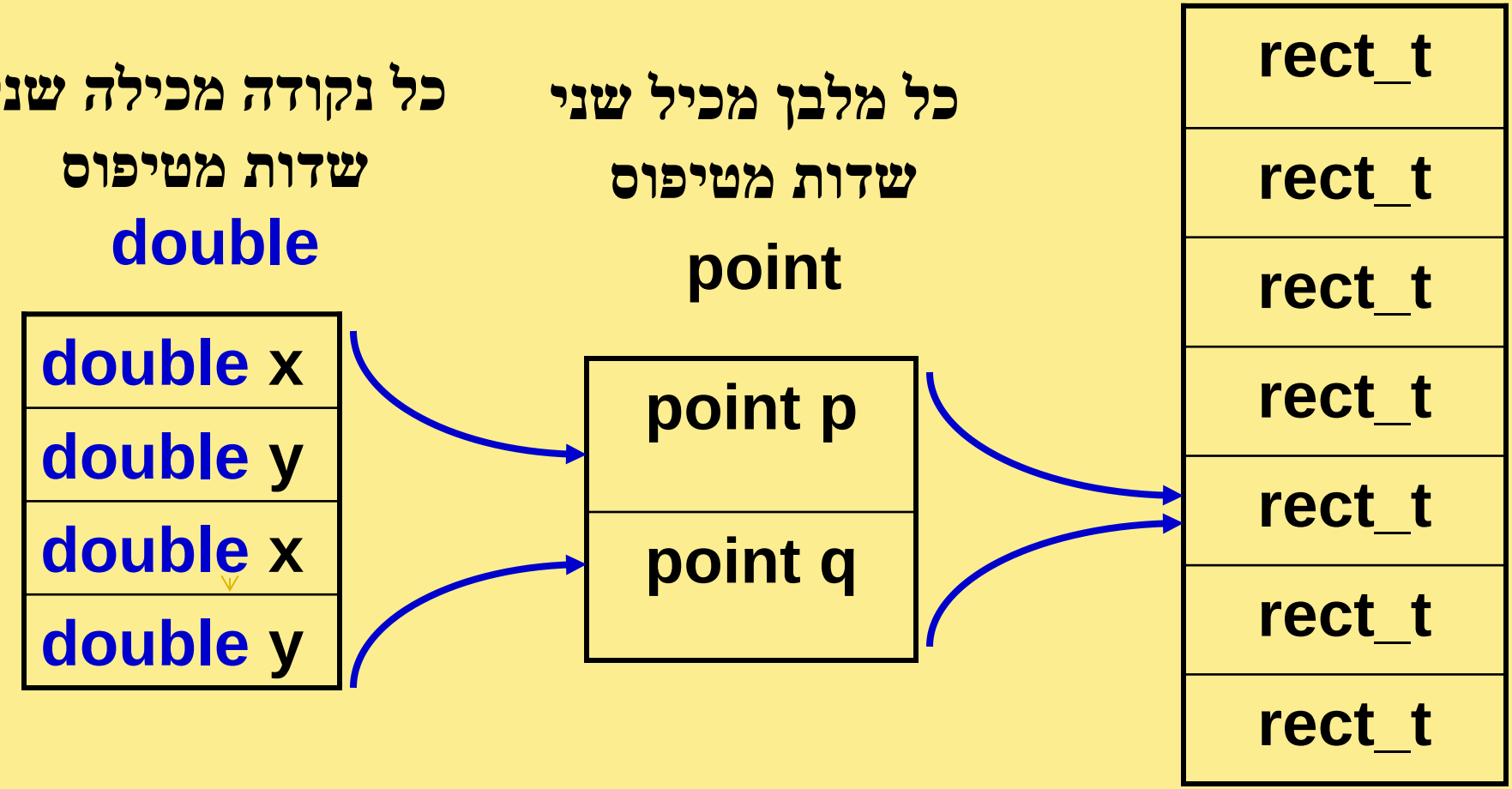
```
int main()
{
   point  arr[20];
   …
   …
   return 0;
}
```

חזרה - מערך של מלבנים בזיכרון

המלבנים נשמרים ברצף בזיכרון:

כל נקודה מכילה שני שדות מטיפוס double

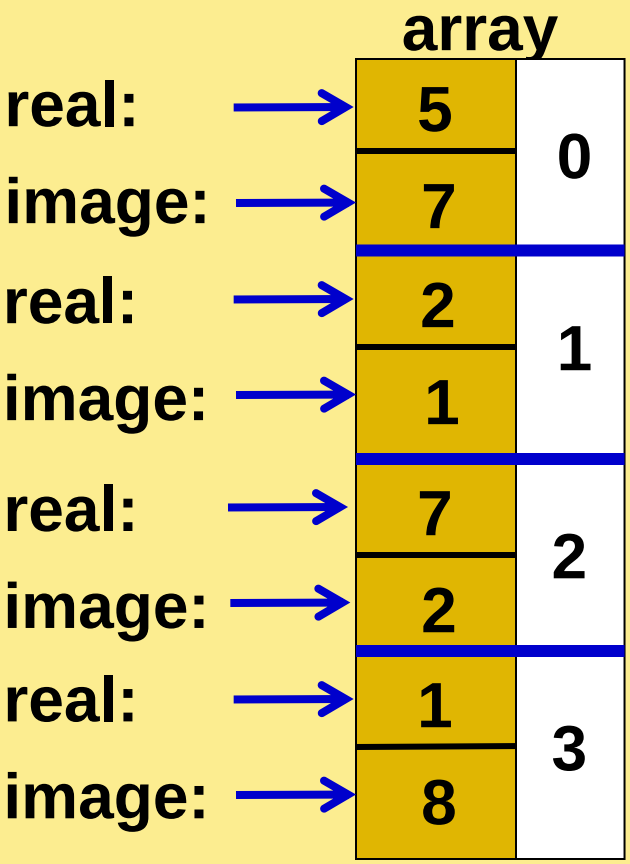
כל מלבן מכיל שני שדות מטיפוס point



חזרה - מערכים ומבנים

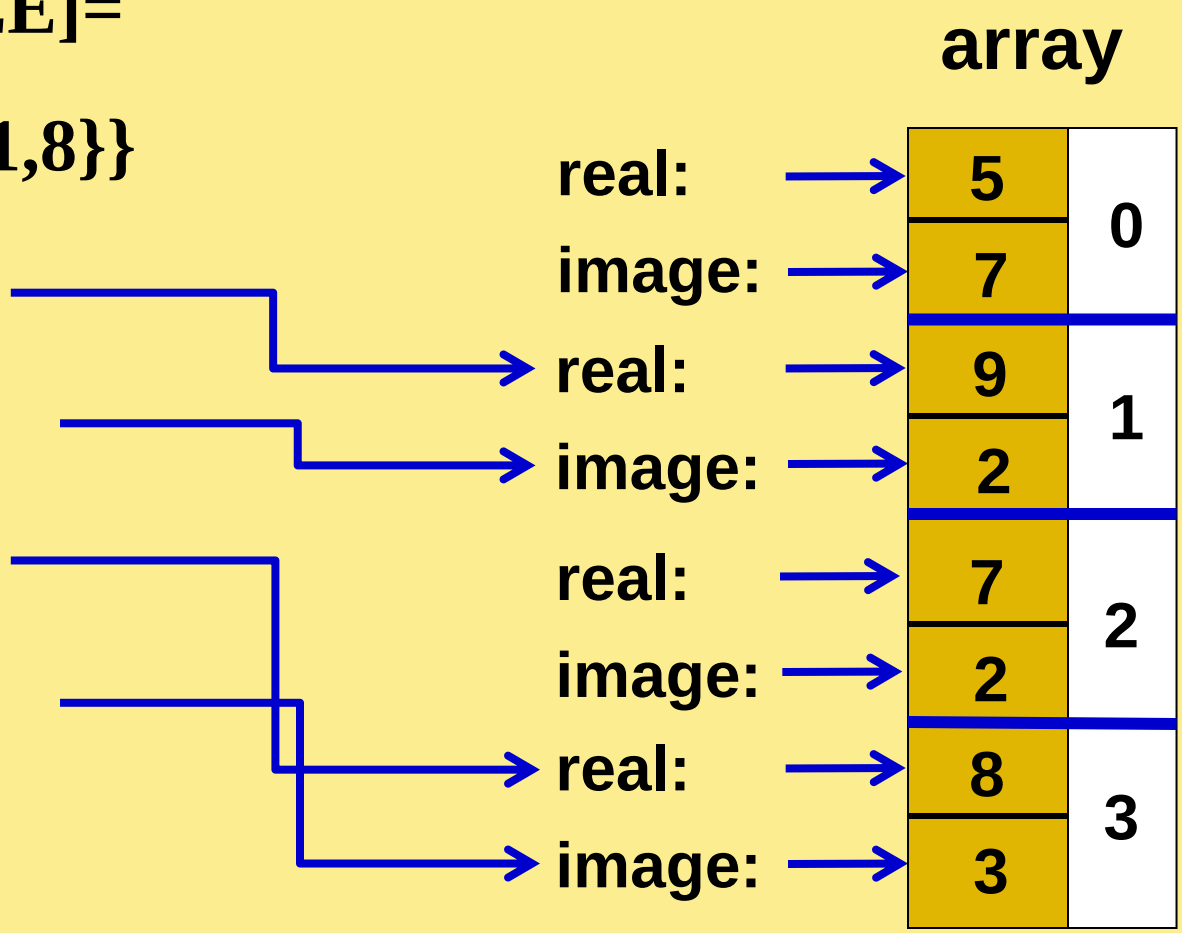
- מאחר ומבנים מגדירים סוג חדש של משתנים הרי שכמו לסוגים רגילים ניתן ליצור מערכים עבור סוגים אלו.

```
complex_t arr[SIZE]=
{{5,7},{2,1},{7,2},{1,8}}
arr[1].real = 2;
arr[1].image = 9;
arr[3].real = 3;
arr[3].image = 8;
```



חזרה -מערכים ומבנים אחרי ביצוע הפקודות


```
complex_t arr[SIZE]=
{{5,7},{2,1},{7,2},{1,8}}
arr[1].real = 9;
arr[1].image = 2;
arr[3].real = 8;
arr[3].image = 3;
```



שאלות על השעורים הקודמים?

רקורסיה

- רֶקוּרְסִיָּה (בעברית: נסיגה) היא תופעה שכל מופע שלה מכיל מופע נוסף שלה, כך שהיא מתרחשת ומשתקפת בשלמותה בתוך עצמה שוב ושוב.
- רקורסיה יכולה להיות <u>רקורסיית עצירה</u> (או רקורסיית קצה), כאשר יש בה "סף עצירה" – רמה שמתחתיה לא מתקיימת עוד הרקורסיה, או <u>רקורסיה אינסופית</u> כאשר בכל רמה תכיל התופעה תופעות משנה מאותו סוג.

רקורסיה

- רקורסיה הדדית מתרחשת בין שתי תופעות או יותר, כאשר האחת מכילה את השנייה וחוזר חלילה: א' מכיל מופע של ב', וב' מכיל מופע של א' (אם נביט על א' וב' כאחד – הם מקיימים רקורסיה רגילה), למשל כאשר מוצבת מראה מול מראה.
- אלגוריתם רקורסיבי הוא אלגוריתם אשר על מנת לפתור בעיה מסוימת, מפעיל את עצמו על מקרים פשוטים יותר של הבעיה (ובמקרים רבים: על תתי בעיות). בדרך כלל יכלול האלגוריתם "תנאי עצירה", שיביא להפסקת הרקורסיה ברמה שבה הפתרון נתון מראש, שאם לא כן תהיה זו לולאה אינסופית.

רקורסיה

- מבחינה טכנית, הרקורסיה ממומשת בדרך כלל על ידי פונקציה שקוראת לעצמה (או לפונקציה שקוראת לה, במקרה של רקורסיה הדדית). בזמן הריצה, בכל קריאה כזאת נשמרת כתובת החזרה ואוסף המשתנים המקומיים של הפונקציה במחסנית הקריאות שבזכרון.
- בסוף ביצוע כל פונקציה, הערכים האלה נשלפים מהמחסנית כך שבסיום הרקורסיה, כל הזיכרון שנדרש לחישובה, מתפנה.
- מאחר שכל קריאה רקורסיבית נשמרת במחסנית המחשב, בזמן החישוב, דורשת הרקורסיה זיכרון בגודל פרופורציונלי לעומקה.

רקורסיה

- ברקורסיבית פשוטות, אפשר להימנע מדרישת זיכרון זו על ידי תרגום הרקורסיה לאלגוריתם איטרטיבי שקול. ברקורסיבית מורכבות יותר, לא ניתן לעשות זאת ללא שימוש במחסנית ואז אין למעשה חיסכון בזיכרון.
- במקרים מסוימים, אלגוריתם רקורסיבי פשוט ובהיר יותר מאלגוריתם איטרטיבי שקול.
- אלגוריתמים רקורסיביים נתמכים במרבית שפות התכנות על ידי תמיכה של השפה בקריאה רקורסיבית של פונקציות.

רקורסיה

- לקריאות רקורסיביות (כמו לקריאות לפונקציות בכלל) ישנה עלות בזמן הריצה של התוכנית, ולכן מבצעות שפות תכנות רבות, ובפרט שפות פונקציונליות.
- אופטימיזציה של קריאות רקורסיביות (הנקראות "רקורסית זנב") והופכות קוד הנכתב באמצעות קריאות רקורסיביות לקוד לינארי המבוצע בפועל באמצעות לולאות.
- כל אלגוריתם רקורסיבי ניתן למימוש בצורה איטרטיבית.
- לעיתים הגרסה הרקורסיבית יעילה (מבחינת זמן ריצה) מזו האיטרטיבית אך לא תמיד.

רקורסיה

- בהרבה מקרים ההגדרה הרקורסיבית קצרה מזאת האיטרטיבית, והיא ההגדרה הטבעית למה שרוצים לחשב
- לפונקציה רקורסיבית חובה להגדיר תנאי עצירה חד-חד ערכי.
- ללא הגדרת תנאי עצירה חד-חד ערכי הפונקציה הרקורסיבית הופכת לאינסופית.
- בד"כ נעביר לפונקציה רקורסיבית פרמטר שיקטן בכל קריאה רקורסיבית עד לסיומה (הקטנת מימד הבעיה).
- פרמטר זה הוא המגביל את מספר הקריאות לרקורסיה. כאשר הפרמטר יהיה קטן מספיק (מקרה הבסיס), יתקיים תנאי העצירה ותסתיים הרקורסיה (ולא תבוצע קריאה רקורסיבית נוספת)

רקורסיה

- הגדרת רקורסיה במתמטיקה:

סדרה שמוגדרת באופן רקורסיבי באמצעות נוסחת נסיגה זאת סדרה שבה כל איבר מוגדר ע"י האיברים הקודמים בסדרה.

- סדרה חשבונית: $a_1=3$, $a_n=a_{n-1}+d$
- סדרה הנדסית: $a1=4$, $a_n=a_{n-1}*q$
- פונקציה רקורסיבית:

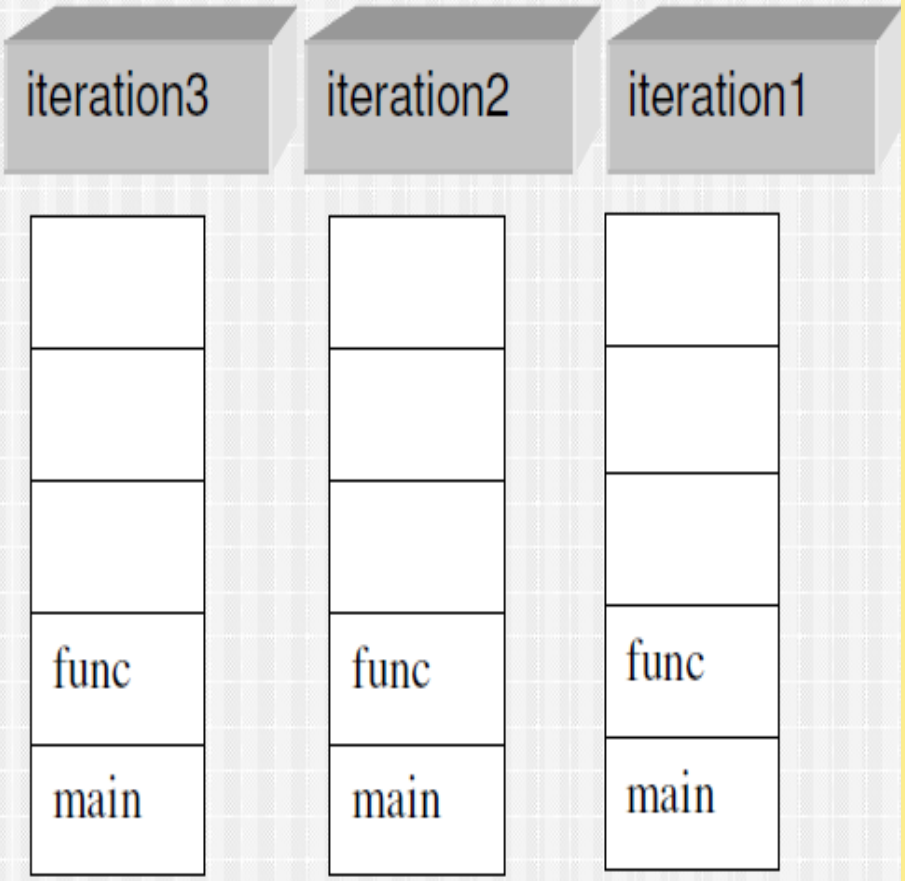
פונקציה הקוראת לעצמה באופן ישיר או עקיף דרך פונקציית עזר, מחייבת תנאי עצירה וסגירה חד ערכי.

שאלות?

רקורסיה מול איטרציה

דוגמא של איטרציה (לולאה):

```
void func()
{
    puts("test");
}

void main()
{
    for(int i=0; i<3; i++)
        func();
}
```



דוגמת רקורסיה מול איטרציה (2)

- במקום לקרוא לפונקציה 3 פעמים הלולאה הפונקציה יכולה לקרוא לעצמה עד שנגיע לכלל הקצה (בסיס הרקורסיה).

```
void func(int i)
{
    if(i>0)
   {
     puts("test");
     func(i-1);
  }
}
void main()
{
     func(3);
}
```

Stopping condition

Recursive call

Return iteration1	Return iteration2	Return iteration3	Call 4	Call 3	Call 2	Call 1
			func(0)			
		func(1)	func(1)	func(1)		
	func(2)	func(2)	func(2)	func(2)	func(2)	
func(3)	func(3)	func(3)	func(3)	func(3)	func(3)	func(3)
main	main	main	main	main	main	main

נקודות חשובות לגבי רקורסיה!

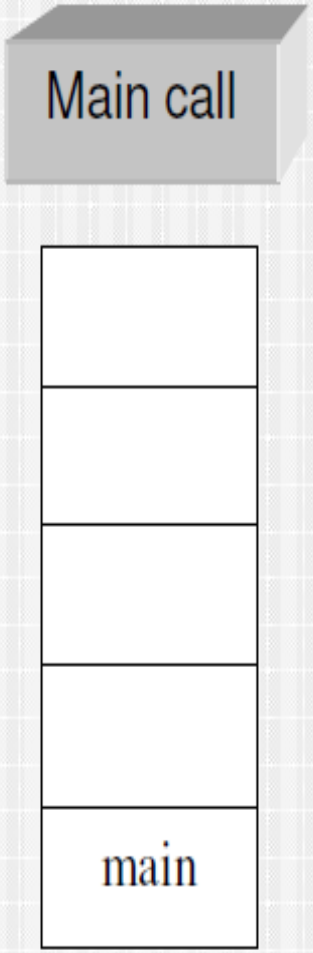
- שימו לב כי רקורסיה ללא כלל עצירה היא רקורסיה אינסופית!
- פונקציה רקורסיבית נשארת במחסנית עד שכל הקריאות הרקורסיביות שנוצרו כתוצאה ממנה מסיימות!

פקודות לאחר הקריאה הרקורסיבית

- לפונקציה רקורסיבית יש שני שלבים:
  1. לפני הקריאה הרקורסיבית: הפונקציה מוכנסת בצורה רקורסיבית למחסנית (הראשון נכנס ראשון).
  2. אחרי הקריאה הרקורסיבית: הפונקציה נמחקת בצורה רקורסיבית מהמחסנית (הראשון נמחק אחרון).
- ניתן לבצע פעולות לפני הקריאה הרקורסיבית וגם לאחריה.
  - פעולות לפני הקריאה – יתבצעו לפי סדר הקריאה.
  - פעולות לאחר הקריאה – יבוצעו רק לאחר החזרה מהקריאה הרקורסיבית (ראשון יבוצע אחרון).
- ושוב – חובה שיהיה תנאי עצירה למנוע לולאה אינסופית של הכנסות רקורסיביות!

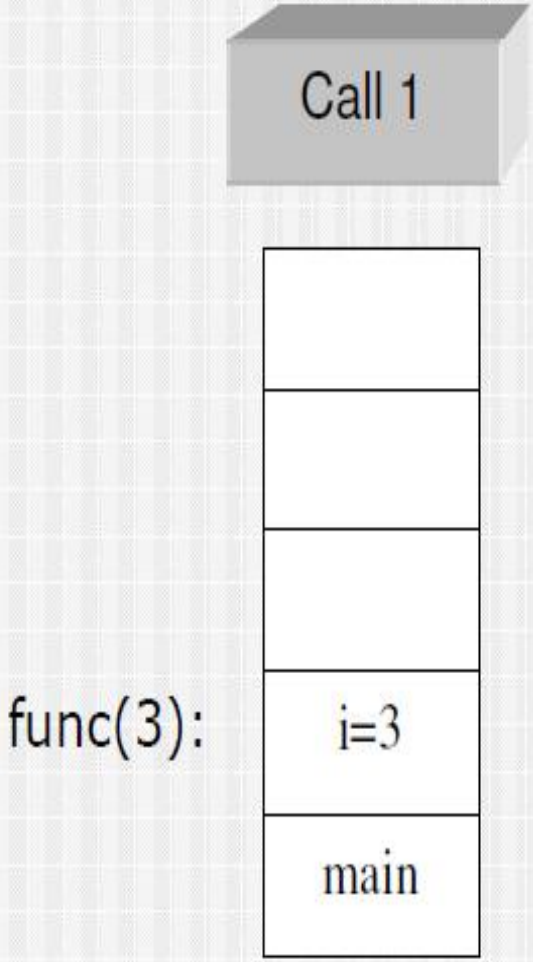
דוג' לפעולות לאחר הקריאה הרקורסיבית


```
void func(int i)
{
   if(i>0)
  {
    func(i-1);
    printf("%d", i);
 }
}
void main()
{
    func(3);
}
```



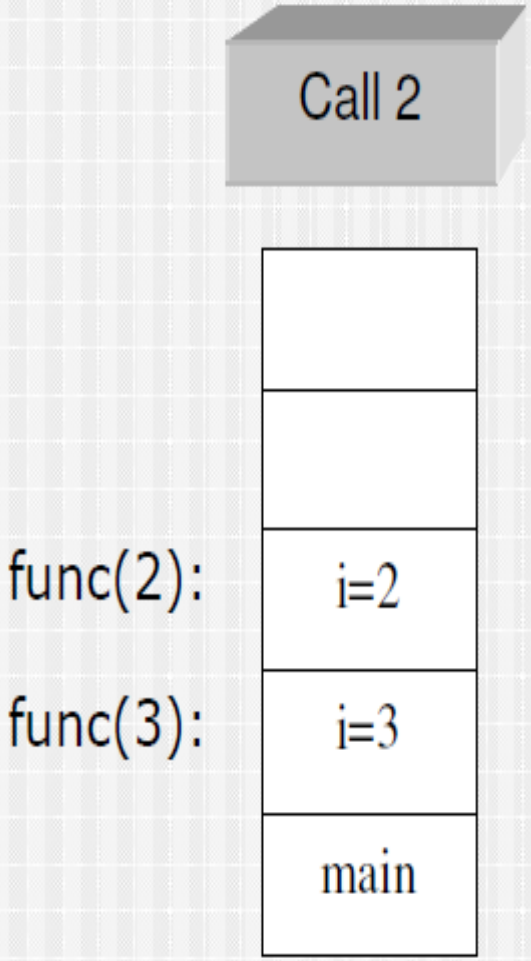
דוג' לפעולות לאחר הקריאה הרקורסיבית


```
void func(int i)
{
   if(i>0)
  {
    func(i-1);
    printf("%d", i);
  }
}
void main()
{
    func(3);
}
```



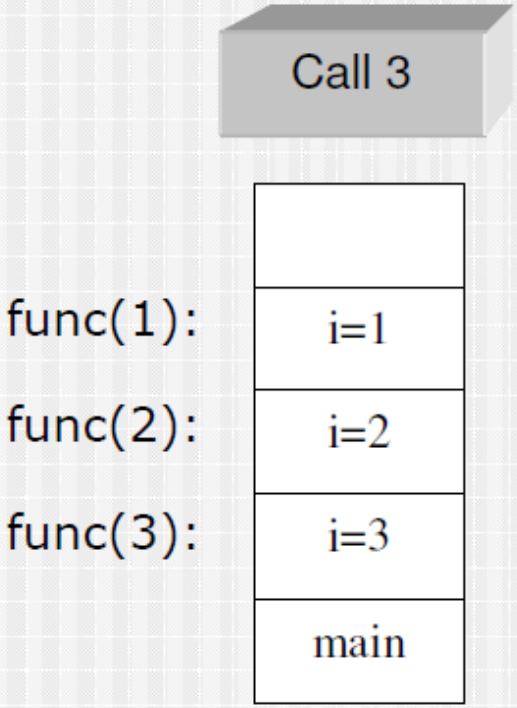
דוג' לפעולות לאחר הקריאה הרקורסיבית


```
void func(int i)
{
   if(i>0)
  {
    func(i-1);
    printf("%d", i);
  }
}
void main()
{
    func(3);
}
```



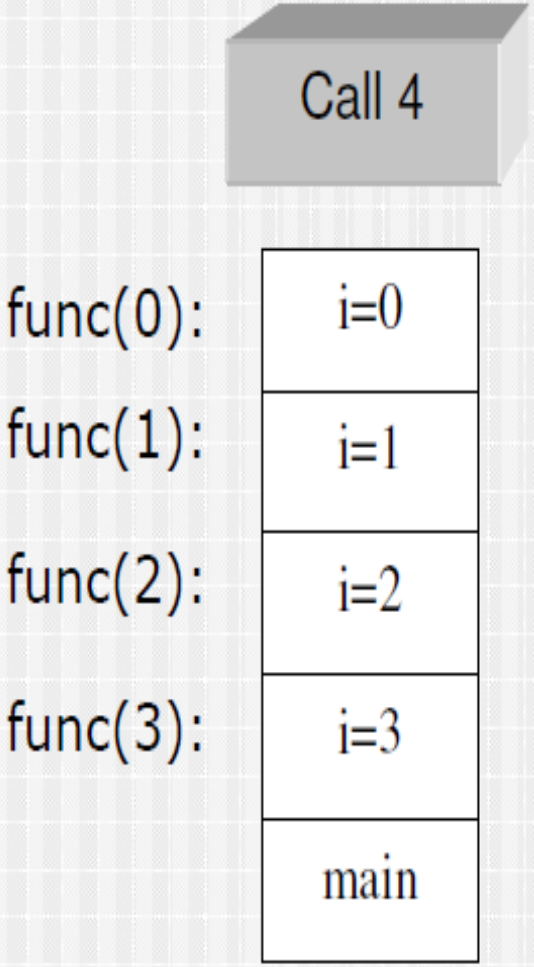
דוג' לפעולות לאחר הקריאה הרקורסיבית


```
void func(int i)
{
   if(i>0)
  {
    func(i-1);
    printf("%d", i);
  }
}
void main()
{
    func(3);
}
```



דוג' לפעולות לאחר הקריאה הרקורסיבית


```
void func(int i)
{
   if(i>0)
  {
    func(i-1);
    printf("%d", i);
  }
}
void main()
{
    func(3);
}
```



דוג' לפעולות לאחר הקריאה הרקורסיבית


```
void func(int i)
{
    if(i>0)
   {
     func(i-1);
     printf("%d", i);
  }
}
void main()
{
     func(3);
}
```

Back to 3

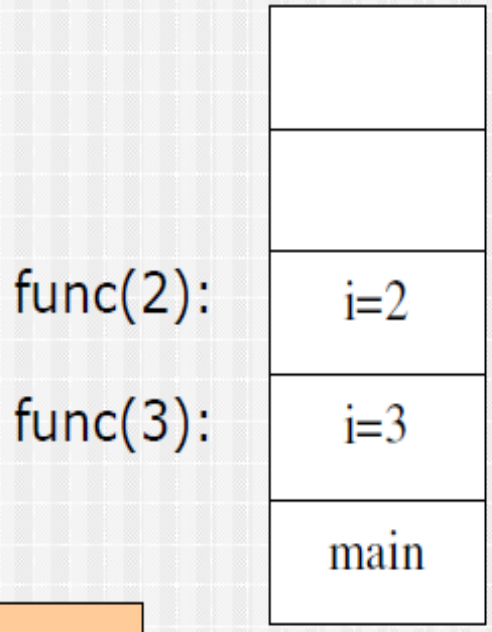
func(1):	i=1
func(2):	i=2
func(3):	i=3
	main

Output: 1

דוג' לפעולות לאחר הקריאה הרקורסיבית


```
void func(int i)
{
    if(i>0)
   {
     func(i-1);
     printf("%d", i);
  }
}
void main()
{
     func(3);
}
```

Back to 2



Output: 1 2

דוג' לפעולות לאחר הקריאה הרקורסיבית


```
void func(int i)
{
    if(i>0)
   {
     func(i-1);
     printf("%d", i);
  }
}
void main()
{
     func(3);
}
```

Back to 1

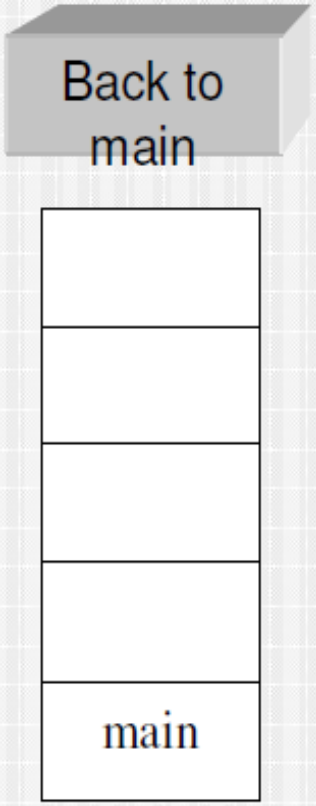
func(3):

i=3
main

Output: 1 2 3

דוג' לפעולות לאחר הקריאה הרקורסיבית


```
void func(int i)
{
    if(i>0)
   {
     func(i-1);
     printf("%d", i);
  }
}
void main()
{
     func(3);
}
```



Output: 1 2 3

חישוב עצרת רגיל ע"י רקורסיה

n ! == n * (n-1) !

```
int factorial(int n)
{
  if (n==0)
        return 1;
return n*factorial(n-1);
}
```

קריאה רקורסיבית לפונקציה עד אשר
ערכו של המשתנה n הופך ל- 0

n ! == n* (n-1) * … * 1

```
int factorial(int n)
{
      int fact =1;
      while (n >= 1)
      {
         fact *=n;
         n--;
       }
return fact;
}
```

חישוב רגיל של עצרת מהמספר

נקודות לתשומת-לב: עצירת הפונקציה

- כדי שהתוכנית תסתיים, יש לוודא שבאיזשהו שלב תנאי העצירה יתקיים, ללא קיומו הרקורסיה תימשך לנצח.
- תנאי עצירה זה הוא זהה לתנאי הסיום של לולאה מכל סוג.
- במידה והיינו קוראים לפונקציה עם אותו פרמטר שהועבר אליה (לדוגמא אם היינו כותבים factorial(n) ולא factorial(n-1)), אז לא היינו מתקדמים אל תנאי העצירה, והפונקציה הייתה חוזרת על עצמה עד אינסוף.
- בדוגמה הבאה מניחים כי n הוא 5.

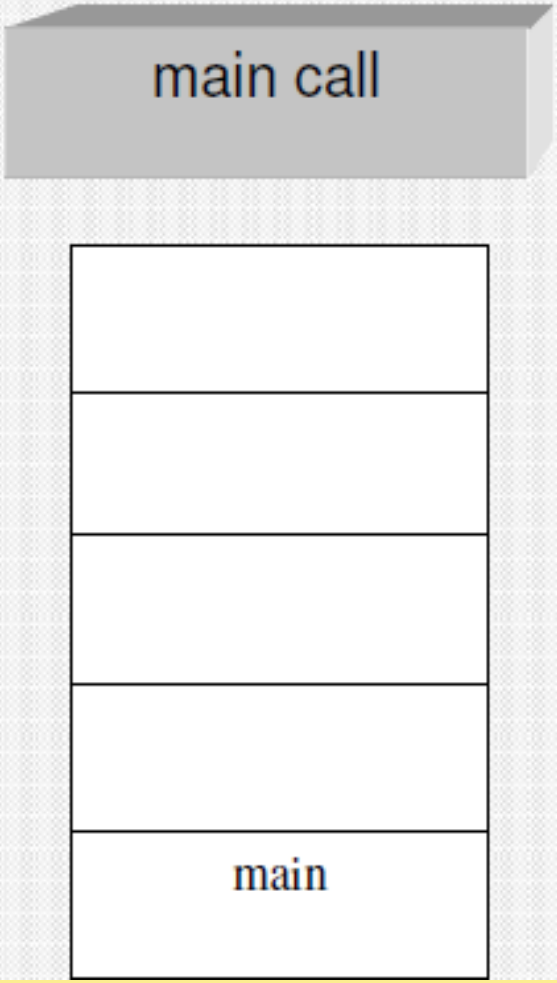
דוגמאות נוספות לפונקציות C רקורסיביות


```
unsigned factorial(unsigned n)
{
    if(n <= 1)
        return 1;
    return (n * factorial(n-1));
}
```

```
factorial(5)
5 * factorial(4)
5 * (4 * factorial(3))
5 * (4 * (3 * factorial(2)))
5 * (4 * (3 * (2 * factorial(1))))
5 * (4 * (3 * (2 * (1 * factorial(0)))))
5 * (4 * (3 * (2 * (1 * 1))))
5 * (4 * (3 * (2 * 1)))
5 * (4 * (3 * 2))
5 * (4 * 6)
5 * 24
120
```

פתרון רקורסיבי לעצרת - מעקב


```
int factorial(int n)
{
   if(n==0)    //recursion base
        return 1;
   else
        return n * factorial(n-1);
}

void main()
{
  factorial(3);
}
```



פתרון רקורסיבי לעצרת - מעקב


```
int factorial(int n)
{
   if(n==0)    //recursion base
        return 1;
   else
        return n * factorial(n-1);
}

void main()
{
  factorial(3);
}
```

Call 1

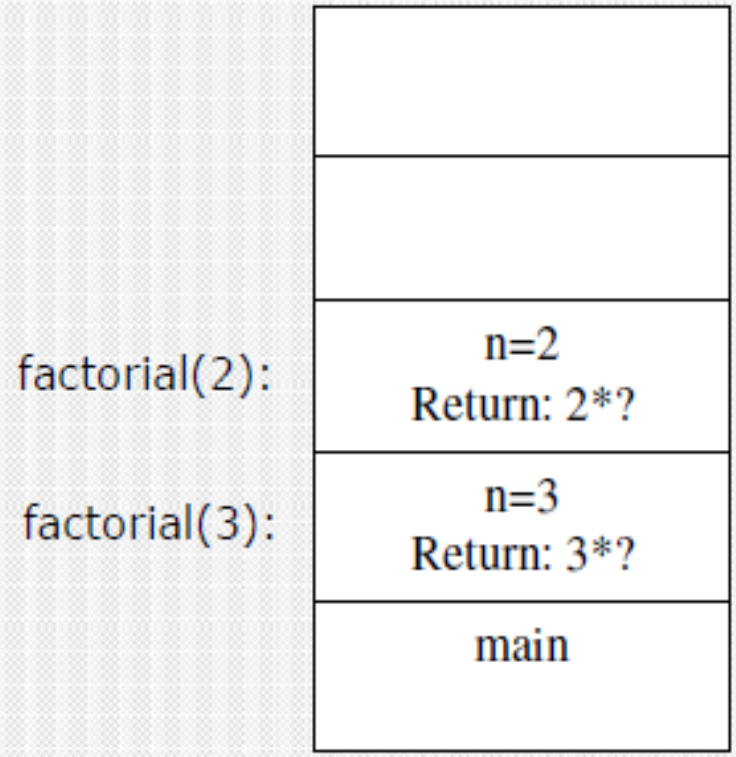
factorial(3):

n=3 Return: 3*?
main

פתרון רקורסיבי לעצרת - מעקב


```
int factorial(int n)
{
   if(n==0)    //recursion base
        return 1;
   else
        return n * factorial(n-1);
}

void main()
{
  factorial(3);
}
```

Call 2



פתרון רקורסיבי לעצרת - מעקב


```
int factorial(int n)
{
   if(n==0)    //recursion base
        return 1;
   else
        return n * factorial(n-1);
}

void main()
{
  factorial(3);
}
```

Call 3

factorial(1):	n=1 Return: 1*?
factorial(2):	n=2 Return: 2*?
factorial(3):	n=3 Return: 3*?
	main

פתרון רקורסיבי לעצרת - מעקב

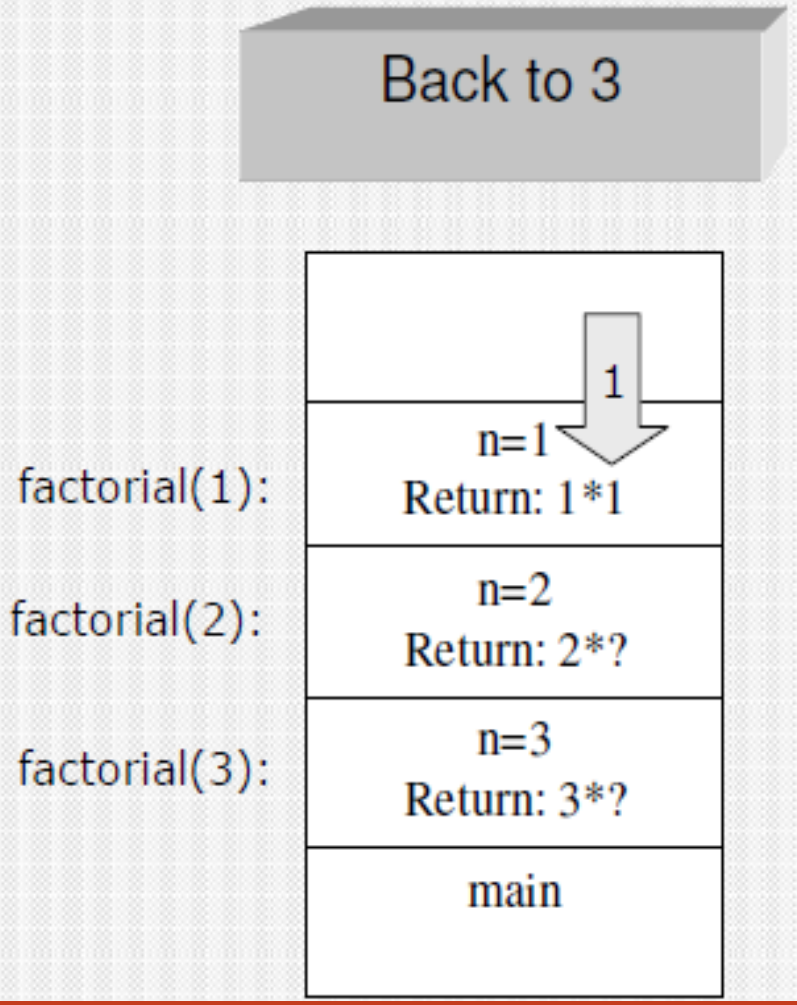

```
int factorial(int n)
{
   if(n==0)    //recursion base
        return 1;
   else
        return n * factorial(n-1);
}

void main()
{
  factorial(3);
}
```

Call 4

factorial(0):	n=0 Return: 1
factorial(1):	n=1 Return: 1*?
factorial(2):	n=2 Return: 2*?
factorial(3):	n=3 Return: 3*?
	main

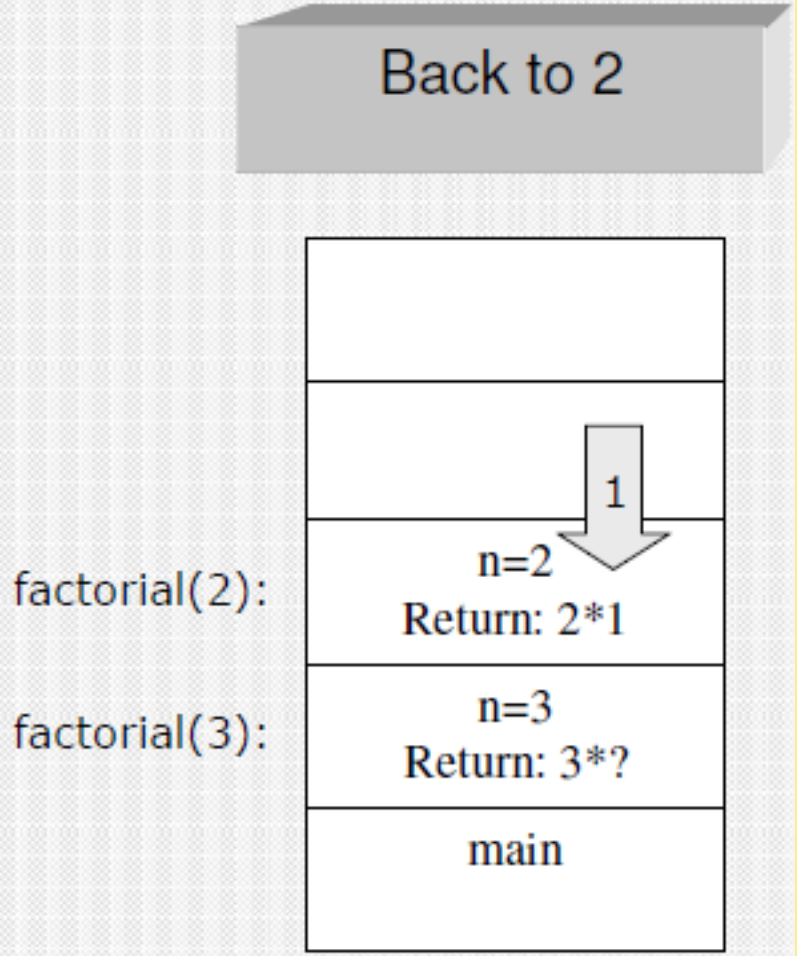
פתרון רקורסיבי לעצרת - מעקב


```
int factorial(int n)
{
   if(n==0)    //recursion base
        return 1;
   else
        return n * factorial(n-1);
}

void main()
{
  factorial(3);
}
```



פתרון רקורסיבי לעצרת - מעקב


```
int factorial(int n)
{
   if(n==0)    //recursion base
        return 1;
   else
        return n * factorial(n-1);
}

void main()
{
  factorial(3);
}
```

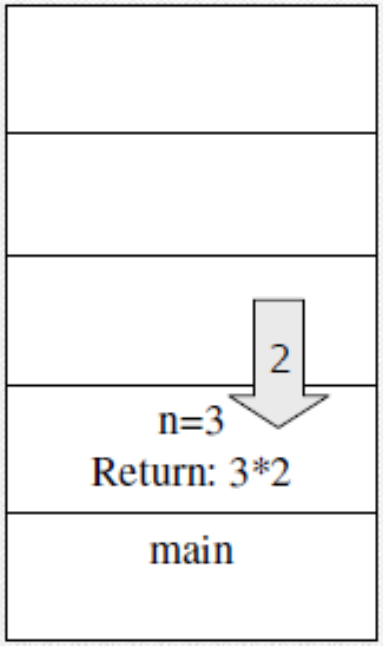


פתרון רקורסיבי לעצרת - מעקב


```
int factorial(int n)
{
   if(n==0)    //recursion base
        return 1;
   else
        return n * factorial(n-1);
}

void main()
{
  factorial(3);
}
```

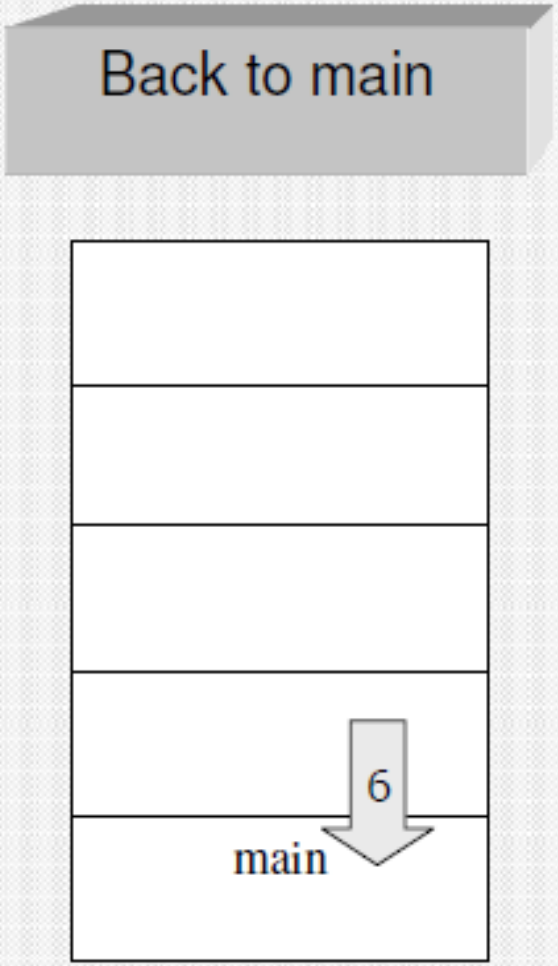
Back to 1

factorial(3):



פתרון רקורסיבי לעצרת - מעקב


```
int factorial(int n)
{
   if(n==0)    //recursion base
        return 1;
   else
        return n * factorial(n-1);
}

void main()
{
  factorial(3);
}
```



תרגול רקורסיה – סכום ספרות של מספר

- כתבו פונקציה המחשבת את סכום הספרות של מספר.

```
int digsum(int x)
{
  int res;
  if (x==0)
  {
     res=0;
     return res;
   }
  else
       res=x%10+digsum(x/10);
return res;}
```

חישוב חזקה באופן רקורסיבי

- ההגדרה הרקורסיבית (למעריך שלם ואי-שלילי):

$a^n = a \cdot a^{n-1}, \quad a^0=1$

- והפונקציה ב- C:

```
double power(double base, double exp)
{
    if (exp<1)
        return 1;
    return base * power(base, exp-1);
}
```

- אפשר לטפל במעריך שלילי כמו בחיובי

חישוב חזקה באופן רקורסיבי

- אם יתכן מעריך שלילי, ההגדרה היא:

עבור $n<0$: $a^{-n} = 1/a^n$

עבור $n\geq 0$: $a^n = a \cdot a^{n-1}, \quad a^0=1$

- הפונקציה ב- C:

```
double power(double base, int exponent)
{
    if (exponent < 0)  return 1/power(base, -exponent);
    if (exponent==0)  return 1;
    return base * power(base, exponent-1);
}
```

תרגול רקורסיה – חישוב חזקה (שלמים)

- כתבו פונקציה המחשבת חזקה של מספר שלם.
- פתרון:

```
int power(int base, int exponent)
{
        if (exponent == 0)
            return 1;
        return base * power(base, exponent -1);
}
```

תרגול רקורסיה

תרגיל :

א. כתבו פונקציה רקורסיבית המחשבת את המכפלה של שני איברים באמצעות חיבור בלבד.

ב. לאחר מכן כתבו פונקציה רקורסיבית נוספת המחשבת את החלוקה של שני מספרים באמצעות פעולות חיסור בלבד.

ג. כתבו תוכנית הקולטת שני מספרים ומבצעת את שתי הפונקציות, של הכפל והחלוקה לשני המספרים ומדפיסה את תוצאת התרגילים.

תרגול רקורסיה – כפל בין מספרים

- פתרון:

```
#include <stdio.h>
int mul(int mis1, int mis2)
{
        if (mis2==1)
                return mis1;
        return mis1 + mul(mis1,mis2-1);
}
```

תרגול רקורסיה – חילוק בין מספרים שלמים

- פתרון:

```
int div(int mis1, int mis2)
{
        if (mis2>mis1)
                return 0;
        return 1 + div(mis1 - mis2 , mis2);
}
```

תרגול רקורסיה – חילוק בין מספרים שלמים

- תוכנית ראשית:

```
int main()
{
    int num1,num2;
    printf("Enter two nums for div\n");
    scanf("%d%d",&num1,&num2);
    printf("Div num1/num2: %d\n",div(num1,num2));
    printf("\n\nEnter two nums for mull\n");
    scanf("%d%d",&num1,&num2);
    printf("Mul num1*num2: %d\n",mul(num1,num2));
    return 0;
}
```

תרגול רקורסיה

- כתוב תוכנית הכוללת פונקציה רקורסיבית הקולטת רצף תווים עד ללחיצה על מקש האנטר ומדפיסה אותו.

```
#include <stdio.h>
void wrt_it();
int main()
{
        printf("Input a line\n");
        wrt_it();
        printf("\n");
        return 0;
}
```

תרגול רקורסיה

- פתרון הפונקציה הרקורסיבית:

```
void wrt_it()
{
        char ch;
        scanf("%c", &ch);
        printf("%c", ch);
        if (ch != '\n')
            wrt_it();
}
```

תרגול רקורסיה

- תרגיל המשך - הפעם הדפס את הקלט מהסוף להתחלה
- פתרון התוכנית הראשית:

```
#include <stdio.h>
void wrt_it();
int main()
{
        printf("Input a line\n");
        wrt_it();
        printf("\n");
        return 0;
}
```

תרגול רקורסיה

- פתרון הפונקציה הרקורסיבית:

```
void wrt_it()
{
        char ch;
        scanf("%c", &ch);
        if (ch != '\n')
            wrt_it();
        printf("%c",ch);
}
```

שאלות?

רקע לרקורסיה: הגדרת נוסחאות

- כידוע, דרך מקובלת להגדיר נוסחה או פעולה מתמטית היא פשוט לרשום את שלבי החישוב שלה.
- דוגמאות: (חישוב עצרת של מספר בשלבים)

$$n!=1*2*3*\ldots*n$$

$$a^n=\underbrace{a*a*\ldots*a}_{n \text{ פעמים}}$$

- מקובל גם לחשב את ערך הנוסחה שלב-אחר-שלב.

למשל ניתן לכפול במספר העוקב בכל שלב:

$$4!=1*2*3*4=2*3*4=6*4=24$$

רקע רקורסיה: הגדרת נוסחאות

- דרך נוספת להגדיר נוסחאות מאפשרת לרשום את כל שלבי החישוב, אפשר להגדיר את הערך של השלב האחרון בעזרת תוצאות השלבים שלפניו.
- למשל, במקום להגדיר עצרת על-ידי:

$$n!=1*2*3*\ldots*n$$

- אפשר להגדיר על-ידי:

$$n!=n*(n-1)!$$

- בצירוף ההגדרה ההתחלתית: $1!=1$.
  - ואז החישוב יהיה:

$$4!=4*3!=4*3*2!=4*3*2*1!=4*3*2*1=4*3*2=4*6=24$$

רקע רקורסיה: הגדרת נוסחאות

- לפי השלבים הקודמים ההגדרה נקראת "הגדרה רקורסיבית", או "נוסחת רקורסיה" של התהליך אותו מבצעים.
- יש צורך לציין את התוצאה עבור ערך התחלתי כלשהו ("בסיס הרקורסיה"), כי אחרת חישוב הנוסחה בעצם לא מסתיים.
- כל חישוב רקורסיבי ניתן להגדיר ולחשב גם לפי פירוט השלבים ("הגדרה איטרטיבית").

רקע רקורסיה: הגדרת נוסחאות

- דוגמא נוספת:

במקום: $a^n=a*a*.....*a$, אפשר להגדיר חזקה על-ידי:

$a^n = a*a^{n-1}, \quad a^0=1$

ואז: $4^2=4*4^1=4*4*4^0=4*4*1=16*1=16$

זו הגדרה שקולה של נוסחאות.

מדוע נחוצה לנו הגדרה זו?

- במקרים רבים, ההגדרה הרקורסיבית קצרה בהרבה, מבחינת טקסט שנכתב ואח"כ מקודד למחשב מזו האיטרטיבית.
- במקרים מסוימים, ההגדרה הרקורסיבית היא ההגדרה הטבעית והנוחה ביותר של מה שרוצים לחשב.
- יש לזכור שלכל הגדרה רקורסיבית יש הגדרה איטרטיבית שקולה, אמנם יתכן כי היא ארוכה יותר, מורכבת וכוללת מספר משתנים נוספים אך בכל מקרה היא קיימת.

מדוע נחוצה לנו הגדרה זו?

- יש לזכור כי הגדרה רקורסיבית מתבצעת פעם אחת אולם התהליך הממוחשב מתחיל משלב ראשון וחוזר באופן רקורסיבי עד להגעה לתנאי הסיום של הרקורסיה.
- חובה להגדיר תנאי עצירה כי אחרת הפונקציה הרקורסיבית הופכת לאינסופית.
- בדרך כלל המשתנה המוגדר בתנאי העצירה מועבר מהתוכנית הראשית אך לעיתים הוא נקלט ישירות לפונקציה.

דוגמא להגדרה רקורסיבית

- בסדרת פיבונצ'י כל איבר מוגדר כסכום של שני האיברים שלפניו.
- שני האיברים הראשונים הם תמיד 1:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,….

- ההגדרה הטבעית עבור האיבר ה- n בסדרה הזאת היא:

Fib(n)=Fib(n-1)+Fib(n-2)

- כאשר:

Fib(1)=1, Fib(2)=1

דוגמא להגדרה רקורסיבית

- ההגדרה האיטרטיבית, במקרה זה אינה נוחה (כלומר נוסחה מפורשת של Fib(n)).
- הרבה יותר נוח וקל לבצע הגדרה רקורסיבית לנוסחת פיבונצ'י.
- החישוב:

Fib(4)=Fib(3)+Fib(2)=Fib(2)+Fib(1)+1=1+1+1=3

דוגמא להגדרה רקורסיבית

- זהו חישוב מסובך ומורכב הבא למצוא את מספרו של האיבר הרצוי בסדרת פיבונצ'י לפי מספרו הסידורי:
- מספר בסדרה:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…

- מספר סידורי:

1, 2, 3, 4, 5, 6, 7 , 8 , 9 , 10 , 11…

תרגול רקורסיה – פיבונצ'י

- פונקציה אטרטיבית למציאת מספר פיבונצ'י

```
int fib_iter(int number)
{
        int  prev1 = 0, prev2 = 1, current, count;
        for (count=0; count<number; count++ )
        {
                current = prev1 + prev2;
                prev1 = prev2;
                prev2 = current;
        }
        return current;
}
```

תרגול רקורסיה – סדרת פיבונצ'י

- פונקציה רקורסיבית למציאת מספר פיבונצ'י

```
int fib_rec(int number)
{
        if (number<= 1)
            return number;
        else
            return fib_rec(number-1)+fib_rec(number-2);
}
```

שאלות?

נקודות לתשומת-לב: משתנים

- כשקוראים לפונקציה מתוך עצמה, משתנים שהוגדרו בפונקציה הקוראת נשארים בזיכרון.
- המשתנים נשארים בזיכרון עד שהפונקציה מסתיימת.
- אם נבצע קריאות רקורסיביות רבות, זה עלול למלא את הזיכרון של המחשב, ולגרום לעומס ולעיתים לתקיעה.
- יש לציין שעבור כל קריאה לפונקציה נידרש מקום חדש המוקצה ע"י המחשב בזיכרון גם עבור הנקודה שאליה היא צריכה לחזור וגם עבור הערך שהיא צריכה להחזיר, כך שנידרש מקום רב גם אם נחסוך במשתנים.

נקודות לתשומת-לב: בהירות

- מצד אחד ישנה טענה כי לשימוש ברקורסיה יש יתרון במקרים מסוימים שבהם קל יותר לכתוב באמצעותה את החישוב.
- מצד שני, לא תמיד קל למצוא הגדרה רקורסיבית לפונקציה, ולא תמיד קל להבין תוכניות שנכתבו באופן רקורסיבי.
- לכן נבחר להשתמש ברקורסיה במקרים שבאמת נוחים לכך.

נקודות לתשומת-לב: יעילות

- במקרה שראינו, מספר הקריאות לפונקציה היה זהה למספר השלבים שהיינו עושים בחישוב על-ידי לולאה. יתכנו מקרים שבהם נזדקק לקריאות רבות בהרבה, ואז זמן הריצה עלול להיות הרבה יותר איטי.
- שימוש במשתנה סטטי נפוץ מאד ברקורסיה במידה ורוצים להכניס לפונקציה רקורסיבית משתנה סוכם יש להגדירו כך:

```
static int sum=0;
```

הגדרה זו תגרום למשתנה לא להתאפס כל קריאה רקורסיבית ולאפשר לנו לסכום בתוכו תוצאות.

סדרת פיבונצ'י – חישוב רקורסיבי

- להלן סדרה המדגימה נוסחה המוגדרת באופן רקורסיבי:

fib(n)=fib(n-1)+fib(n-2), fib(1)=1, fib(2)=1

- נוכל לכתוב את זה כך ב- C:

```
int fib (int n)
{
    if ((n==1) || (n==2))
        return 1;
    return fib(n-1)+fib(n-2);
}
```

- מה ההבדל בין הדוגמא הזו לשתי הדוגמאות הקודמות שראינו?

חישוב סדרת פיבונצ'י

- כל קריאה לפונקציה הזאת עם n>2, יוצרת שתי קריאות נוספות לפונקציה, אם גם בהן n>2, אז כל אחת מהן יוצרת שתי קריאות נוספות, וכן הלאה.
- עבור n קטן יחסית, למשל 50, נקבל מספר עצום של קריאות לפונקציה, שימלאו את הזיכרון במשתנים (ה-n של כל הקריאות לפונקציה).

```
int fib (int n)
{
   if ((n==1) || (n==2))
      return 1;
   return fib(n-1)+fib(n-2);
}
```

תיעוד קריאות לפונקציה fib

number	ערך	מספר קריאות
1	1	1
2	1	1
3	2	3
23	28657	92735
24	46368	150049

סדרת פיבונצ'י - יעילות

- כשנבחן את מדד היעילות נמצא כי צפויה לנו בעיה רצינית בפונקציה הרקורסיבית שתיארנו בגלל הבזבוז של מספר הקריאות.
- על מנת לחשב את האיבר ה-23 בסדרה צריך לחשב לכל היותר את 22 הערכים שלפניו, אבל בחישוב הרקורסיבי יתבצעו עשרות אלפי קריאות לפונקציה, כלומר יחושבו עשרות אלפי ערכים.
- המסקנה המתבקשת: לא תמיד הפונקציה הרקורסיבית יעילה יותר מהאיטראטיבית.

סדרת פיבונצ'י - יעילות

- כל קריאה לפונקציה פותחת איטרציה חדשה במחשב, כל איטרציה חדשה כזו מורכבת ממשאבי זיכרון ומאטה את מהירות המעבד.
- מספר רב של קריאות גורם לבזבוז משאבים הגורם לחוסר יעילות.
- המסקנה המתבקשת: יש לוודא היטב האם יש צורך בפונקציה רקורסיבית לפני שכותבים ומפעילים אותה.

סדרת פיבונצ'י - יעילות

- הסיבה לחוסר היעילות היא ששני הערכים שמחושבים בקריאה הרקורסיבית לא נשמרים, ולכן ברקורסיה יבוצעו הרבה פעמים את הקריאות fib(1), fib(2), וכו'.
- המסקנות המתבקשות מכך הן:

1. אם פונקציה רקורסיבית צריכה לקרוא לעצמה יותר מפעם אחת, אז חישובה הוא בעייתי, ויתאפשר רק למספרים קטנים.
2. עבור מספרים גדולים נצטרך להגדיר, לכתוב ולהפעיל פונקציה איטרטיבית.

תרגול רקורסיה – מספר מקסימלי במערך

- כתבו פונקציה רקורסיבית המוצאת את האיבר המקסימלי במערך

```
int max(int mis1, int mis2)
{
        if(mis1>mis2)
                return mis1;
        else
                return mis2;
}
int max_arr(int arr[], int number)
{
   if (number == 1)    return arr[0];
   return max(arr[0], max_arr(arr+1, number-1));
}
```

תרגול רקורסיה – סיכום איברי המערך

- רקורסיה יכולה לשמש אותנו לא רק לחישוב נוסחאות.
- אפשר להשתמש בה עבור כל בעיה שניתן לפתור על-ידי פתרון של מקרה יותר קטן/פשוט שלה.
- לדוגמא אפשר לחשב סכום איברי מערך לפי סכום האיברים בלי האיבר האחרון, ועוד האיבר האחרון: (עוצרים כשנשאר אחד)

```
int sum_arr(int arr[], int size)
{
  if (size==1)
      return arr[0];
  return arr[0] + sum_arr(arr+1,size-1);
}
```

תרגול רקורסיה – מציאת תו במחרוזת

- הפונקציה הבאה בודקת האם תו נמצא במחרוזת, ומחזירה 1 אם כן ואחרת 0.
- הפונקציה נעצרת כשמגיעים לסוף המחרוזת או כשמצאנו את התו המבוקש.

```
int in_str(char *str, char letter)
{
    if (*str == '\0') return 0;
    if (*str == letter) return 1;
    return in_str(str+1, letter);
}
```

רקורסיה – עוד דוגמא

- הפונקציה הבאה מחזירה את מספר ההופעות של תו במחרוזת.
- אם התו הראשון הוא התו שמחפשים, אז מספר ההופעות הוא 1 ועוד מספר הופעותיו במקומות הבאים. אחרת, זה רק מספר הופעותיו במקומות הבאים (עוצרים כשהגענו לסוף המחרוזת).

```
int in_str(char *str, char letter)
{
    if (*str == '\0') return 0;
    if (*str == letter) return 1 +
                                in_str(str+1,    letter);
    return in_str(str+1, letter);
}
```

שאלות?

רקורסיה - Recursion

- אלגוריתם רקורסיבי לפתרון בעיה הינו אלגוריתם הניתן לשימוש בפתרונות של אותה בעיה עבור פרמטרים קטנים יותר.
- מתי משתמשים ברקורסיה ?
  - כאשר נתונה בעיה עם פרמטר n.
  - אם יש בידינו פתרון לאותה בעיה עם פרמטר קטן יותר (לדוגמא n-1)
  - אם הפתרון הולך בקלות עבור n.
- כדי שתהליך החישוב יסתיים, מגדירים את בסיס הרקורסיה: זהו מקרה פשוט של הבעיה בו הפתרון ידוע ללא הפעלת הרקורסיה.

פתרון בעיות ברקורסיה - דוגמאות

- דוגמא 1:
  - הבעיה: חישוב $f(n) = n!$
  - נניח כי נתון: $f(n-1) = (n-1)!$
  - נציג את הפתרון: $f(n) = n * f(n-1)$
  - בסיס הרקורסיה: $f(0) = 1$

פתרון בעיות ברקורסיה - דוגמאות

- דוגמא 2:
  - הבעיה: חישוב gcd(m, n)
  - נציג את הפתרון: gcd(m, n) = gcd(n, m % n)
  - בסיס הרקורסיה : gcd(m, 0) = m

פתרון בעיות ברקורסיה – דוגמאות, המשך

- דוגמא 3:
  - הבעיה: $f(x, n) = x^n$
  - פתרון רקורסיבי: $f(x, n) = x * f(x, n-1)$
  - בסיס הרקורסיה: $f(x, 0) = 1$ לכל x

פתרון בעיות ברקורסיה – דוגמאות, המשך

- דוגמא 4:
  - הבעיה: יש לקבוע האם הערך x נמצא במערך בגודל n הממוין בסדר עולה.
  - נציג את הפתרון:
    - אם x שווה לאיבר האמצעי, אזי מצאנו את x.
    - אם x קטן מהאיבר האמצעי במערך, נחפש את x בתת המערך השמאלי שגודלו $\lfloor n/2 \rfloor$
    - אם x גדול מהאיבר האמצעי במערך, נחפש את x בתת המערך הימני שגודלו $\lceil n/2 \rceil$
  - בסיס הרקורסיה:
    - x איננו נמצא במערך ריק (גודל 0).

רקורסיה לעומת לולאות

- היתרון ברקורסיה:
  - פתרון מנוסח באופן אלגנטי וקריא, קל לכתיבה ולתחזוקה.
  - "הנהלת החשבונות" של התכנית (כגון הקצאת מערך ועדכון אינדקסים) נסתרת ושקופה הן למתכנת ובוודאי למפעיל.
  - בדוגמת היפוך הקלט יוקצה זיכרון בהתאם לאורך המחרוזת, ואין צורך להניח חסם עליון על מספר התווים.
- החיסרון במימוש הרקורסיבי: מספר רב של קריאות לפונקציה
  - יכול לגרום לצריכת זיכרון מופרזת בהשוואה לפתרון איטרטיבי (מבוסס על לולאות)
- לכל חישוב רקורסיבי ניתן לבנות חישוב איטרטיבי שקול וההפך
  - צריך לחשוב ולבחור את הפתרון המתאים ביותר מבחינת יעילות או נוחות

דוגמאות נוספות לפונקציות C רקורסיביות

gcd – מחלק משותף מקסימאלי של שני מספרים שלמים הוא המספר הגדול ביותר שמחלק את שניהם.

למשל, המחלק המשותף המקסימלי של 12 ו־18 הוא 6.

```
unsigned gcd(unsigned m, unsigned n)
{
    if(n == 0)
        return m;
    return gcd(n, m%n);
}
```

סיבוכיות זמן ומקום של הפונקציות הרקורסיביות

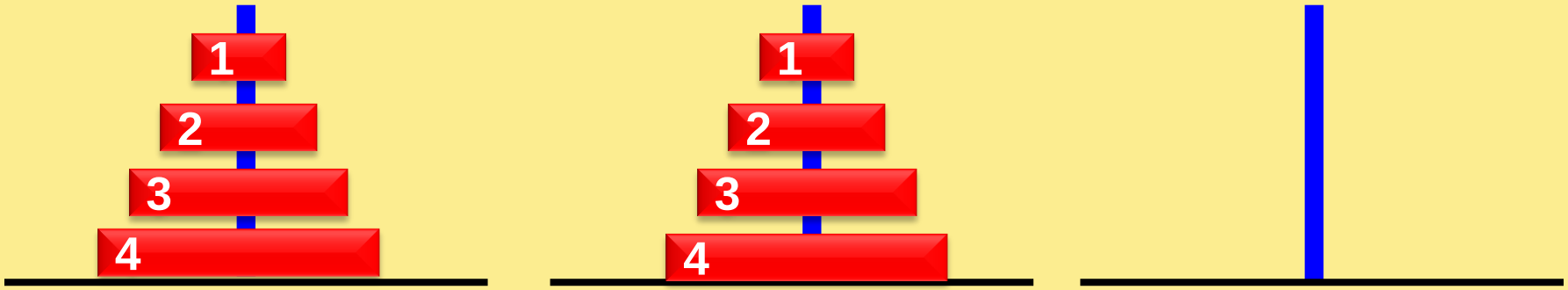
- <u>סיבוכיות הזמן</u> (מספר פעולות החישוב) ו<u>סיבוכיות המקום</u> (כמות הזיכרון) של `factorial(n)` ושל `power(x,n)` הן O(n) (גידול ליניארי עם n)
- סיבוכיות הזמן וסיבוכיות המקום של `exists(a,n,x)`
- הן O(log n) (גידול ליניארי עם log n)
- לכל אחת מהבעיות הללו, ניתן לכתוב קוד איטרטיבי פשוט
- בעל סיבוכיות זמן הזהה לזו של הקוד הרקורסיבי ובעל סיבוכיות מקום קטנה יותר - O(1) (גודל קבוע שאינו תלוי ב-n)

מימוש יעיל יותר של חזקה


```
double power(double x, unsigned n)
{
    if (n == 0) return 1;
    double y = power(x, n/2);
    y *= y;
    if (n % 2)
        y *= x;
    return y;
}
```

- סיבוכיות הזמן: $\log_2 n$
- סיבוכיות המקום: $\log_2 n$

פתרון בעיות ברקורסיה – מגדלי הנוי



A
מוט המקור

B
מוט המטרה

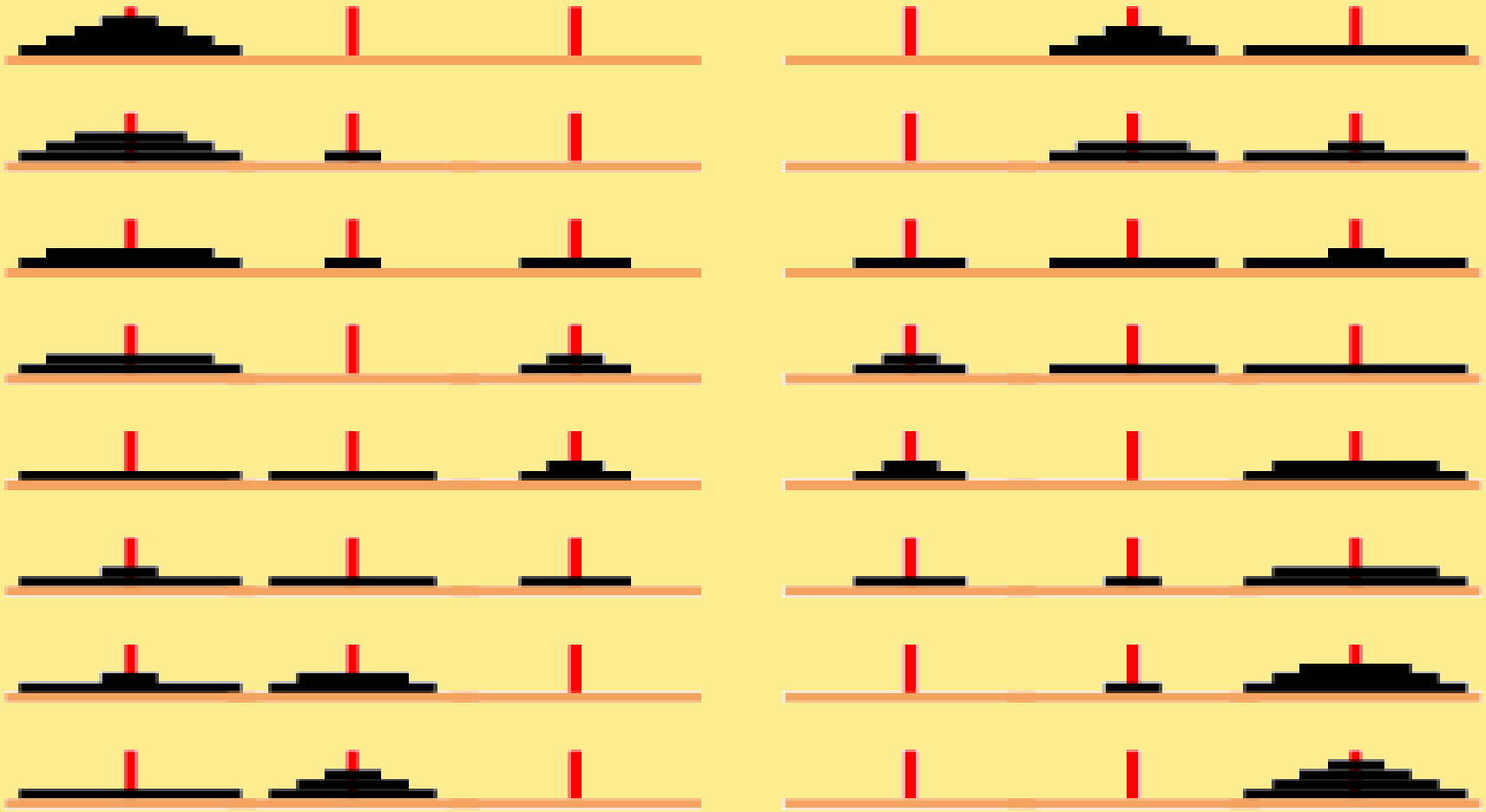
C
מוט העזר

- **נתון:**
- **המשימה: להעביר את כל הדיסקים (ארבעה במספר) ממוט A למוט B תוך שימוש במוט C כמוט עזר זמני בלבד, לפי הכללים הבאים:**
  - **בכל צעד מותר להעביר דיסק בודד ממוט למוט.**
  - **אסור מצב שבו דיסק גדול מונח על דיסק קטן באף אחד מהמוטות.**

פתרון בעיות ברקורסיה – מגדלי הנוי

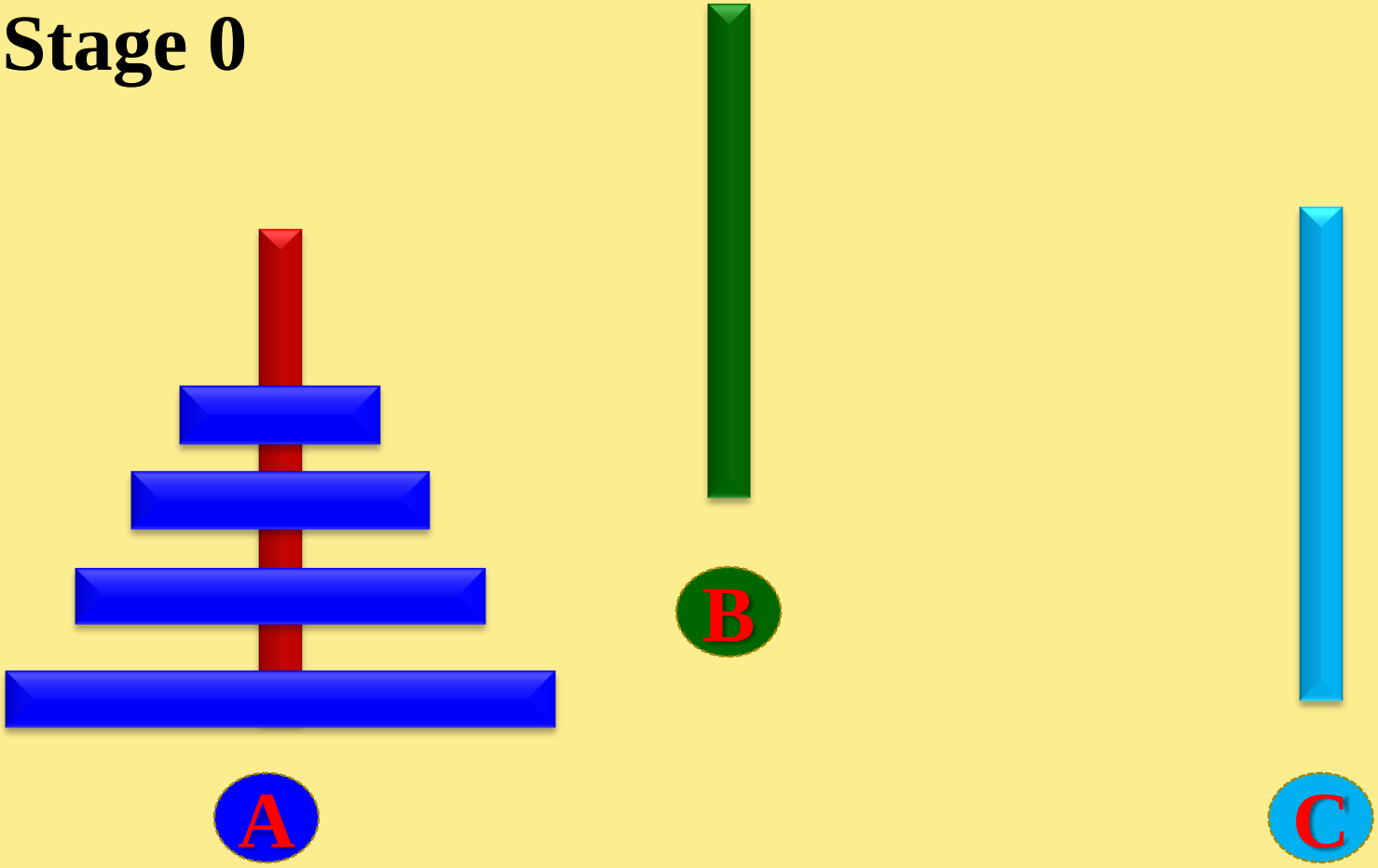
- פתרון: נניח כי ידוע כיצד מעבירים n-1 דיסקים ממגדל מקור למגדל יעד תוך שימוש במגדל עזר.
  - נעביר את הדיסקים 1, 2, ... ,n-1 ממגדל A למגדל C תוך שימוש במגדל B.
  - נעביר את דיסק n ממגדל A למגדל B.
  - נעביר את הדיסקים 1, 2, ... ,n-1 ממגדל C למגדל B תוך שימוש במגדל A.
- בסיס הרקורסיה: כאשר n=0, לא עושים דבר.

פתרון בעיית מגדלי הנוי – ארבעה דיסקים



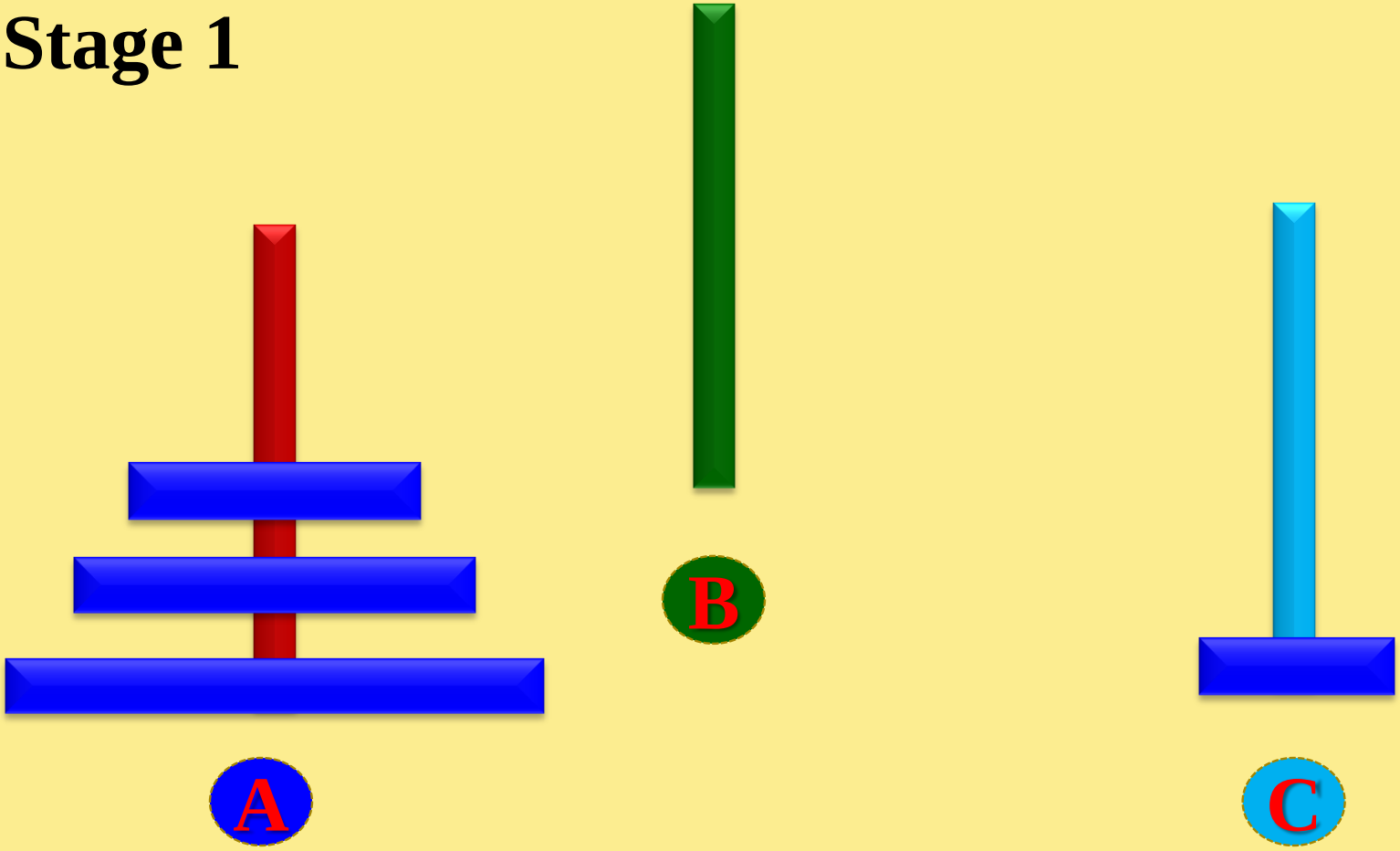
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 0**



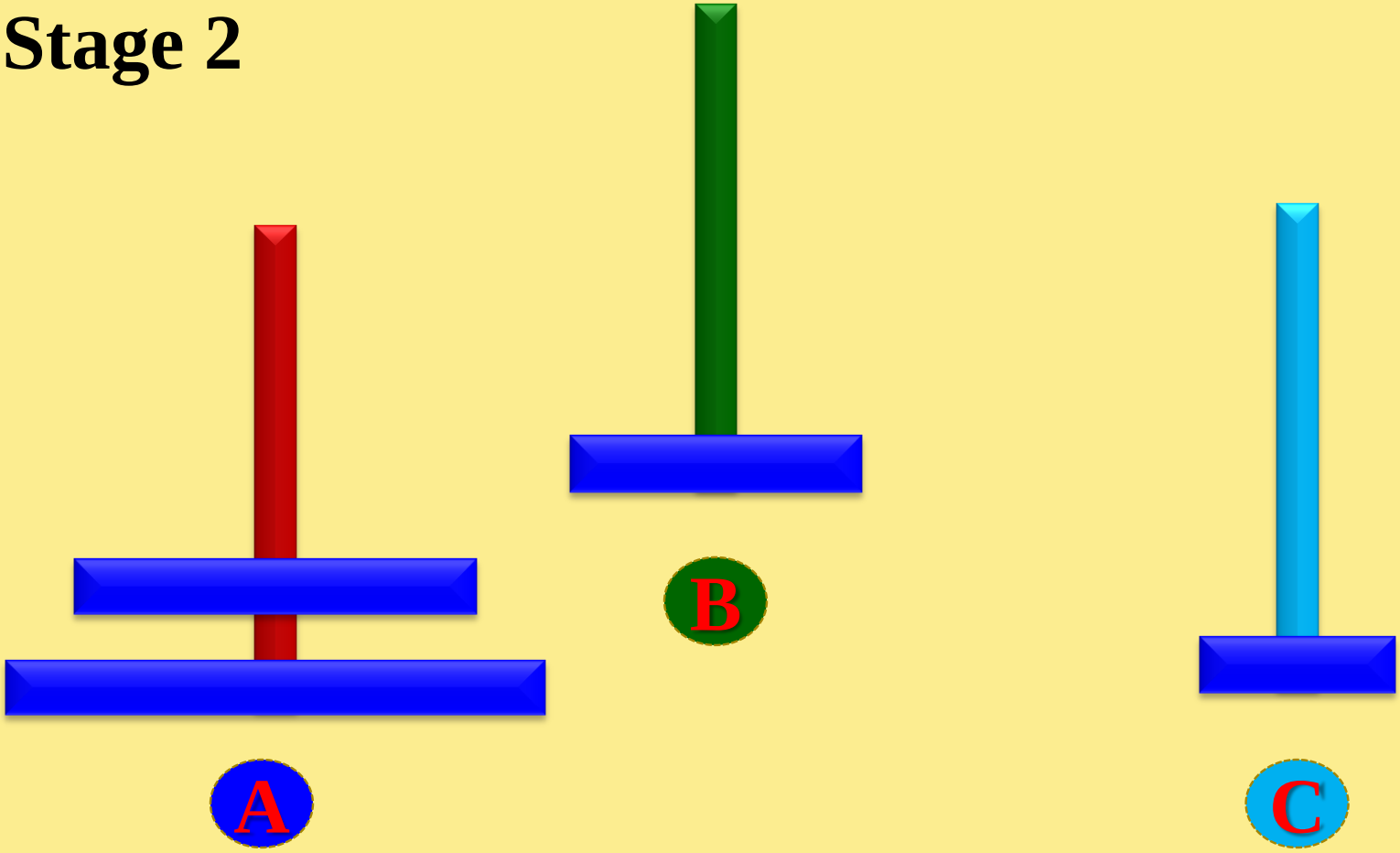
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 1**



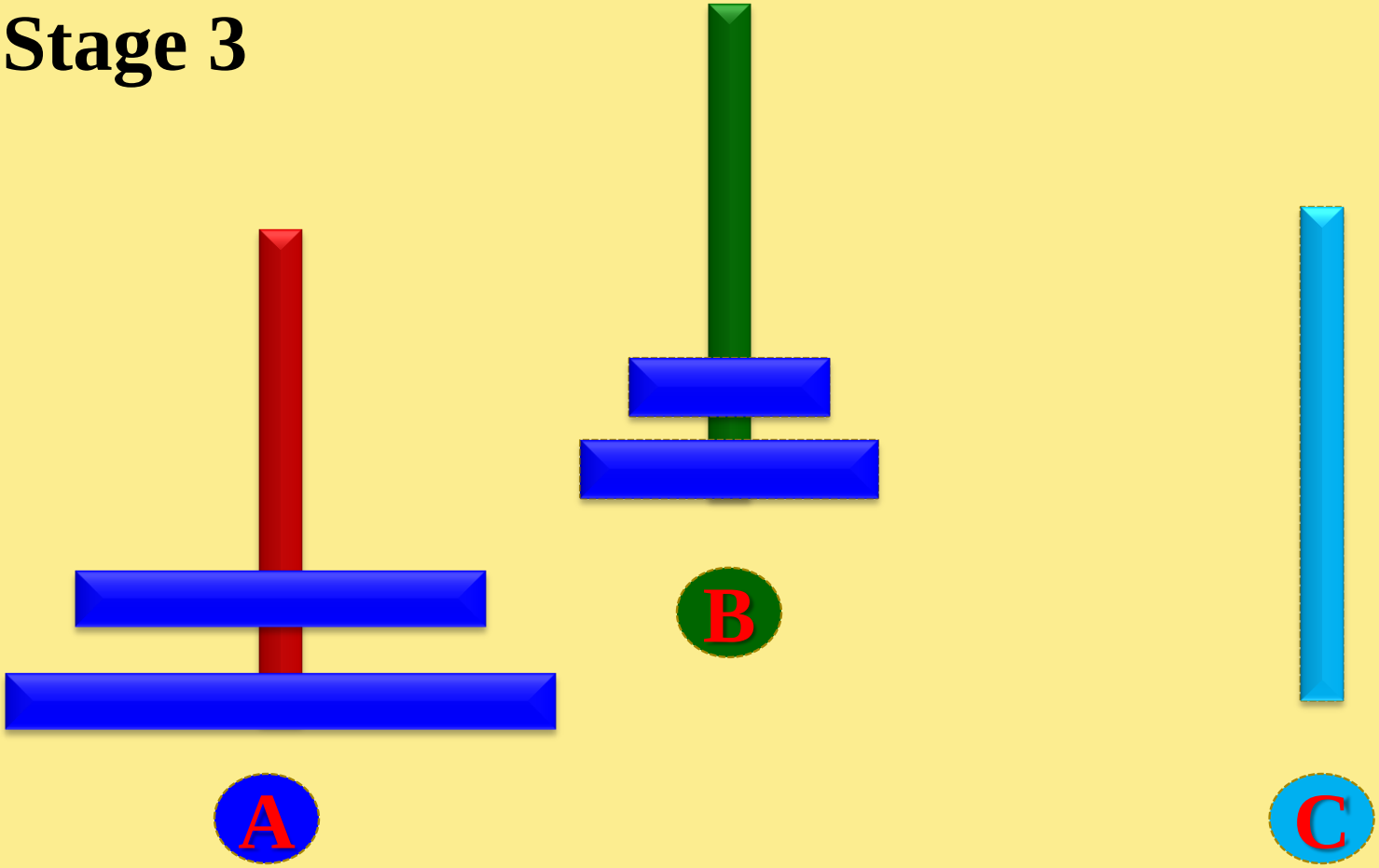
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 2**



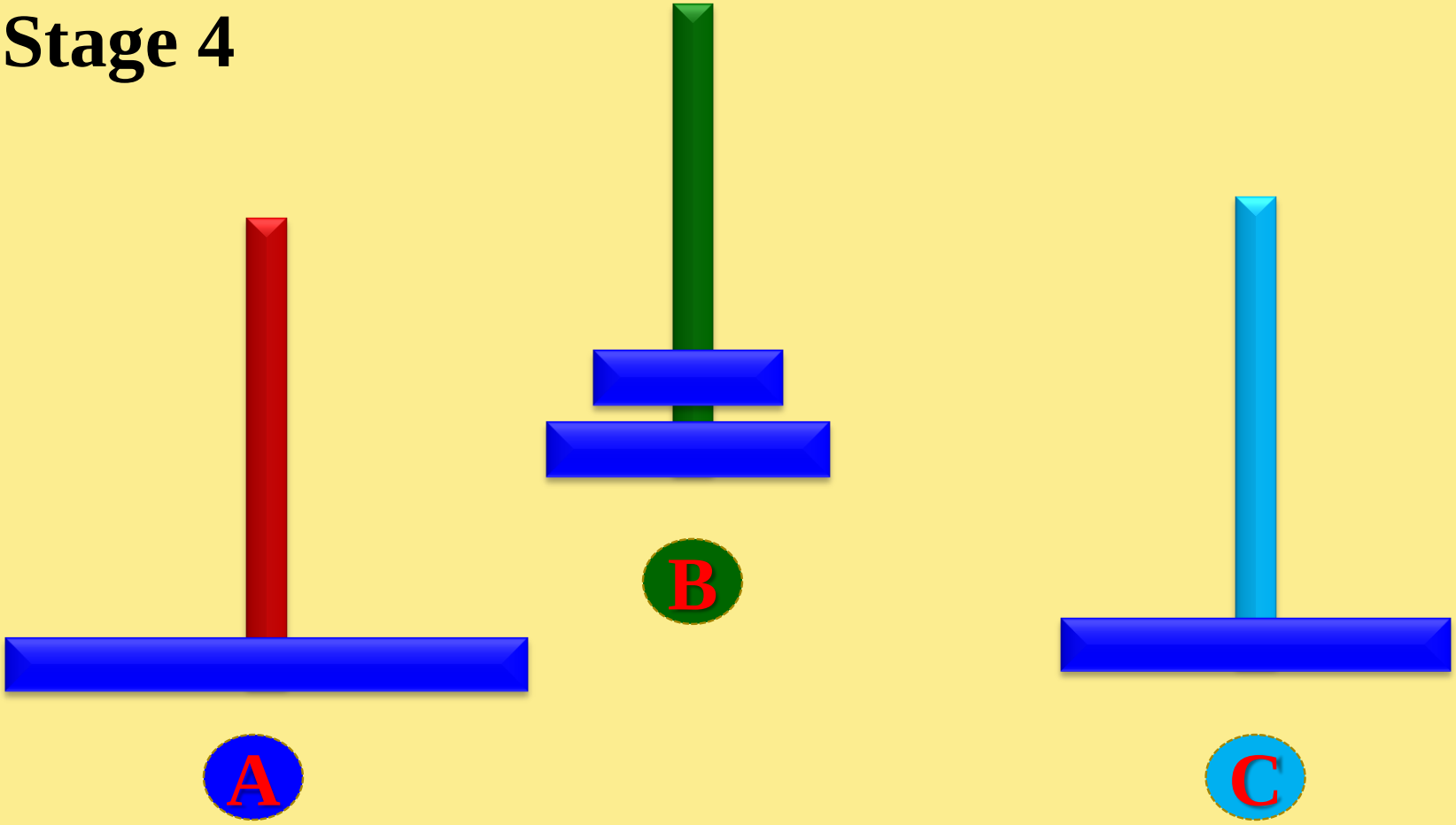
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 3**



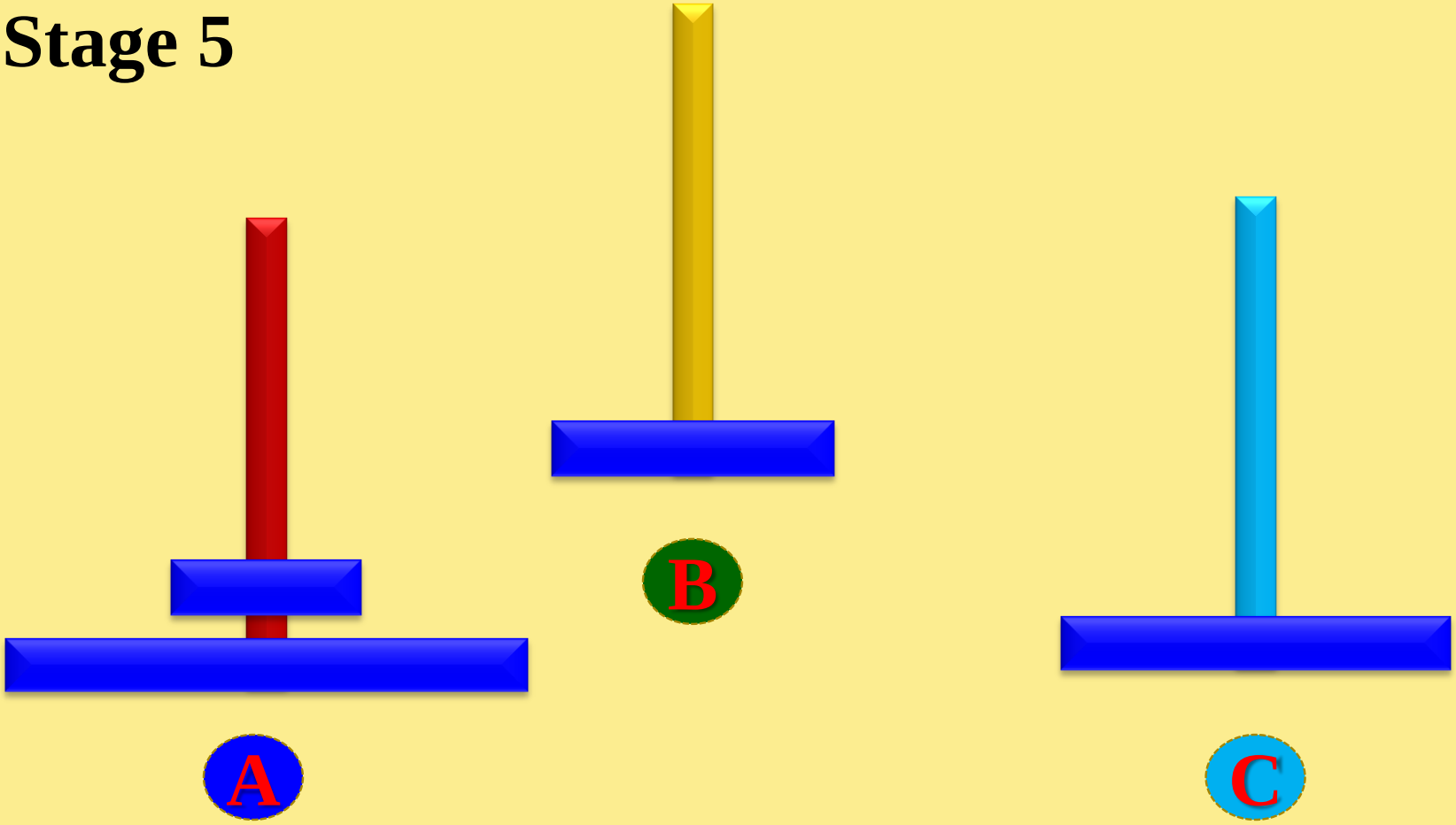
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 4**



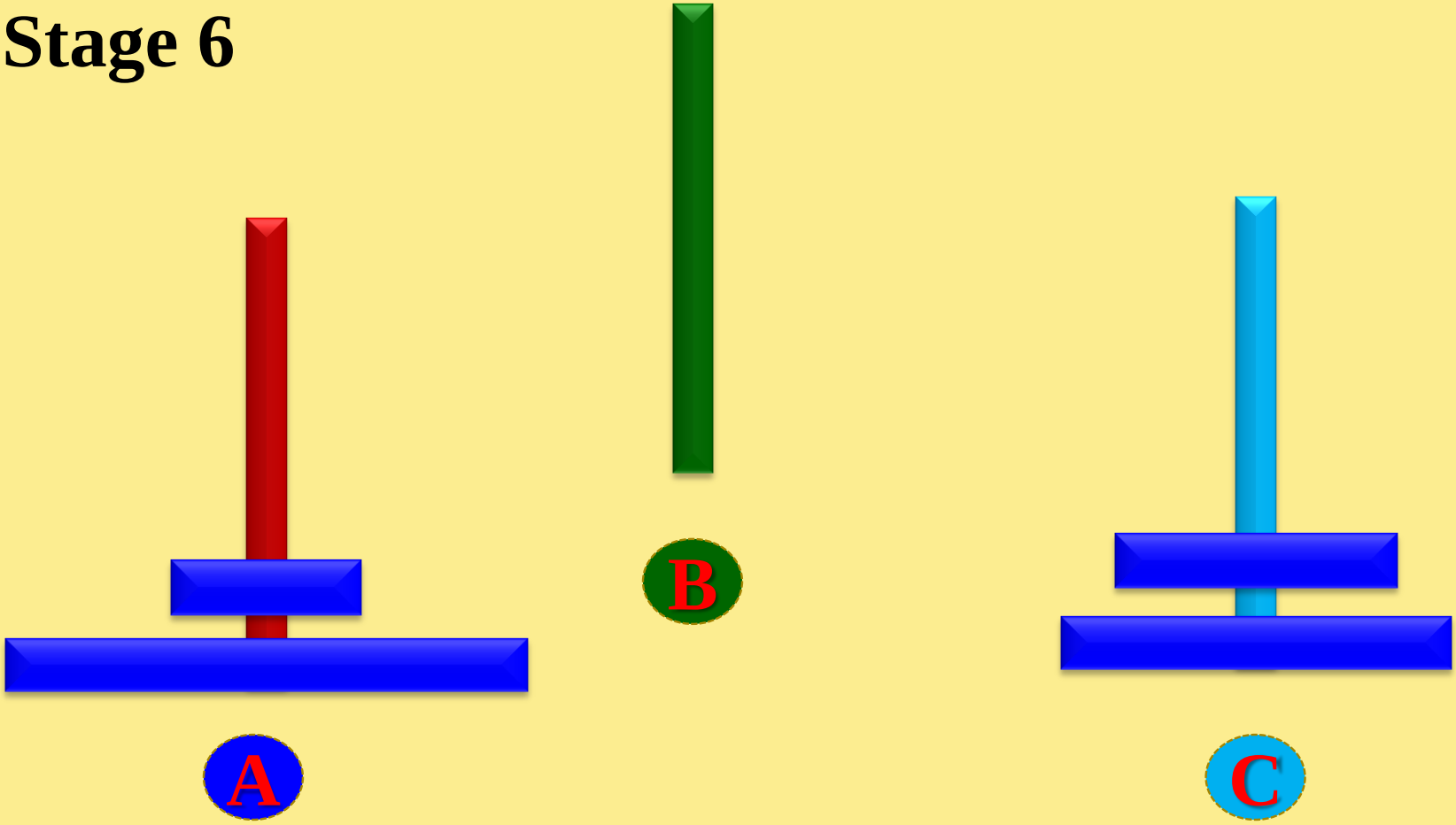
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 5**



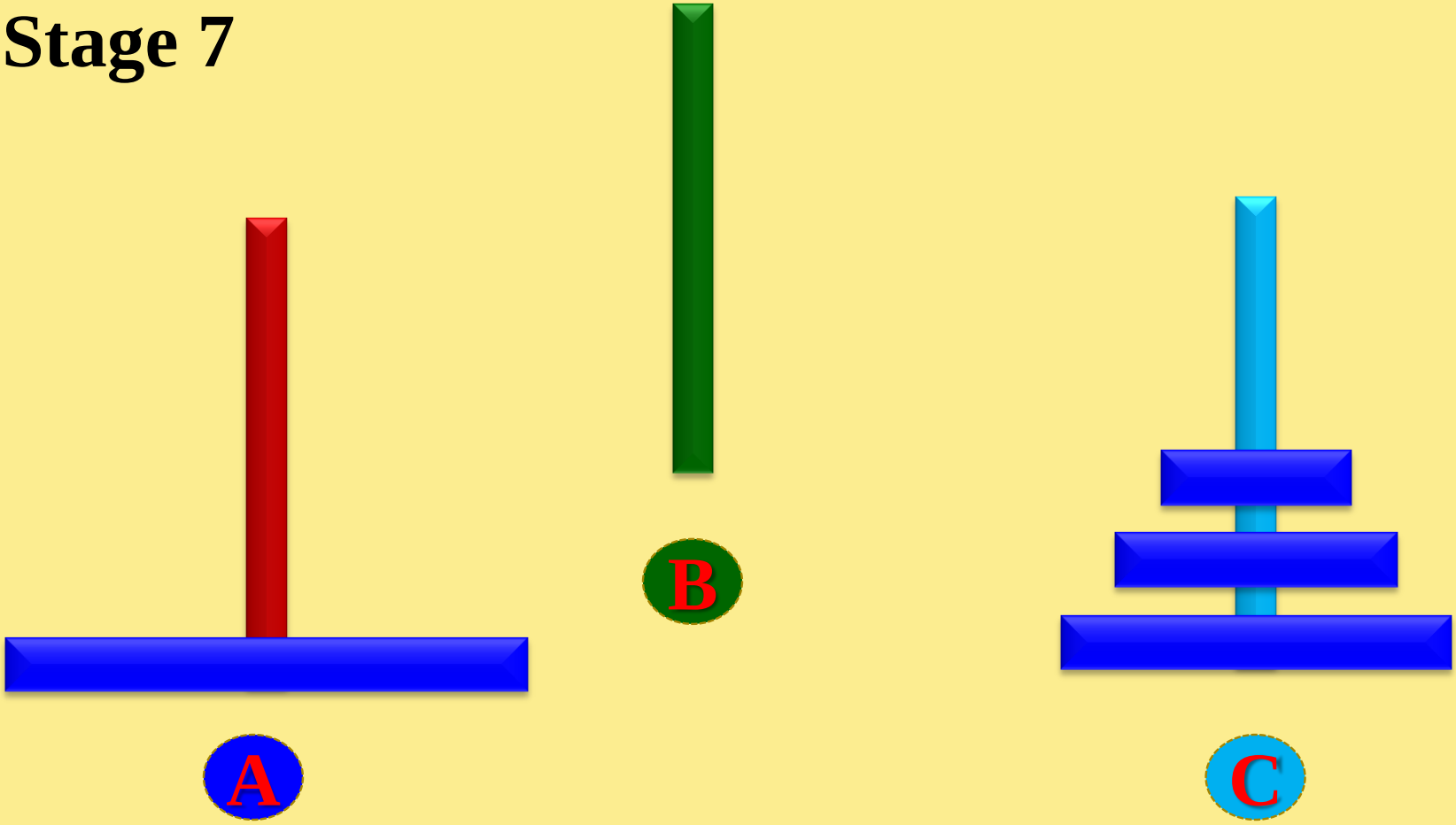
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 6**



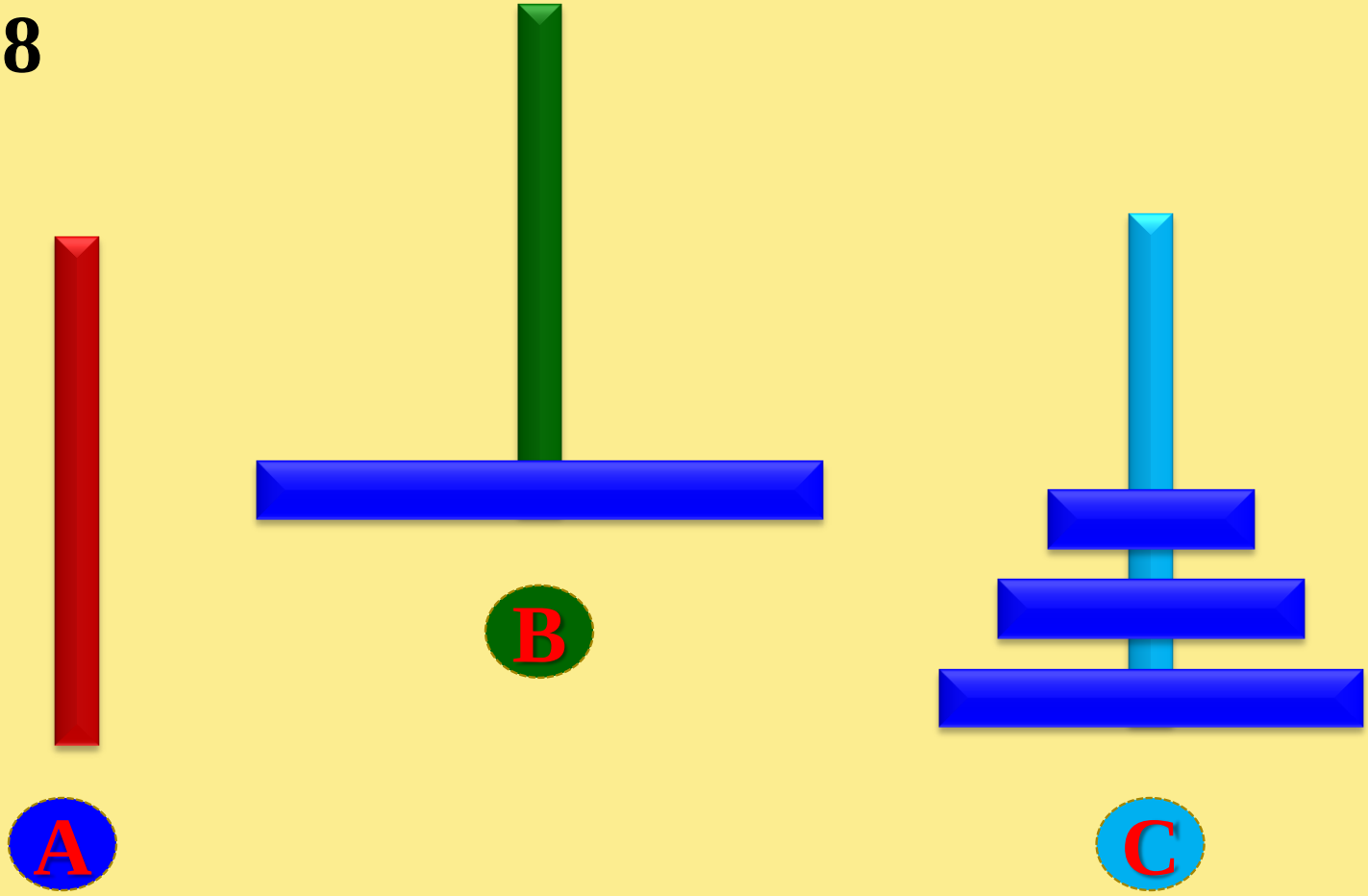
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 7**



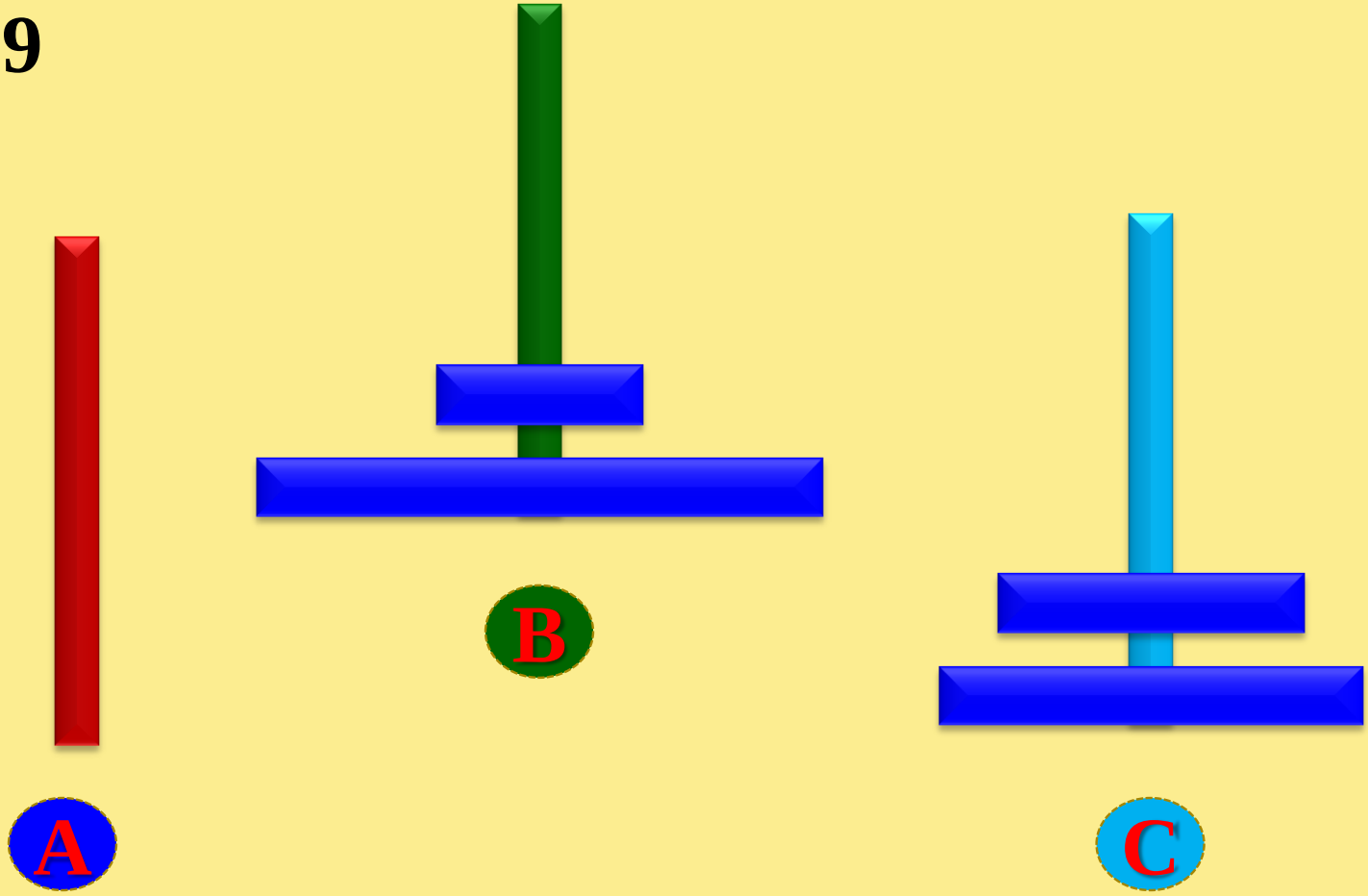
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 8**



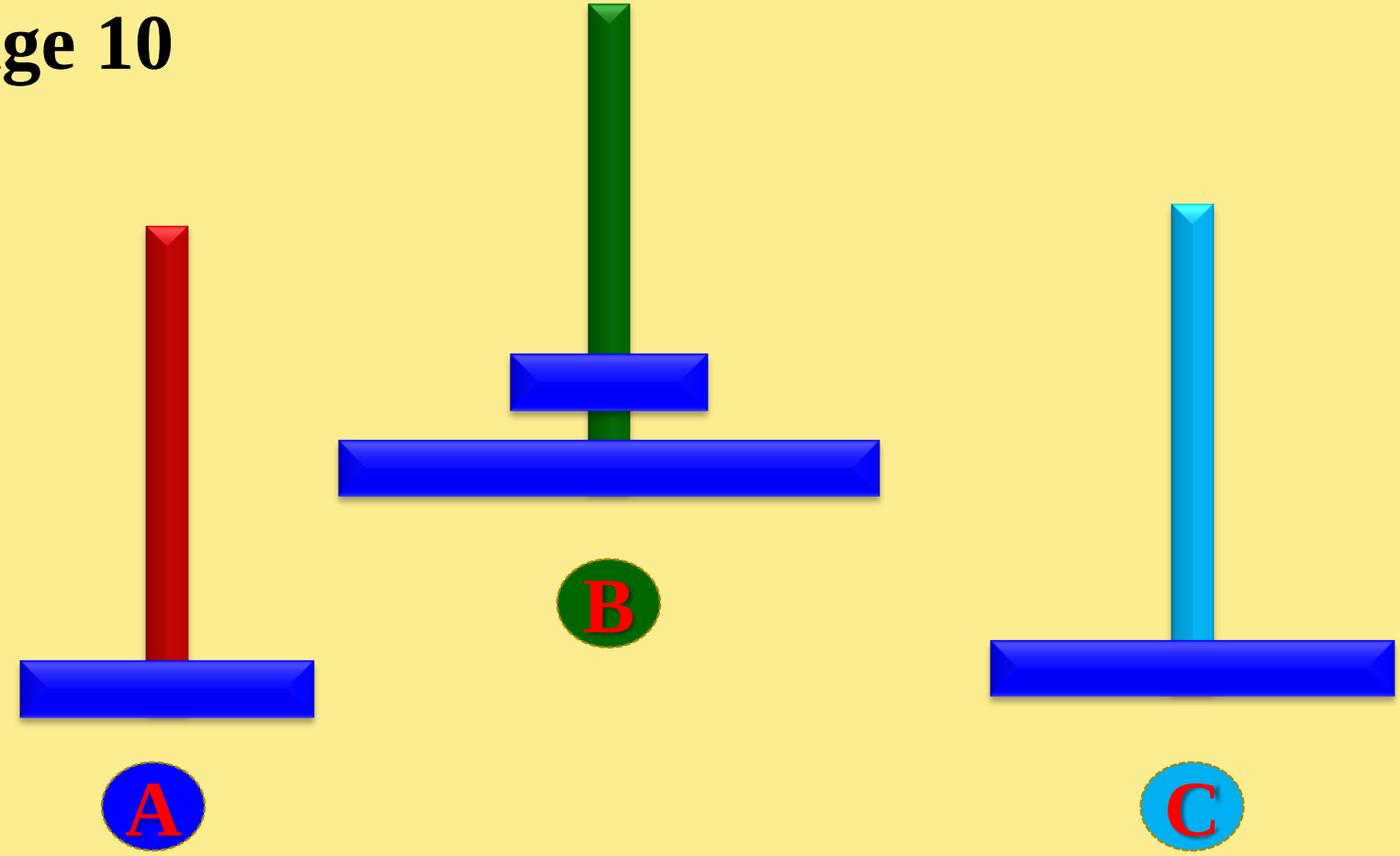
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 9**



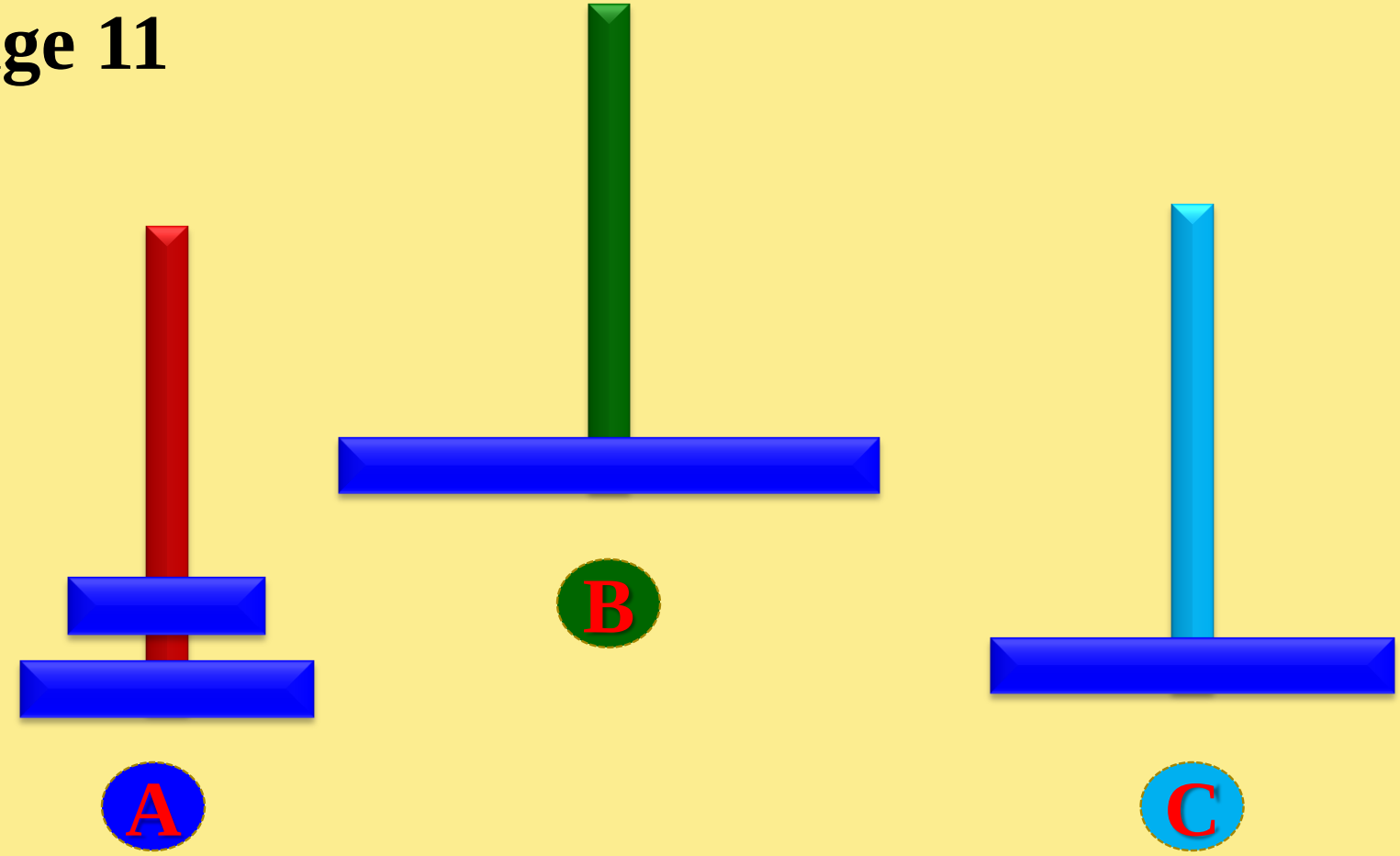
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 10**



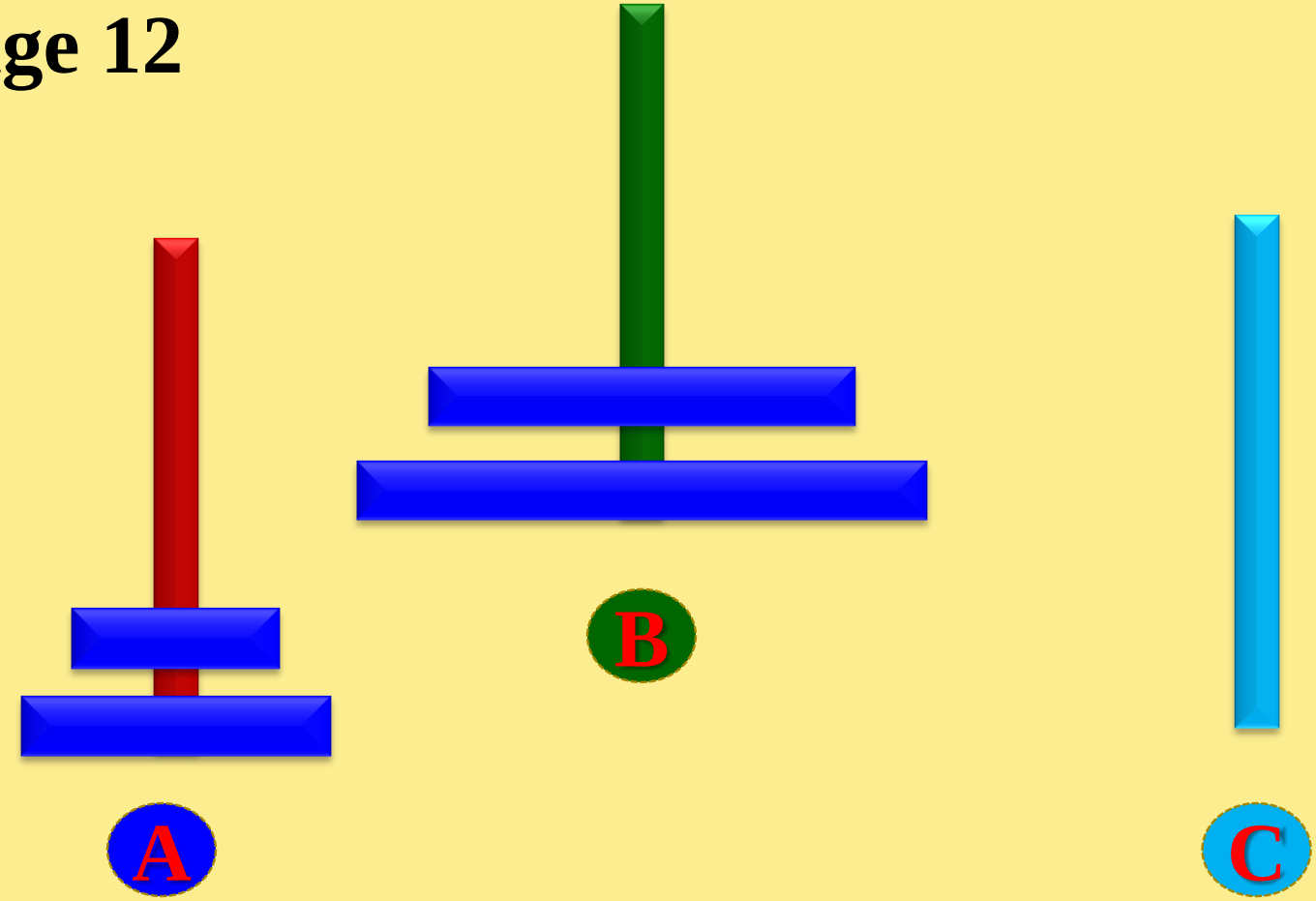
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 11**



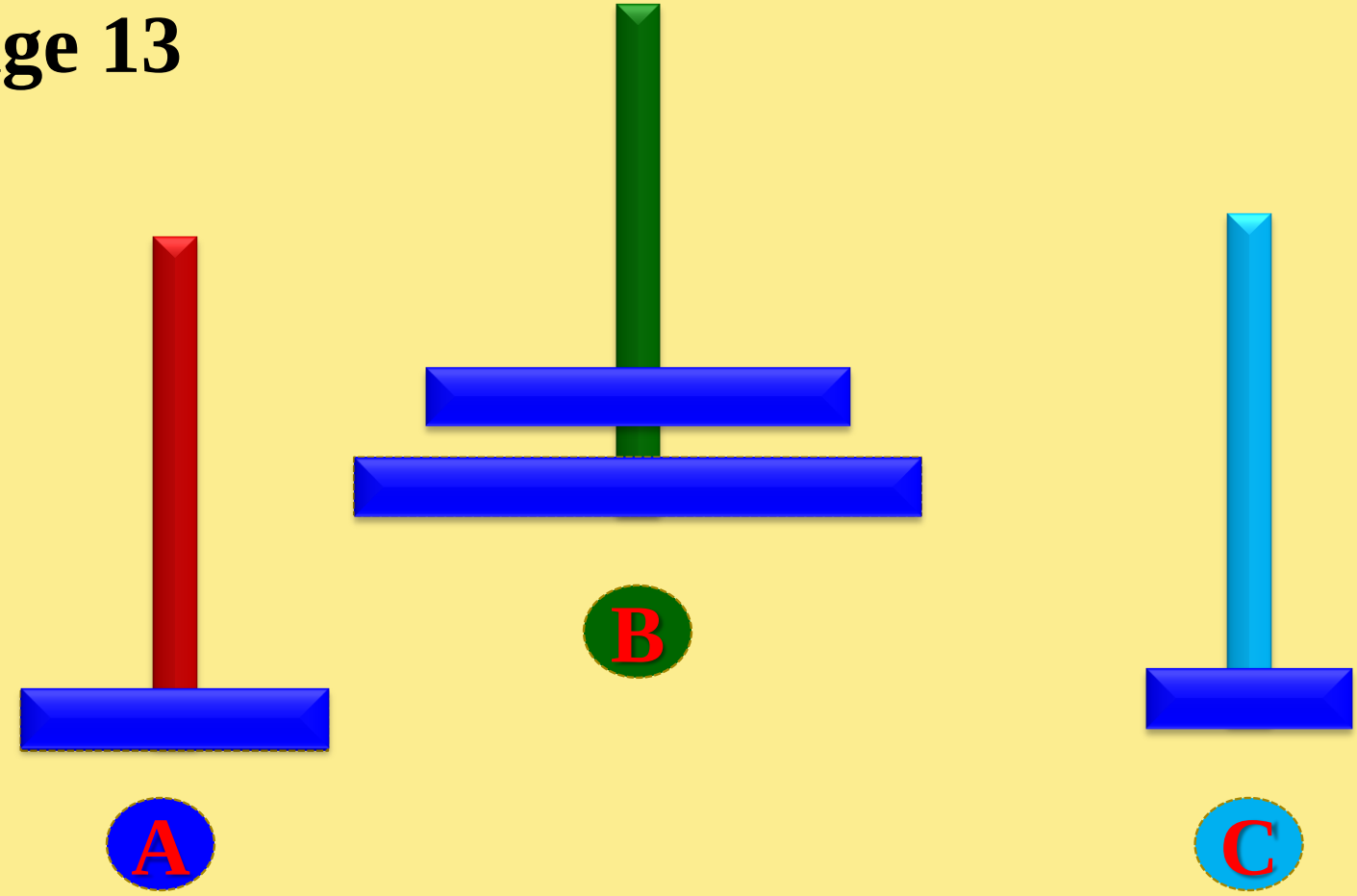
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 12**



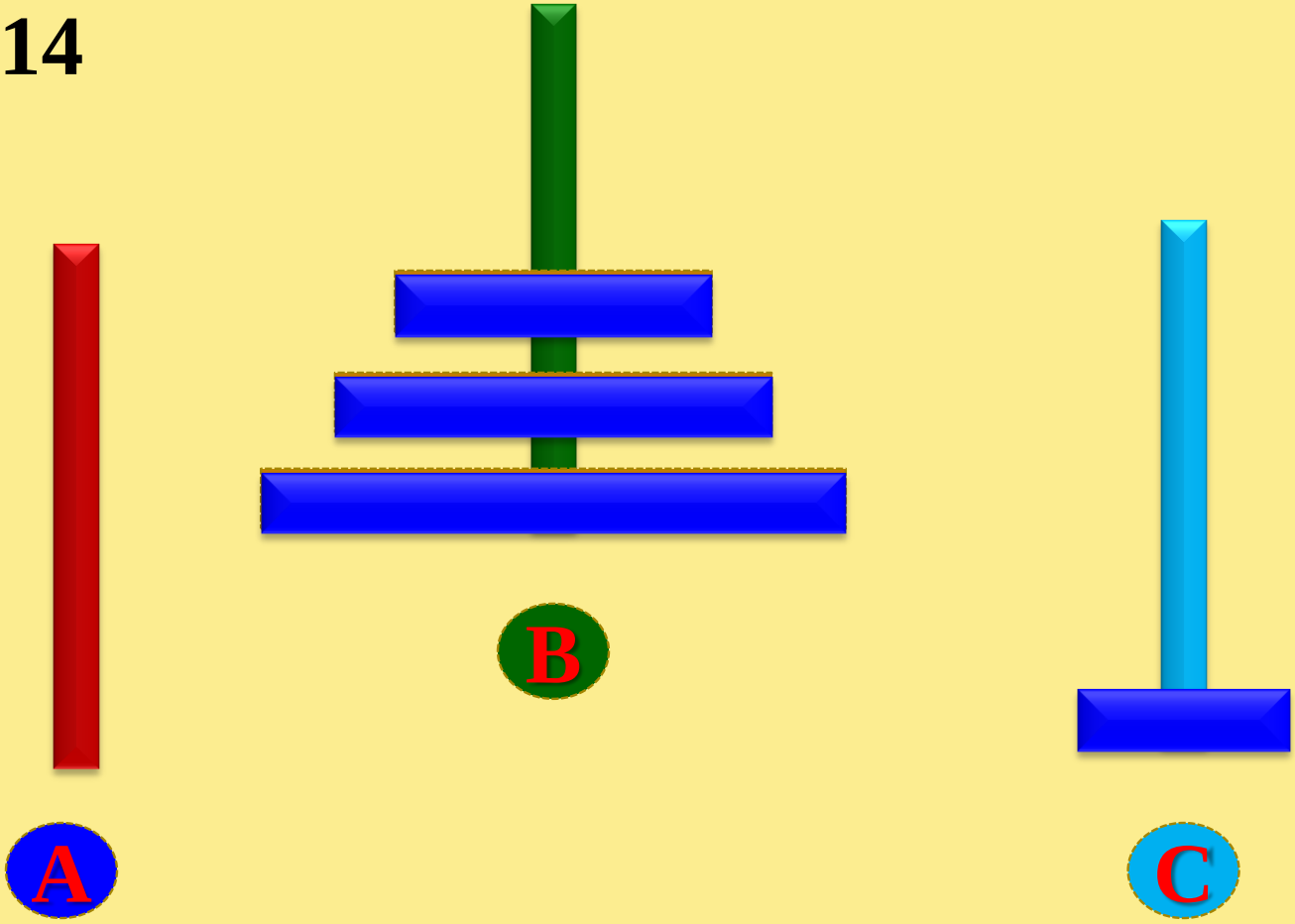
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 13**



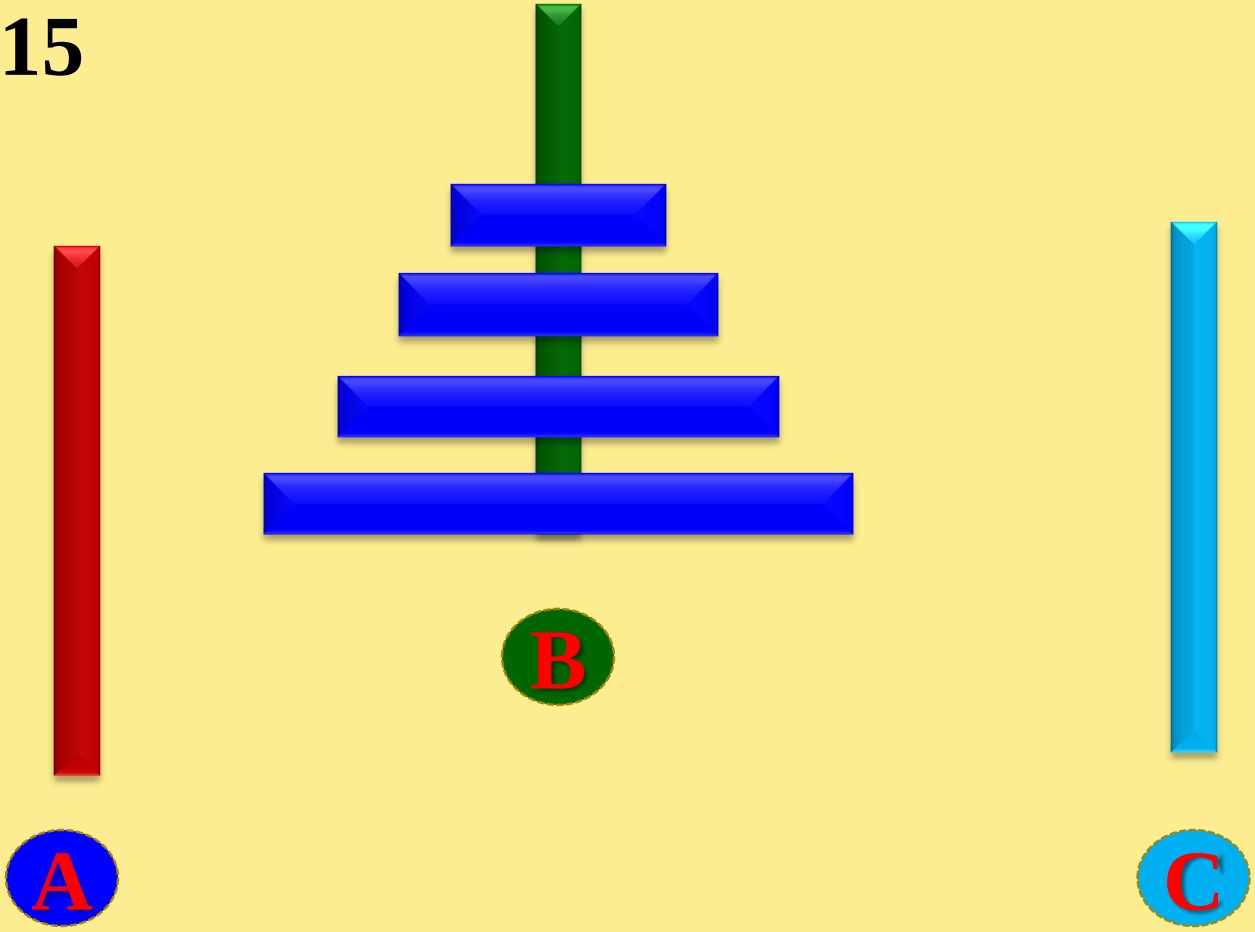
פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 14**



פתרון בעיית מגדלי הנוי – ארבעה דיסקים

- **Stage 15**



מימוש פתרון ברקורסיה – מגדלי הנוי


```
void hanoi(int disknum, char From, char To, char Temp);
int main()
{
    int disknum;
    printf ("Enter number of discs:\n");
    scanf("%d", &disknum);
    hanoi(disknum , 'A', 'B', 'C');
 return 0;
}
```

מימוש פתרון ברקורסיה – מגדלי הנוי


```
void hanoi (int disknum, char From, char To, char Temp)
{
  if(disknum ==1)
        printf("Move disc from tower %c to tower %c\n",
        From, Temp);
  else
    {
        hanoi(disknum-1, From, Temp, To);
        printf("Move disc from tower %c to tower
                %c\n", From, Temp);
        hanoi(disknum-1, To, From, Temp);
    }
}
```

מגדלי הנוי – ניתוח סיבוכיות מקום

- בזמן נתון יש לכל היותר n+1 קריאות לפונקציה `hanoi ()` על מחסנית הקריאות.
- בכל קריאה מוקצים משתנים בסיבוכיות מקום O(1).
- סיבוכיות המקום של האלגוריתם היא O(n).

מגדלי הנוי – ניתוח סיבוכיות זמן

- נסמן ב- $T(n)$ את מספר הקריאות לפונקציה hanoi() הדרושות להעברת מגדל של n חישוקים.
- עלות הפעולות המתבצעות בקריאה בודדת ל-hanoi() הינה $O(1)$.
- סיבוכיות הזמן הכוללת של האלגוריתם הינה $\Theta(2^n)$.

$$T(0) = 1$$

$$T(n) = 1 + 2T(n-1)$$

$$= 1 + 2(1 + 2T(n-2))$$

$$= 3 + 4T(n-2)$$

$$\ldots$$

$$= (2^i - 1) + 2^i T(n-i)$$

$$\ldots$$

$$= (2^n - 1) + 2^n T(0)$$

$$= 2^{n+1} - 1$$

שאלות?

רקורסיה

- **הגדרה:** המונח רקורסיה (recursion) מתאר מצב שבו פונקציה קוראת לעצמה באופן ישיר או באופן עקיף וכך היא מבצעת עבודתה בצורה נסיגתית כלומר מהסוף להתחלה.
- **שימוש:** נוח להשתמש בפונקציות רקורסיביות על מנת לפתור בעיות בעלות אופי רקורסיבי.
- **יתרון יחסי:** באופן כללי, השיטה תהיה להקטין את ממד הבעיה, לפתור את הבעיה על הממד היותר קטן ולהשתמש בפתרון שמתקבל על מנת לפתור את הבעיה בממד אחד יותר גבוהה.

תזכורת – משתנים סטאטיים

- מגדירים באופן הבא: static int sum=0;
- ניתן לאתחל בשורת ההגדרה בלבד כאשר חייבים לאתחל בביטויי קבוע!!!
- אם אנחנו לא מאתחלים אותם, המערכת תאתחל אותם אוטומטית לאפס.
- ה-scope של משתנים סטאטיים הוא רק הפונקציה שבה הם הוגדרו אך הם <u>ממשיכים להתקיים גם אחרי שביצוע הפונקציה נגמר</u> (למעשה עד סוף ריצת התוכנית) .
- לפיכך הערך של משתנה סטאטי בפונקציה כלשהי נשמר בין קריאות עוקבות לפונקציה זו .

דוגמה מס' 1 – harmonic number

- מס' הרמוני של המס' השלם n מוגדר באופן הבא:

$$\frac{1}{n}+\frac{1}{n-1}+\ldots+\frac{1}{2}+1=h(n)$$

- ברצוננו לכתוב פונקציה רקורסיבית rec_harmonic_num(n) אשר תדפיס למסך את הפיתוח של h(n).

- למשל, עבור הקריאה rec_harmonic_num(5), נקבל על המסך: 1/5+1/4+1/3+1/2+1=2.28

דוגמה מס' 1 – harmonic number


```
void rec_harmonic_sum(int n)
{
        static double sum=1;
        if(n==1) /*Stop condition*/
        {
                printf("1=%.2f\n",sum);
                sum=1;
/*Must initialize the static variable "sum" so that we can call
this function repeatedly in the same program run. Make sure
you understand why!!! */
```

harmonic number – 1 'דוגמה מס


```
        return;
        }
        printf("1/%d+",n);
        sum+=1./n;
/*The "1." syntax is used in order to prevent automatic casting
to integer type (so that the expression "1/n" will be evaluated as
type  double). */
        rec_harmonic_sum(n-1);
/*The recursive call. */
}
```

דוגמה מס' 2 – abc

כתוב פונקציה רקורסיבית אשר חתימתה היא:

```
void abc(char arr[],int lastPlace, int curPlace)
```

הפונקציה תקבל מערך של char-ים, את אינדקס סוף המערך ומספר שלם שהוא המקום במערך ממנו אנו מעוניינים להתחיל במילוי המערך בתווים.

בקריאה ראשונה יהיה מס' זה שווה ל- 0.

דוגמה מס' 2 – abc

הפונקציה מדפיסה את כל האפשרויות למלא את המערך מהמקום שקיבלה, curPlace, עד המקום lastPlace באותיות A,B,C.

במילים אחרות, מטרת הפונקציה היא להדפיס את כל האפשרויות ליצור מילה באורך מסוים בעזרת האותיות A,B,C.

דוגמה מס' 2 – abc

למשל, עבור התוכנית הבאה:

```
void abcInvokingFun(char word[],int lengthOfWord)
{
        abc(word, lengthOfWord,0);
}
void main()
{
        char word[SIZE];
        abcInvokingFun(word,3);
}
```

דוגמה מס' 2 – abc


```
void abc(char arr[],int lastPlace, int curPlace)
{
    if (curPlace == lastPlace)
    {
        arr[curPlace] = ' \0';
        printf("%s\t",arr);
        return;
    }
    arr[curPlace] = 'A';
    abc (arr,lastPlace,curPlace+1);
    arr[curPlace] = 'B';
    abc (arr,lastPlace,curPlace+1);
    arr[curPlace] = 'C';
    abc (arr,lastPlace,curPlace+1);
}
```

תנאי העצירה:

אם הגענו ל- lastPlace אז נשים במקום הנוכחי יש '\0' (הגענו לסוף), ונדפיס את המחרוזת (מהתחלתה) וחזור.

כעת אנחנו עובדים על המקום curPlace במערך:

- מציבים בו A, וקוראים לפונק' abc (עם כתובת המערך, ואינדקס המקום הבא) אשר תדאג למילוי כל האפשרויות הקיימות בשאר המערך.
- לאחר שחוזרים, מציבים את B במקום A, ושוב קוראים לפונקציה עם המקום הבא במערך כפרמטר.
- כנ"ל חוזרים על הפעולה לגבי התו- C.

דוגמה מס' 2 – ABC

נקבל את הפלט הבא:

AAA	AAB	AAC	ABA	ABB	ABC
ACA	ACB	ACC	BAA	BAB	BAC
BBA	BBB	BBC	BCA	BCB	BCC
CAA	CAB	CAC	CBA	CBB	CBC
CCA	CCB	CCC			

רקורסיה - סיכום

- פונקציה רקורסיבית היא פונקציה הקוראת לעצמה, כלומר מוגדרת בעזרת הפעלתה עבור פרמטרים יותר קטנים/פשוטים.
- הקריאה העצמית לפונקציה מתבצעת בצירוף תנאי התחלה עבור פרמטר כלשהו, שמובטח שנגיע אליו במהלך החישוב.
- כשיש לנו נוסחה רקורסיבית, קל לתרגם אותה לפונקציה רקורסיבית ב-C.
- לא תמיד קל למצוא ניסוח רקורסיבי אם הוא לא נתון לנו ולעיתים עדיף לבצע חישוב איטרטיבי.

רקורסיה - סיכום

- אם בכל שלב הפונקציה קוראת לעצמה יותר מפעם אחת, אז כבר עבור ערכים קטנים ידרשו הרבה זמן וזיכרון לחישוב.
- לכן, ניתן לחשוב כי החישוב הרקורסיבי הוא בעייתי במקרה כזה.
- אם הפונקציה קוראת לעצמה רק פעם אחת בכל שלב, אז כתיבה רקורסיבית יכולה לשמש כדרך קצרה ואלגנטית לביצוע חישובים ופעולות נוספות.
- דוגמאות נוספות יוצגו בתרגול.

שאלות?

תרגילי כיתה

1. כתבו פונקציה רקורסיבית המקבלת מספר שלם ומדפיסה אותו הפוך מהסוף להתחלה.

- <u>דוגמא</u>: בהינתן המספר 1234 הפונקציה הרקורסיבית תדפיס 4321.
- מה יופיע על המסך אם נבצע את ההדפסה אחרי הבדיקה?
- שדרג את הפונקציה הרקורסיבית שתייצר מספר חדש הפוך מהמקורי.

2. כתבו פונקציה רקורסיבית שעושה אותו דבר למחרוזת.
3. נסה לבנות פונקציה רקורסיבית המחזירה את התו המקסימלי.

תרגילי כיתה

4. כתבו תוכנית הקולטת עד 10 ערכים שלמים למערך, מדפיסה אותו ומוצאת את הערך המקסימאלי ואת מיקומו במערך(קלט של המספר 1- מציין סוף קלט) .

לשם כך, ניתן לפרק את הבעיה לארבע בעיות קטנות יותר:

a) כתבו פונקציה רקורסיבית שקולטת את המערך.

b) כתבו פונקציה רקורסיבית שמחזירה את הערך המקסימאלי.

c) כתבו פונקציה רקורסיבית המוצאת את מיקומו של הערך המקסימלי שבמערך.

d) כתבו פונקציה רקורסיבית המדפיסה את איברי המערך.

מומלץ להקפיד כי בכל סיבוב קליטה של ערך לתוך המערך יש תנאי יציאה ידוע ומבוקר !!!

תרגילי כיתה

5. כתוב פונקציה רקורסיבית אשר מקבלת מחרוזת s, תו ch, ומספר שלם n.

- הפונקציה תחפש את המופע ה- n-י של ch במחרוזת s ותחזיר את האינדקס שלו.
- <u>דוגמא</u>: בהינתן המחרוזת "axftbbncf" , התו b, והשלם 2 (עבור n) הפונקציה תחזיר 2.
- התו 'b' מופיע פעמיים החל מהמקום החמישי, כתובת מספר 4, במחרוזת s.

תרגילי כיתה

6. נתונה הפונקציה הרקורסיבית הבאה:

```
#include <stdio.h>
int secret( int n)
{
    if( n<0 )
        return 1 + secret ( -1 * n);
    if ( n<10 )
        return 1;
    return 1 + secret( n/10 );
}
```

a) מה הערך של secret(-4321) ו- secret(12345) ?

b) מה מבצעת הפונקציה secret עבור פרמטר חיובי ועבור שלילי?

c) כתוב פונקציה זו מחדש בצורה לא רקורסיבית.

תרגילי כיתה

7. עיין בפונקציה הבאה:

```
int what(int a, int b)
{
    if(!a && !b)
        return 1;
    if(a > b)
        return a * what(a-1, b);
    return b * what(a, b-1);
}
```

תרגילי כיתה

בהנחה שהפונקציה הנ"ל מקבלת שני ערכים אי-שליליים (חיוביים או אפס), סמן את כל התשובות הנכונות (בדף התשובות):

1. הפונקציה נכנסת לרקורסיה אינסופית – שקר/אמת
2. הערך המוחזר של הפונקציה יכול להיות 1 – שקר/אמת
3. הערך המוחזר של הפונקציה תמיד 0 – שקר/אמת
4. הערך המוחזר של הפונקציה יכול להיות 0 – שקר/אמת
5. הערך המוחזר של הפונקציה תמיד 1 – שקר/אמת
6. בקבלת שני ערכים <u>חיוביים</u> a ו- b, אם הערכים לא גדולים מידי, הפונקציה מחזירה את הערך של - a! x b! - שקר/אמת
7. אף לא אחת מהתשובות לעיל – שקר/אמת

תרגילי כיתה

8. לפניך פונקציה רקורסיבית, בנה טבלת מעקב לתאור פעולת הפונקציה במידה וקיבלה את המספר 123 והצג את התוצאה:

```
int what (int n)
{
        int k;
        if(n<10)  return n;
        else
        {
                k = what ((n/100)*10 + n%10);
                return (k*10 + (n%100) /10);
        }
}
```

תרגילי כיתה

9. כתוב תוכנית המבצעת את המשימות הבאות:

- מגדירה ומייצרת מערך חד ממדי מלא ב-25 מספרים רנדומליים.
- מתפעלת פונקציה שתמיין מיון בועות את המערך.
- התוכנית תתפעל פונקציה רקורסיבית לביצוע חיפוש בינארי במערך הממוין של מספר הנקלט מהמשתמש.
- במידה והמספר שנקלט נמצא בפונקציה הרקורסיבית תציג הודעה מתאימה.

תרגילי כיתה

10. כתוב תוכנית המבצעת את המשימות הבאות:

- מגדירה ומייצרת מערך חד ממדי מלא ב-10 מספרים רנדומליים.
- התוכנית תתפעל פונקציה רקורסיבית לביצוע מיון בחירה של המערך.
- התוכנית תדפיס את המערך המקורי כפי שנוצר ואת המערך החדש הממוין.

עוד דוגמאות לרקורסיה

דוג' 2 - מה עושה התוכנית הבאה?


```
void count_dn(int count) {
    --count;
   printf ("Count = %d\n", count);
   if (count > 0)
          count_dn(--count);
   if (count == 0)
          printf ("Middle!\n");

   printf ("Count = %d\n", count);
}
```

count_dn(7) - output:

Count = 6

Count = 4

Count = 2

Count = 0

Middle!

Count = 0

Count = 1

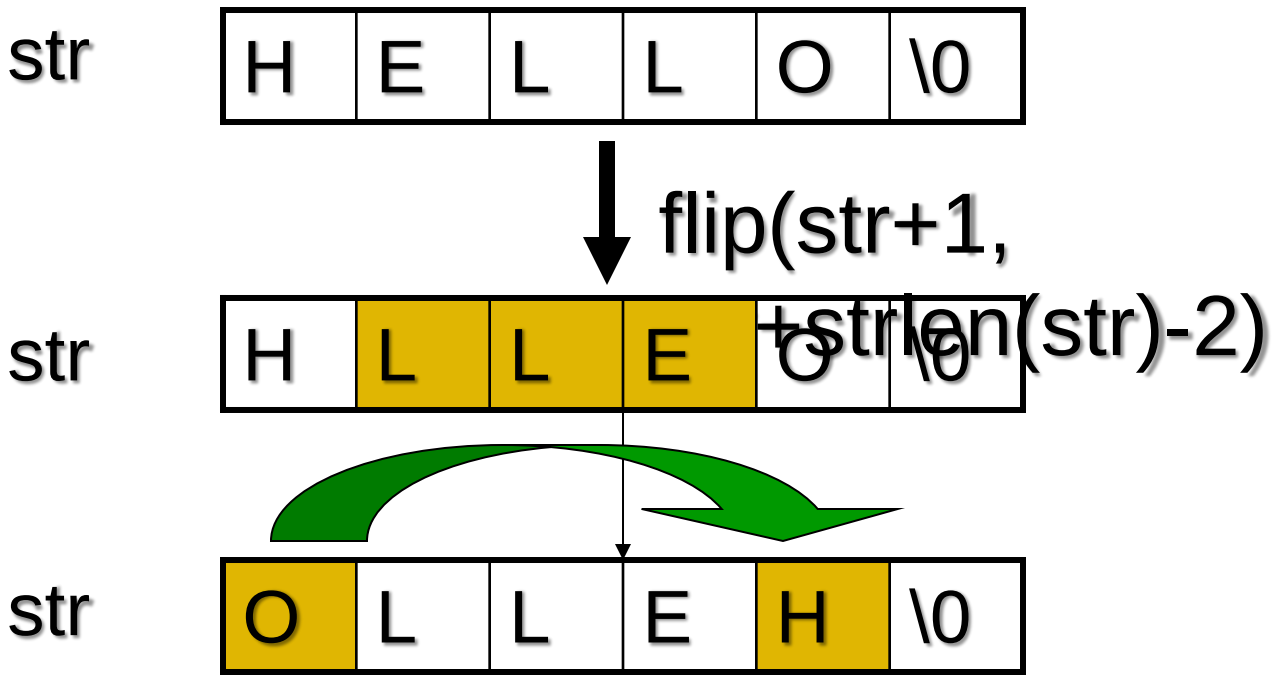
Count = 3

Count = 5

דוג' 3 - שאלה

- עליכם לכתוב פונקציה flip המקבלת מחרוזת והופכת את איברי המחרוזת.
- למשל עבור s = "Hello World" הפונקציה תהפוך את s להיות המחרוזת s = "dlroW olleH".

דוג' 3 – הבנת הפתרון



דוג' 3 - פתרון


```
void flip_hlp (char *begin, char *end)  {
   if (begin >= end) return;
   swap (begin, end);
   flip_hlp(begin+1, end-1);
}


void flip(char *str) {
   flip_hlp(str, str+strlen(str)-1);
}
```

דוג' 4 - Quick Sort

- נתון מערך באורך n.
- על מנת למיין את המערך, נבחר את המספר שנמצא במקום האמצעי במערך (אם n זוגי, נבחר את n/2), נקרא לו – pivot, ונסמנו ב-k.
- נחליף את k עם האיבר השמאלי ביותר.
- נעבור על כל איברי המערך.
- את כל האיברים הקטנים מ-k נעביר לתחילת המערך (אחרי k)
- לבסוף, נעביר את k למקומו במערך.
- ברגע זה, כל האיברים שממוקמים לפניו קטנים ממנו, וכל האיברים שממוקמים אחריו גדולים ממנו.
- נחזור על התהליך עבור חלק המערך הקטן מ-k, ועבור החלק הגדול ממנו.
- כאשר מגיעים למערך באורך 1, חוזרים בחזרה.
- בסיום התהליך, מקבלים מערך ממוין בסדר עולה.

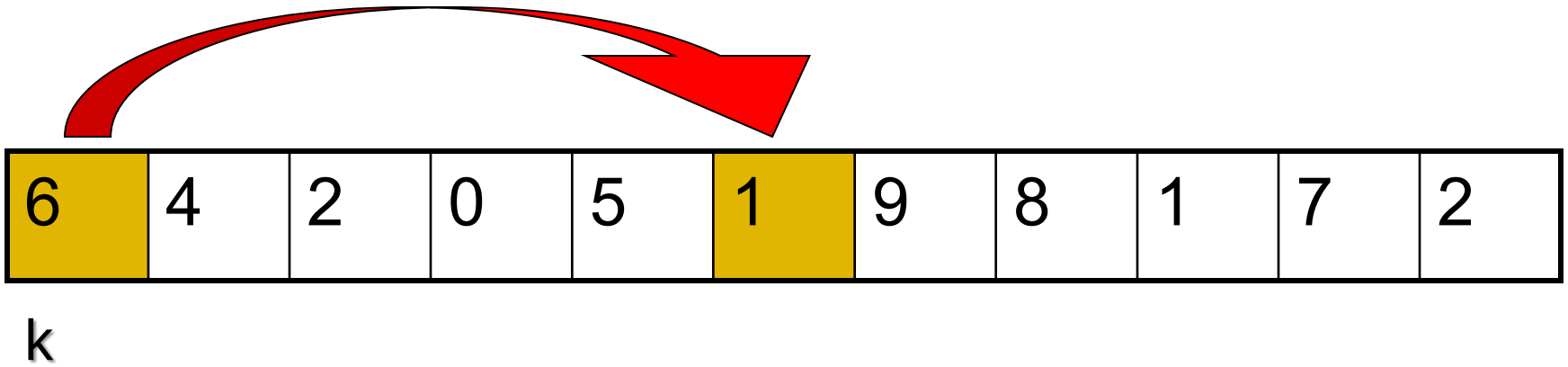
```
//determining the pivot
swap(vec, left, (left+right)/2);
```

1	4	2	0	5	6	9	8	1	7	2

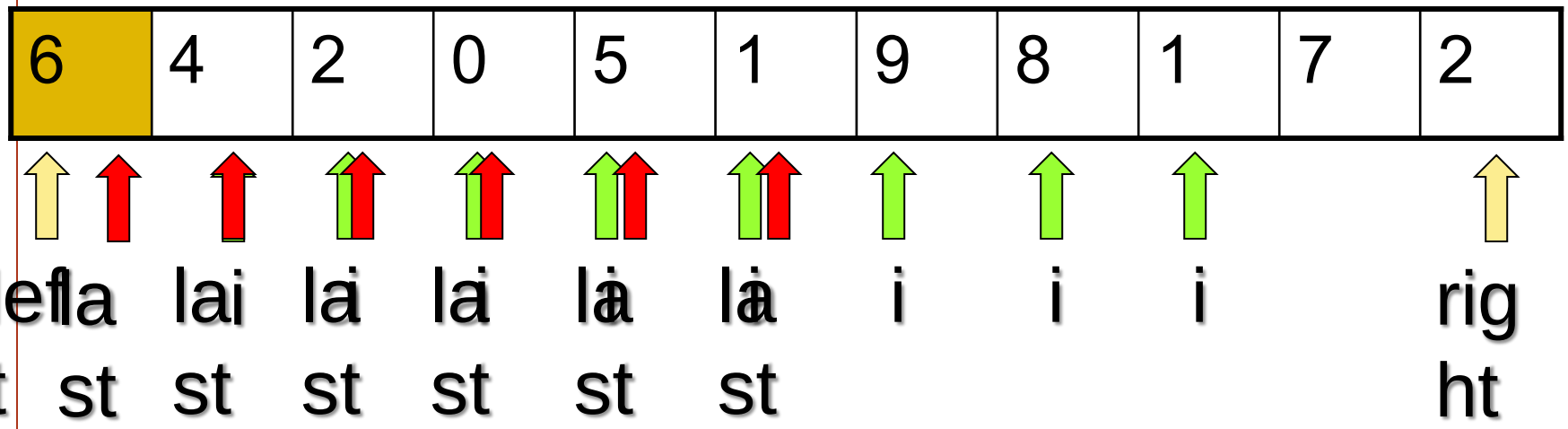
k

//determining the pivot

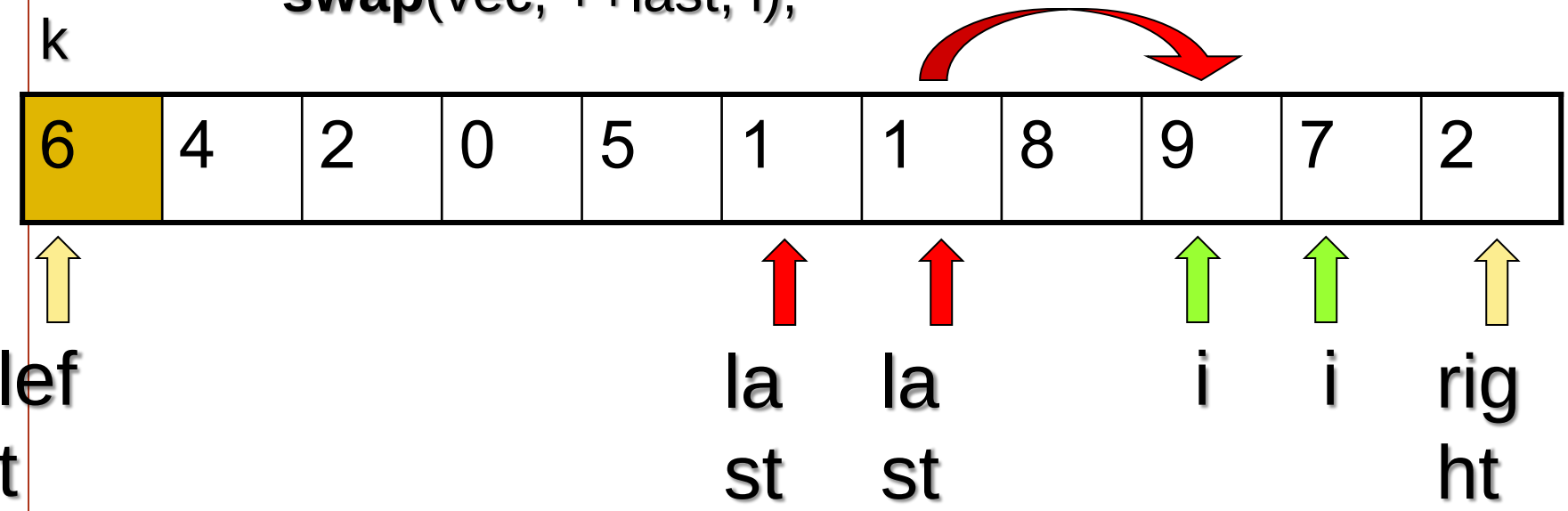
swap(vec, left, (left+right)/2);



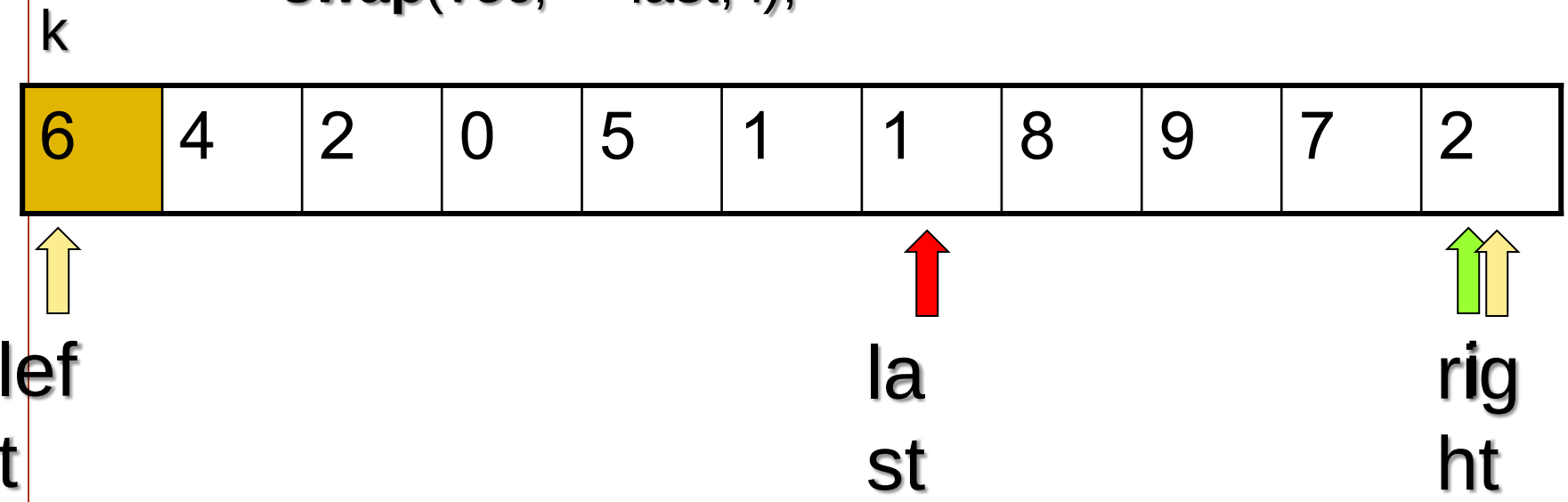
```
//moving all the smaller element
for (i=left+1; i<=right; i++)
    if (vec[i] < vec[left])
        swap(vec, ++last, i);
```



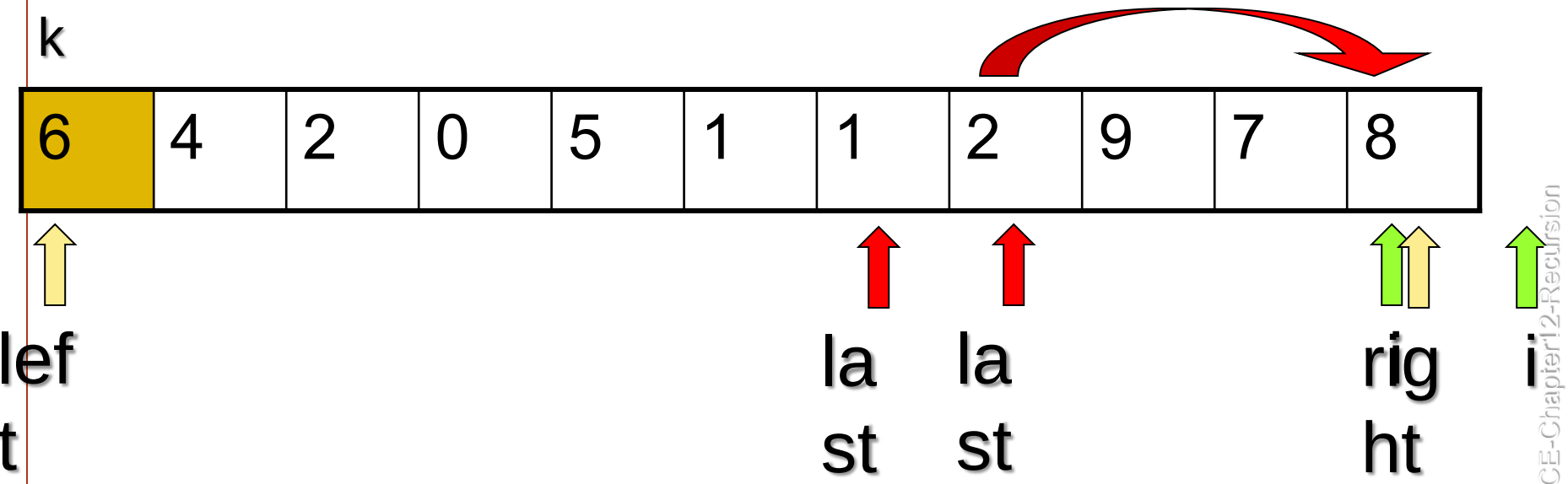
```
//moving all the smaller element
for (i=left+1; i<=right; i++)
    if (vec[i] < vec[left])
        swap(vec, ++last, i);
```



```
//moving all the smaller element
for (i=left+1; i<=right; i++)
    if (vec[i] < vec[left])
        swap(vec, ++last, i);
```

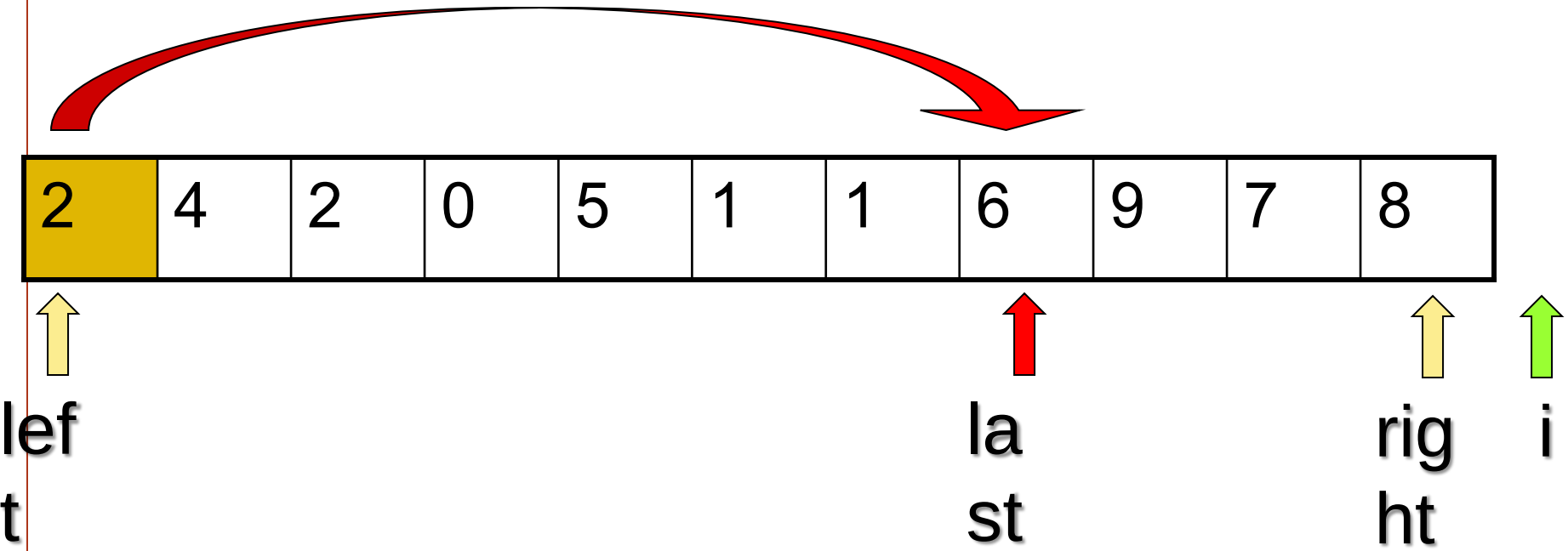


```
//moving all the smaller element
for (i=left+1; i<=right; i++)
    if (vec[i] < vec[left])
        swap(vec, ++last, i);
```



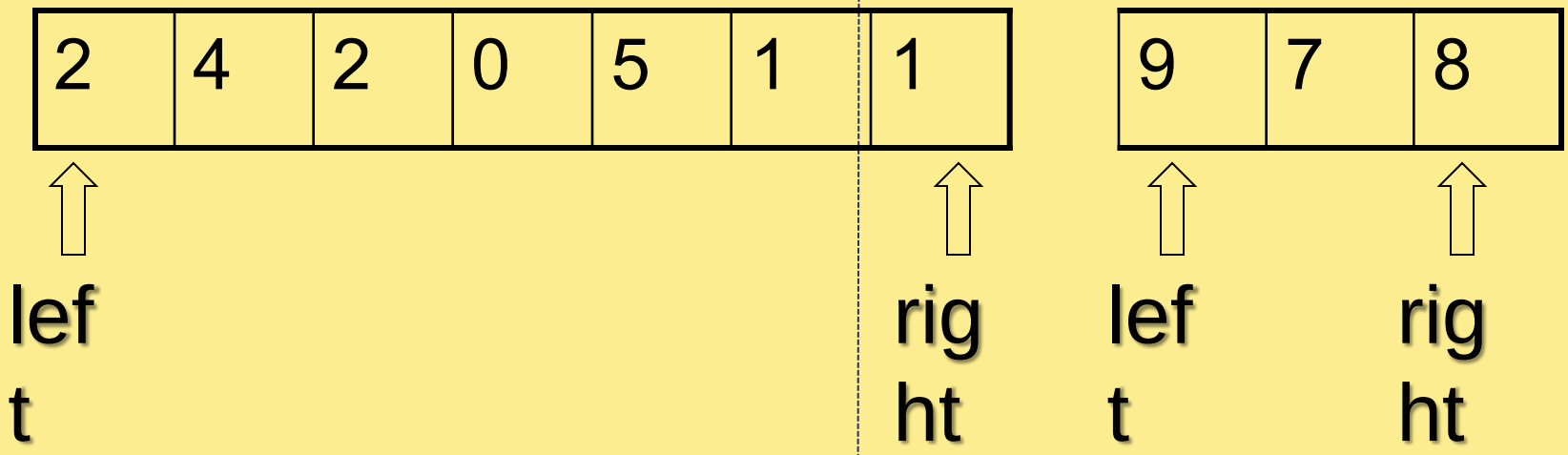
//moving the pivot to its place

swap(vec,left,last);



//continue sorting both sides

```
qsort(vec, left, last-1);
qsort(vec, last+1, right);
```



דוג' 4 - הקוד

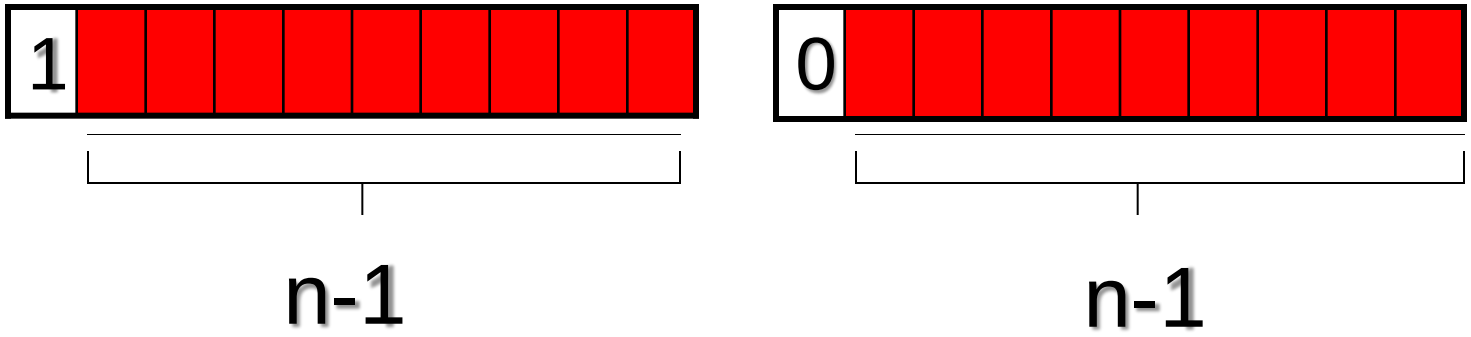

```
void qsort(int vec[], int left, int right) {
    int i, last;
    //stopping condition
    if (left>=right)
        return;
    //determining the pivot
    swap(vec, left, (left+right)/2);
    last = left;
    //moving all the smaller elements
    for (i=left+1; i<=right; i++)  {
        if (vec[i] < vec[left]) {
                swap(vec, ++last, i);
        }
    }
    //moving the pivot to its place
    swap(vec,left,last);

    //continue sorting both sides
    qsort(vec, left, last-1);
    qsort(vec, last+1, right);
}
```

דוג' 5 - ווקטורים בינאריים

- נרצה לכתוב תוכנית המחשבת את מספר הווקטורים הבינאריים באורך n

= מספר הווקטורים באורך n-1 כשמוסיפים להם ביט ראשון 0

+ מספר הווקטורים באורך n-1 שמוסיפים להם ביט ראשון 1.



דוג' 5 - ווקטורים בינאריים - הקוד


```
int CalcNum(int n) {
   if (n == 1)
         return 2;
   return 2*CalcNum(n-1);
}
```

דוג' 5 ב' - ווקטורים בינאריים

- טוב, זה לא חוכמה. הרי אנחנו יודעים כי מספר הווקטורים הבינאריים באורך n הוא 2^n .
- אבל כיצד נדפיס את כל הווקטורים הבינאריים באורך n?

Backtracking

- Backtracking היא שיטה לפתרון בעיות שבהם צריכים לעבור על כל מרחב הפתרונות האפשריים.
- במקרה זה אנו מחפשים שיטה לפתרון הבעיה ועוברים על כל המקרים האפשריים באופן רקורסיבי.

דוגמא ל- Backtracking

- נתון מבוך בגודל NxN ורוצים לדעת האם יש פתרון למבוך, ז"א האם קיים מסלול מהנקודה (0,0) לנקודה (N,N) שעובר אך ורק במשבצות שיש בהן את הערך 1.

1	0	0	0	1	0	0	0	1	0
1	0	0	0	1	0	0	1	1	0
1	1	1	1	1	1	1	1	0	1
0	1	0	0	0	1	0	1	0	1
1	1	1	0	0	1	0	1	1	1
0	0	1	1	1	1	0	0	0	0
0	0	1	1	0	0	0	0	1	1
0	0	0	1	1	0	0	0	1	0
0	0	0	1	1	1	1	0	1	0
0	0	1	1	0	0	1	1	1	1
0	1	1	0	0	0	0	0	0	1

1	0	0	0	1	0	0	0	1	0
1	0	0	0	1	0	0	1	1	0
1	1	1	1	1	1	1	1	0	1
0	1	0	0	0	1	0	1	0	1
1	1	1	0	0	1	0	1	1	1
0	0	1	1	1	1	0	0	0	0
0	0	1	1	0	0	0	0	1	1
0	0	0	1	1	0	0	0	1	0
0	0	0	1	1	1	1	0	1	0
0	0	1	1	0	0	1	1	1	1
0	1	1	0	0	0	0	0	0	1

הרעיון

- נעבור על כל המסלולים האפשריים.
- בעיה: אנחנו יכולים להיכנס ללולאה אינסופית.
  - לדוגמא: (0,0) → (1,0) → (0,1) → (0,0) → ...

- פתרון: נסמן את המקומות במבוך שביקרנו בהם וכך לא נחזור על עצמנו.

נגדיר:

```
enum {CANT_WALK, CAN_WALK, WALKED}
```

1	0	0	0	1	0	0	0	1	0
1	0	0	0	1	0	0	1	1	0
1	1	1	1	1	1	1	1	0	1
0	1	0	0	0	1	0	1	0	1
1	1	1	0	0	1	0	1	1	1
0	0	1	1	1	1	0	0	0	0
0	0	1	1	0	0	0	0	1	1
0	0	0	1	1	0	0	0	1	0
0	0	0	1	1	1	1	0	1	0
0	0	1	1	0	0	1	1	1	1
0	1	1	0	0	0	0	0	0	1

פתרון המבוך


```
int SolveMaze(int maze[N][N]) {
    return SolveMazeHlp(maze,0,0);
}
int SolveMazeHlp(int maze[N][N], int i, int j) {
    int solved = 0;
    if (maze[i][j] != CAN_WALK)  return 0;
    // else - SolveMaze[i][j] == CAN_WALK
    maze[i][j] = WALKED;
    if ((i == N-1) && (j == N-1)) //we're now at the end of the maze
            return 1;
    if (i < N-1)        solved = solved ||  SolveMazeHlp (maze, i+1, j); //go down
    if (i > 0)          solved = solved ||  SolveMazeHlp (maze, i-1, j);  //go up
    if (j < N-1)        solved = solved ||  SolveMazeHlp (maze, i, j+1); //go right
    if (j > 0)          solved = solved ||  SolveMazeHlp (maze, i, j-1);  //go left
    if (solved)
            return 1;
    maze[i][j] = CANT_WALK;             //could not finished mazed from here
    return 0;
}
```

פתרון המבוך – המשך

- נשים לב כי בסוף תהליך הריצה אם קיים פתרון למבוך הוא יקבל את הערך 2 (WALKED)

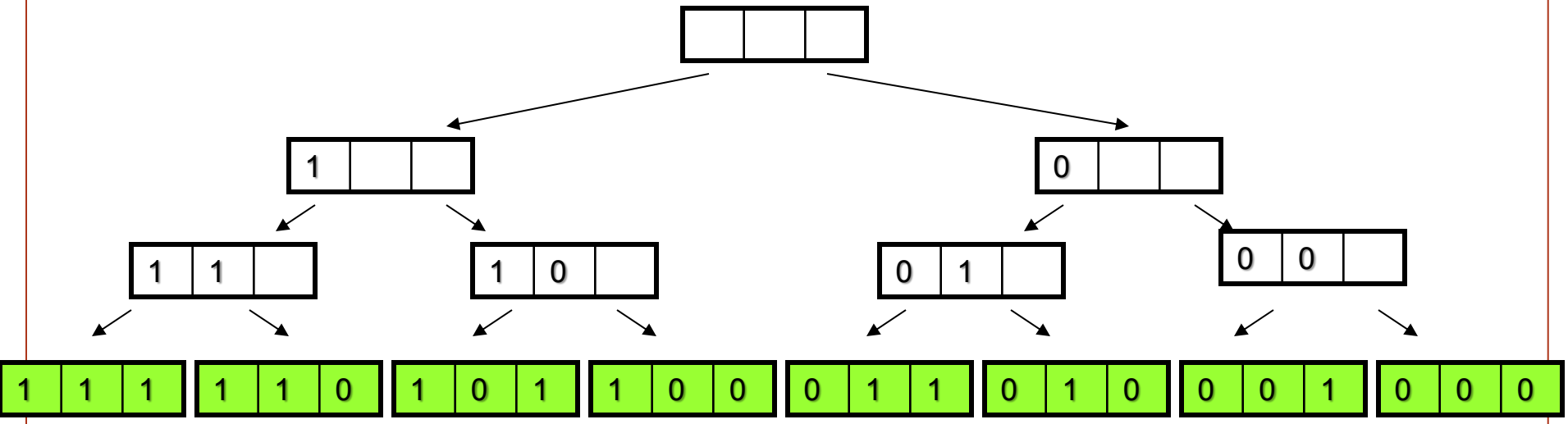
2	0	0	0	1	0	0	0	1	0
2	0	0	0	1	0	0	1	1	0
2	2	1	1	1	1	1	1	0	1
0	2	0	0	0	1	0	1	0	1
1	2	2	0	0	1	0	1	1	1
0	0	2	1	1	1	0	0	0	0
0	0	2	2	0	0	0	0	1	1
0	0	0	2	1	0	0	0	1	0
0	0	0	2	2	2	2	0	1	0
0	0	1	1	0	0	2	2	2	2
0	1	1	0	0	0	0	0	0	2

דוג' 5ב' - ווקטורים בינאריים

- נחזור לשאלה המקורית:
- נרצה לכתוב תוכנית המדפיסה את כל הווקטורים הבינאריים באורך n, איך נעשה זאת?

דוג' 5ב' - ווקטורים בינאריים

- נשמור ווקטור באורך n ונמלא אותו ברקורסיה
- כאשר הווקטור מתמלא נדפיס אותו



דוג' 5ב' - ווקטורים בינאריים - הקוד


```
void BinHelp(int n, int arr[], int place) {
    if (place == n)  {
       print_arr(arr, n);
       return;
    }

    arr[place] = 0;
    BinHelp(n, arr, place+1);

    arr[place] = 1;
    BinHelp(n, arr, place+1);
}
```

```
void Bin(n) {
    int* arr =
        (int*)malloc(sizeof(int)*n);
    BinHelp(n, arr, 0);
}
```