קורס תכנות

שיעור חמישי: רקורסיה

פונקציות

טיפוס הערך המוחזר | שם הפונקציה | משתני הקלט | גוף הפונקציה | החזרת ערך

```
type function_name( type argument_1 ... )
{
   statement 1
   statement 2
   ...
   ...
   return value;
}
```

פונקציות

```
#include <stdio.h>

double power(double base, int exponent);      // הכרזה

int main()
{
   double result = power(2,20) + power(3,15) + power(5,17);      // שימוש

   printf("2^20 + 3^15 + 5^17= %lf", solution);
   return 0;
}

double power(double base, int exponent)      // הגדרה
{
   int i=0;
   double result=1;

   for (i=1; i<=exponent; i++)
      result = result*base;

   return result;
}
```

power(2,20)

base = 2;
exponent = 20;

משתנים בפונקציה

- משתנים בפונקציה:
  - מוכרים אך ורק בפונקציה הזאת (ב-scope שלה).
  - לא מוכרים בפונקציות אחרות (גם לא ב-main).
    - כולל גם את משתני הקלט.
- בסיום ריצת פונקציה:
  - המשתנים לא מוגדרים יותר.
  - ערכי המשתנים לא נשמרים מקריאה אחת לשנייה.

בזמן ריצה

source code

```
double power(double base, int exponent)
{
   int i = 0;
   double result = 1;

   for (i = 1; i <= exponent; i++)
      result = result * base;

   return result;
}

int main()
{
   double result = power(2,20) +
                   power(3,15) +
                   power(5,17);

   printf("2^20 + 3^15 + 5^17= %g",
      result);
   return 0;
}
```

call stack

main: result

power: base = 2, exponent = 20, i, result

5

מגדלי הנוי

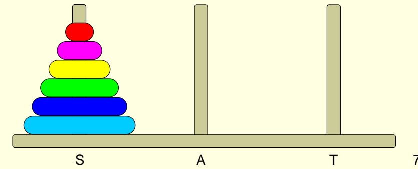
- **משימה:**
  - העבירו את כל הדיסקיות מיתד S (source) ליתד T (target)
  - השתמשו ביתד עזר A (auxiliary)

S A T

6

החוקים

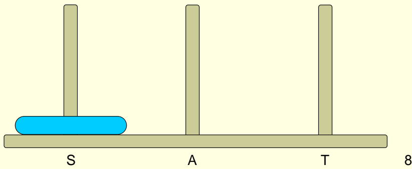
- מותר להעביר רק את הדיסקית העליונה
- אסור להניח דיסקית גדולה על דיסקית קטנה



7

עבור דסקית אחת – קל מאוד!

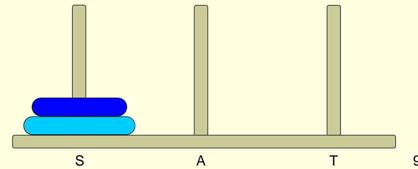
- דסקית תכלת מ-S ל-T



8

עבור 2 דסקיות – קל מאוד!

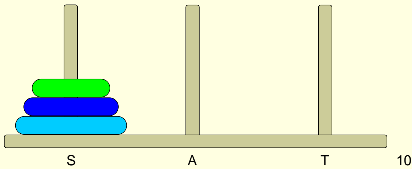
- דסקית כחולה מ-S ל-A
- דסקית תכלת מ-S ל-T
- דסקית כחולה מ-A ל-T



9

עבור 3 דסקיות – קל

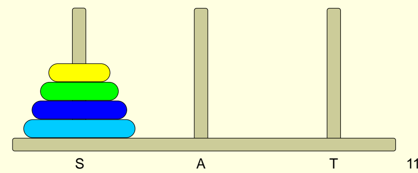
- דסקית ירוקה מ-S ל-T
- דסקית כחולה מ-S ל-A
- דסקית ירוקה מ-T ל-A
- דסקית תכלת מ-S ל-T
- דסקית ירוקה מ-A ל-S
- דסקית כחולה מ-A ל-T
- דסקית ירוקה מ-S ל-T



10

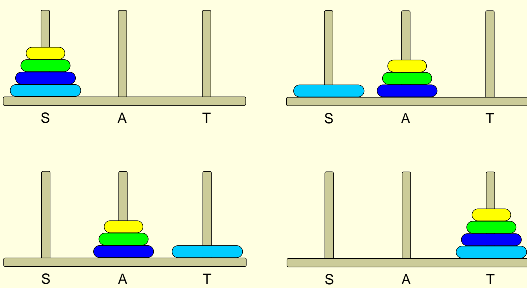
עבור 4 דסקיות?

- ראינו שיש פתרון עבור 3 דסקיות
- אפשר להשתמש בו על מנת לפתור עבור 4!



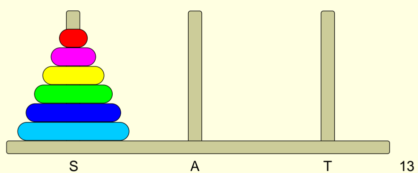
11

פתרון עבור 4 דסקיות



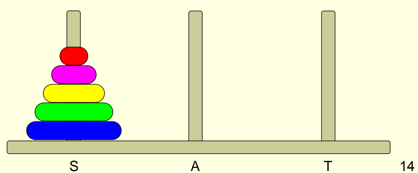
12

עבור n דסקיות?



13

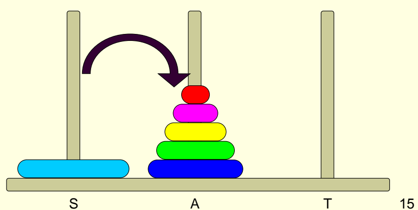
נניח שיש לי פתרון עבור n-1 דסקיות



14

העברת n דיסקיות מ-S ל-T

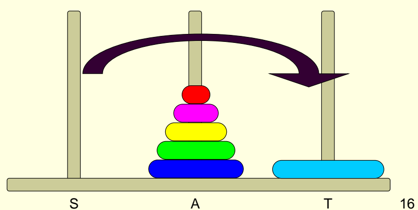
1. העברת n-1 הדיסקיות הקטנות מ-S ל-A, בעזרת יתד עזר T



15

העברת n דיסקיות מ-S ל-T (המשך)

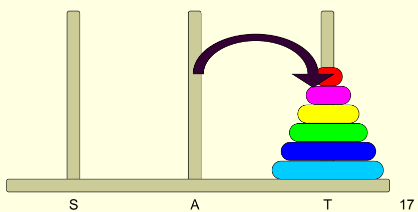
2. העברת הדיסקית שנותרה מ-S ל-T



16

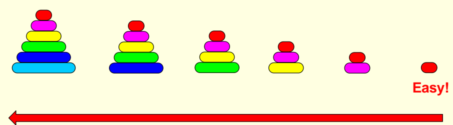
העברת n דיסקיות מ-S ל-T (המשך)

3. העברת n-1 הדיסקיות העליונות מ-A ל-T, בעזרת יתד עזר S



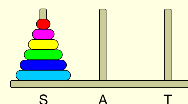
17

ומה עם פתרון עבור n-1 דסקיות?



18

אלגוריתם – עבור n דסקיות



1. אם צריך להעביר דסקית אחת – קל!
2. העבר n-1 דסקיות מ-S ל-A (יתד עזר T)
3. העבר דסקית מ-S ל-T
4. העבר n-1 דסקיות מ-A ל-T (יתד עזר S)

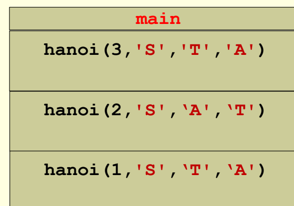
נתרגם זאת לשפת C...

תוכנית C לפתירת מגדלי הנוי

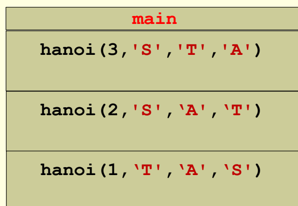
```
void hanoi(char diskNum, char s, char t, char a);

int main(){
        hanoi(3,'S','T','A');
        return 0;
}

void hanoi(char diskNum, char s, char t, char a){
        if (diskNum == 1){
                printf("Move disk from %c to %c\n", s, t);
                return;
        }
        hanoi(diskNum-1,s,a,t);
        printf("Move disk from %c to %c\n", s, t);
        hanoi(diskNum-1,a,t,s);
}
```

1. אם צריך להעביר דסקית אחת – קל!
2. העבר n-1 דסקיות מ-S ל-A (יתד עזר T)
3. העבר דסקית מ-S ל-T
4. העבר n-1 דסקיות מ-A ל-T (יתד עזר S)

זוכרים את מחסנית הקריאות?



זוכרים את מחסנית הקריאות?



וכן הלאה...

רקורסיה

- פתרנו את בעיית מגדלי הנוי בעזרת **רקורסיה**
  - כלומר בעזרת פונקציה שקוראת לעצמה.
- רקורסיה מאפשרת לנו לפתור בעיה "גדולה" בעזרת פתרון של בעיות "קטנות" המרכיבות אותה.
- בכל **קריאה רקורסיבית** אנחנו "מקטינים" את הבעיה ולבסוף מגיעים למקרה קצה שאותו קל לפתור באופן ישיר.

דוגמא: הגדרת נוסחאות

- דרך מקובלת להגדיר נוסחה או פעולה מתמטית היא לרשום את שלבי החישוב שלה:

n! = 1*2*3*….*n

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

- אפשר לחשב את ערך הנוסחה שלב-אחר-שלב (איטרטיבית) למשל:

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

הגדרת נוסחאות

- דרך נוספת להגדיר נוסחאות – הגדרה **רקורסיבית**
מגדירים את הערך של השלב האחרון בעזרת תוצאות השלבים שלפניו.
- במקום להגדיר עצרת על-ידי:
n!=1*2*3*....*n
- אפשר להגדיר על-ידי:

$$\begin{cases} n!=n*(n-1)! \\ 0!=1 \end{cases}$$

ואז החישוב יהיה:

```
4! = 4*3! = 4*(3*2!) = 4*(3*(2*1!)) =
  4*(3*(2*(1*0!))) = 4*(3*(2*1)) = 4*(3*2) =
  4*6 = 24
```

הגדרה רקורסיבית

- מורכבת משני חלקים:
  1. פירוט של שלב אחד בנוסחה
  2. ציון תוצאה עבור ערך התחלתי כלשהו ("בסיס הרקורסיה")
  (אחרת חישוב הנוסחה לא מסתיים)
- דוגמא נוספת:
הגדרה איטרטיבית:
$a^n = a*a*....*a$
הגדרה רקורסיבית:

$$\begin{cases} a^n = a*a^{n-1} \\ a^0 = 1 \end{cases}$$

ואז:

$4^3 = 4*4^2 = 4*(4*4^1) = 4*(4*(4*4^0)) =$
$= 4*(4*(4*1)) = 4*(4*4) = 4*16 = 64$

יתרונות

- קצר
  - בהרבה מקרים, ההגדרה הרקורסיבית קצרה בהרבה מהאיטרטיבית
- נוח
  - במקרים מסוימים, ההגדרה הרקורסיבית היא ההגדרה הטבעית והנוחה ביותר של מה שרוצים לחשב

דוגמא - סדרת פיבונאצ'י

- איברי הסדרה

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...
```

  - שני האיברים הראשונים הם 0 ו-1
  - שאר האיברים מוגדרים כסכום שני האיברים שלפניהם

```
Fib(1) = 0
Fib(2) = 1
Fin(n) = Fin(n-1) + Fib(n-2)
```

- פחות נוח לרשום במקרה הזה הגדרה איטרטיבית (נוסחה מפורשת).
- החישוב עפ"י הנוסחה הרקורסיבית הוא:
Fib(4) = Fib(3)+Fib(2) = (Fib(2)+Fib(1))+Fib(2) = (1 + Fib(1)) + Fib(2) = … = 2

חישוב נוסחאות רקורסיביות בשפת C

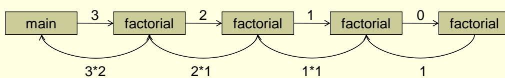
- עצרת

$$\begin{cases} n! = n*(n-1)! \\ 0! = 1 \end{cases}$$

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

דוגמת הרצה

```
int main()
{
    printf("%d\n", factorial(3));

    return 0;
}
```



דוגמת הרצה

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

int main()
{
    printf("%d\n", factorial(3));

    return 0;
}
```

main		

דוגמת הרצה

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

int main()
{
    printf("%d\n", factorial(3));

    return 0;
}
```

fac	n	3
main		

דוגמת הרצה

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

int main()
{
    printf("%d\n", factorial(3));

    return 0;
}
```

fac	n	2
fac	n	3
main		

דוגמת הרצה

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

int main()
{
    printf("%d\n", factorial(3));

    return 0;
}
```

fac	n	1
fac	n	2
fac	n	3
main		

בסיס הרקורסיה

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

int main()
{
    printf("%d\n", factorial(3));

    return 0;
}
```

1

fac	n	0
fac	n	1
fac	n	2
fac	n	3
main		

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

int main()
{
    printf("%d\n", factorial(3));

    return 0;
}
```

1

fac	n	1
fac	n	2
fac	n	3
main		

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

int main()
{
    printf("%d\n", factorial(3));

    return 0;
}
```

fac	n	2
fac	n	3
main		

2

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

int main()
{
    printf("%d\n", factorial(3));

    return 0;
}
```

fac	n	3
main		

6

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

int main()
{
    printf("%d\n", factorial(3));

    return 0;
}
```

main		

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

int main()
{
    printf("%d\n", factorial(3));

    return 0;
}
```

printf		6
main		

נקודות לתשומת-לב

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

ללא תנאי העצירה (בסיס הרקורסיה),
החישוב לא יסתיים לעולם

בכל פונקציה רקורסיבית חייב להיות תנאי עצירה:
מקרה בו הפונקציה תחזיר ערך מבלי לקרוא לעצמה שוב

נקודות לתשומת-לב

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

התקדמות לעבר תנאי העצירה

כדי שהתוכנית תסתיים, צריך לדאוג לכך שבאיזשהו
שלב תנאי העצירה יתקיים (בדיוק כמו בלולאות)

נקודות לתשומת-לב: זכרון

- כשקוראים לפונקציה מתוך עצמה, משתנים שהוגדרו בפונקציה הקוראת נשארים בזיכרון (בסביבה במחסנית).
  - משום שהמשתנים נשארים בזיכרון עד שהפונקציה מסתיימת.
- הרבה מאוד קריאות רקורסיביות עלולות למלא את הזיכרון של המחשב (דוגמא בהמשך)
- גם אם נחסוך במשתנים, נדרש מקום בכל קריאה לפונקציה:
  - עבור הנקודה שאליה היא צריכה לחזור
  - עבור הערך שהיא צריכה להחזיר

איך נחשב חזקה באופן רקורסיבי?

- ההגדרה הרקורסיבית (למעריך שלם ואי-שלילי):

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

- והפונקציה ב- C:

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

אפשר לטפל גם במעריך שלילי

איך נחשב חזקה באופן רקורסיבי

- אם יתכן מעריך שלילי, ההגדרה היא:

עבור $n<0$: $a^n = 1/a^{-n}$

עבור $n\geq0$: $a^n = a \cdot a^{n-1}$, $a^0=1$

- והפונקציה ב- C:

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

סדרת פיבונאצ'י

- נוסחה שמוגדרת באופן רקורסיבי:

```
fib(n)=fib(n-1)+fib(n-2)
fib(1)=0, fib(2)=1
```

- נוכל לכתוב את זה כך ב- C:

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

מה ההבדל בין הדוגמא הזאת לשתי הדוגמאות הקודמות שראינו?

חישוב סדרת פיבונאצ'י

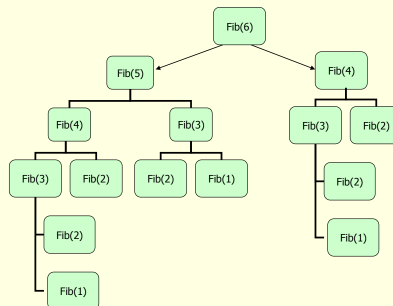
- כל קריאה לפונקציה עם n>2, יוצרת **שתי קריאות** נוספות לפונקציה
- אם גם בהן n>2, כל אחת מהן יוצרת שתי קריאות נוספות
- וכן הלאה.
- כבר עבור n קטן יחסית (למשל 50):
  - נקבל מספר עצום של קריאות לפונקציה

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

קריאות לפונקציה fib

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

קריאות לפונקציה fib



סדרת פיבונאצ'י – בעיית יעילות

- לחישוב האיבר ה-24 בסדרה צריך לחשב את 23 הערכים שלפניו
  - אבל בחישוב הרקורסיבי יתבצעו עשרות אלפי קריאות לפונקציה
  - כלומר יחושבו עשרות אלפי ערכים.
- הסיבה היא ששני הערכים שמחושבים בקריאה הרקורסיבית לא נשמרים
  - לכן ברקורסיה יבוצעו הרבה פעמים את הקריאות fib(1), fib(2) וכו'.
- מסקנה:
  - אם פונקציה רקורסיבית צריכה לקרוא לעצמה יותר מפעם אחת, אז חישובה הוא בעייתי, ויתאפשר רק למספרים קטנים.
  - עבור מספרים גדולים נצטרך לחשב ולהגדיר איטרטיבית.

סדרת פיבונאצ'י - איטרטיבי

```
int fib(int n)
{
   int i, next, fib1 = 1, fib2 = 1;

   if (n == 1  ||  n == 2)
      return n-1;

   for (i = 3; i <= n; i++)
   {
      next = fib1 + fib2;
      fib1 = fib2;
      fib2 = next;
   }

   return fib2;
}
```

שימוש ברקורסיה

- מצד אחד:
  - לפעמים לשימוש ברקורסיה יש יתרון - קל יותר לכתוב באמצעותו את החישוב
- מצד שני:
  - לא תמיד קל למצוא הגדרה רקורסיבית לפונקציה
  - לא תמיד קל להבין תוכניות שנכתבו באופן רקורסיבי
- לכן נבחר להשתמש ברקורסיה במקרים שבאמת נוחים לכך.