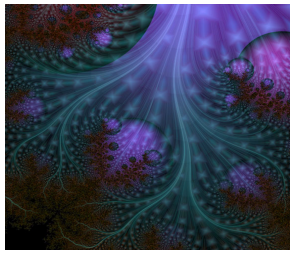
פרק 13
רקורסיה

רקורסיה

- **רקורסיה** הינה שיטה לתכנון אלגוריתמים, שבה הפתרון לקלט כלשהו מתקבל על ידי פתרון **אותה הבעיה בדיוק** על קלט פשוט יותר, והרחבת פתרון זה לאחר מכן לפתרון עבור הקלט המורכב.
- באופן מעשי, **פונקציה רקורסיבית** הינה פונקציה **f()** שתוך כדי ריצתה גורמת לקריאה חוזרת לעצמה.



- ברקורסיה ישירה הפונקציה **f()** מבצעת קריאה מפורשת לעצמה.
- ברקורסיה עקיפה הפונקציה **f()** קוראת לפונקציה אחרת **g()**, וזו קוראת בשלב מאוחר יותר ל-**f()**.

רקורסיה

אלגוריתם רקורסיבי מורכב משני אלמנטים:

- צעד המעבר: זהו האופן בו מפרקים את הבעיה הנתונה לתתי-בעיות זהות אך פשוטות יותר, ולאחר מכן מרחיבים את הפתרונות שלהן לפתרון מלא של הבעיה המקורית.
- מקרה הבסיס: זהו אופן הפתרון של הבעיה (או הבעיות) בקצה השרשרת הרקורסיבית; בעיות אלו נחשבות הבעיות "הכי פשוטות", ואותן לא מפרקים יותר.

וידוא נכונות של אלגוריתם רקורסיבי דורש:

- ✓ וידוא הנכונות של שני חלקי הרקורסיה (צעד המעבר ומקרה הבסיס).
- ✓ וידוא שכל שרשרת קריאות רקורסיביות אכן מסתיימת באחד ממקרי הבסיס.

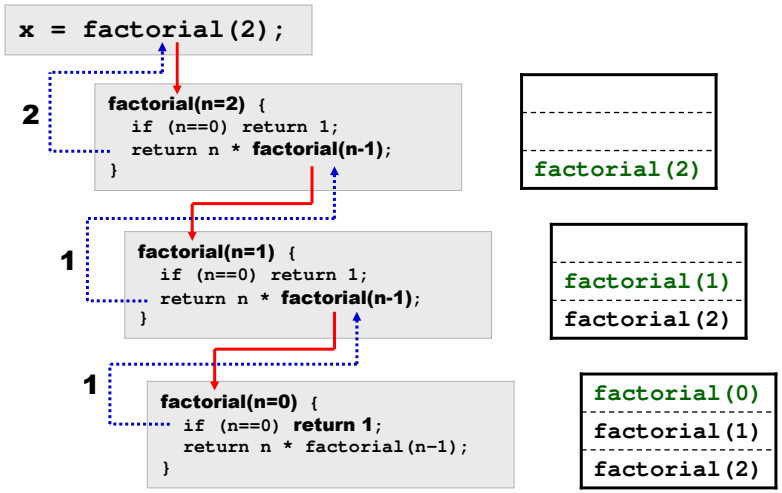
רקורסיות פשוטות: חישוב עצרת

- חישוב פעולת העצרת **n!** באמצעות רקורסיה:

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

- מקרה הבסיס הוא **n==0**, שעבורו התוצאה ידועה.
- צעד המעבר מצמצם את הבעיה מ-**n** ל- **n-1** , באמצעות הנוסחה **n! = n · (n-1)!**.
- קל לוודא שמכל **n** שמתחילים, תמיד מגיעים בסוף השרשרת למקרה הבסיס **n==0**, כיוון שאנו מקטינים בכל קריאה את **n** ב-1.

תמונת המחסנית ב- factorial()



רקורסיות פשוטות: פיבונאצ'י

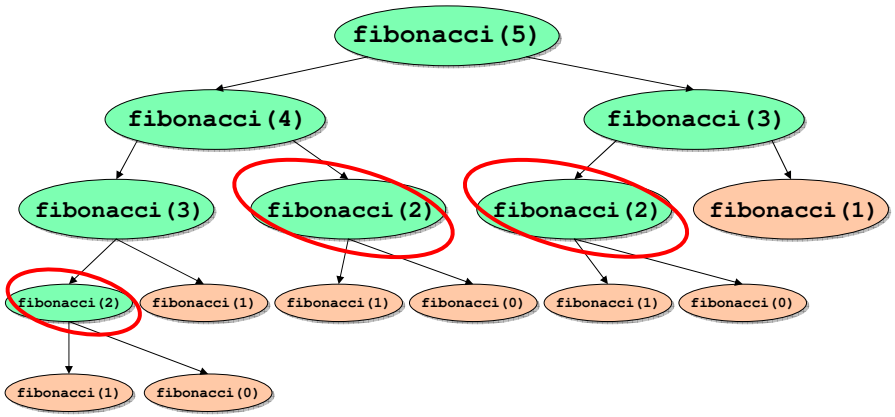
- חישוב סדרת פיבונאצ'י באמצעות רקורסיה:

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

- במקרה זה צעד המעבר מורכב משתי קריאות רקורסיביות.
- שימו לב שדרושים לנו כאן שני מקרי בסיס (עבור 0 וגם עבור 1), כיוון שהצעד הרקורסיבי שכתבנו מוגדר רק ל-**n**-ים החל מ-2.
- כיוון שצעד הרקורסיה כאן איננו פשוט כמו במקרה של עצרת, עלינו לוודא בקפדנות שלכל **n** שנתחיל ממנו, השרשרת אכן תסתיים תמיד באחד ממקרי הבסיס.

תמונת הקריאות ב- fibonacci()

- הגרף הבא מתאר את שרשרת הקריאות הרקורסיביות שמתבצעות עבור **n=5**. כפי שניתן לראות, מספר הקריאות גדול מאוד ונעשים חישובים כפולים רבים.



מספר הקריאות ב- fibonacci()

- בגלל העבודה הכפולה, מספר הקריאות הרקורסיביות בפונקציה זו גדל **מהר מאוד** ביחס ל-**n**. שימו לב למספר הקריאות המבוצעות עבור **n**-ים שונים!

n	מספר הקריאות הרקורסיביות	זמן החישוב (מחשב P-IV)
10	177	0.0 שניות
20	21,891	0.0 שניות
30	2,692,537	0.11 שניות
40	331,160,281	13.8 שניות
50	40,730,022,147	1,864 שניות = 31.1 דקות
55	451,702,867,433	20,685 שניות = 5.7 שעות
60	5,009,461,563,921	229,412 שניות = 2.6 ימים

- מסקנה: בדוגמה זו, למרות הבהירות והפשטות של הקוד שכתבנו, לא ניתן להשתמש ברקורסיה כפי שמימשנו אותה, בגלל חוסר יעילות.

רקורסיות פשוטות: gcd()

- גם את **gcd()** ניתן לכתוב בצורה קומפקטית בעזרת רקורסיה:

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

- במקרה זה לרקורסיה יתרון ברור: היא (כמעט) לא פחות יעילה מהגרסה עם הלולאה, והקוד הרבה יותר ברור.
- דוגמאות נוספות לרקורסיות פשוטות (נסו בעצמכם!): חישוב החזקה x^n עבור n טבעי, חישוב סכום של סדרה חשבונית והנדסית, מציאת איבר מקסימאלי במערך וכן הלאה.
- בכל הדוגמאות הללו בדרך כלל קל לכתוב את הפונקציה גם ללא רקורסיה, תוך שימוש בלולאה.

Bubble Sort – מימוש רקורסיבי

- נתון מערך **a[n]** שברצוננו למיין. נוכל לכתוב פונקציה רקורסיבית שממשת bubble sort באופן פשוט למדי:

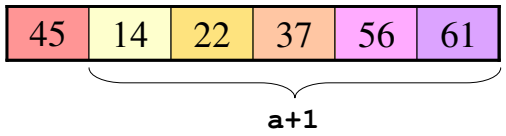
```
void bubble_sort(int a[], int n)
{
  int i=0;
  if (n <= 1) return;

  bubble_sort(a+1, n-1);
  while(a[i] > a[i+1]) {
    swap(a+i, a+i+1);
    if ((++i) == n-1) break;
  }
}
```

- הערה: הפונקציה **swap()** מקבלת שני מצביעים ומחליפה את תוכנם.

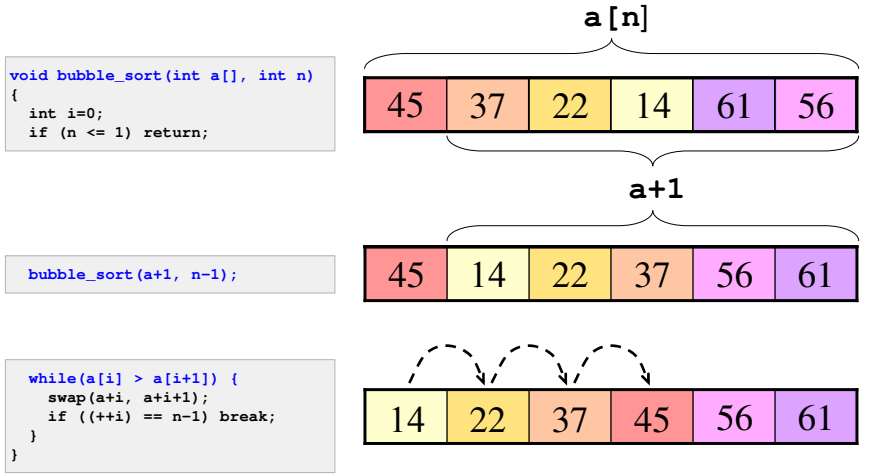
Bubble Sort – מימוש רקורסיבי

ניסוח האלגוריתם, בגרסתו הרקורסיבית:

1. <u>מקרה הבסיס</u>: עבור מערך בגודל 1 – אין צורך לבצע דבר.
2. <u>צעד המעבר, עבור מערך בגודל n>1</u>:
   - מיינים רקורסיבית את המערך החל מהמקום השני בו.
   - "מבעבעים" את האיבר הראשון למקומו הנכון.



Bubble Sort – מימוש רקורסיבי



ניתוח תהליך הרקורסיה

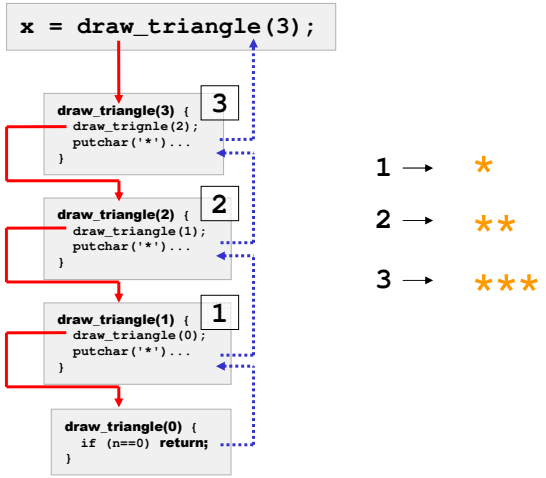
- נתכנת פונקציה שמציירת משולש בגובה **n** שורות:

```
*
**
***
****
*****
******
*******
```

- האלגוריתם הרקורסיבי יהיה זה:
  1. הדפס משולש בגובה **n−1**.
  2. הוסף שורת כוכביות באורך **n**.

```
void draw_triangle(int n)
{
  int i;
  if (n <= 0) return;
  draw_triangle(n-1);
  for (i=0; i<n; ++i) putchar('*');
  putchar('\n');
}
```


מפת הקריאות בהדפסת משולש



ניתוח תהליך הרקורסיה

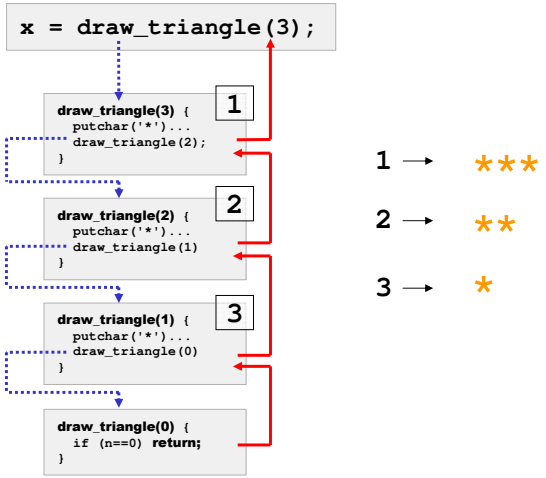
- ומה אם היינו רוצים את המשולש הפוך ?

```
*******
******
*****
****
***
**
*
```

- נשתמש באלגוריתם הרקורסיבי הבא:
  1. הדפס שורת כוכביות באורך **n**.
  2. הדפס משולש הפוך בגובה **n−1**.

```
void draw_triangle(int n)
{
  int i;
  if (n <= 0) return;
  for (i=0; i<n; ++i) putchar('*');
  putchar('\n');
  draw_triangle(n-1);
}
```


מפת הקריאות בהדפסת משולש הפוך



ניתוח תהליך הרקורסיה

- שתי התוכניות מדגימות את המקרה הפשוט ביותר של רקורסיה – **רקורסיה ליניארית**. בסוג זה של רקורסיה הפונקציה הרקורסיבית קוראת לעצמה פעם אחת ויחידה, וכך נוצרת שרשרת ליניארית של קריאות רקורסיביות לעומק.
- עם זאת, יש הבדל עקרוני אחד בין שתי התוכניות:
- בתוכנית הראשונה מתבצעת ראשית קריאה רקורסיבית לעומק, שבה לא מתבצע דבר למעשה. כל פעולת הציור מתבצעת בשלב ה"גלגול לאחור" של הרקורסיה.
- בתוכנית השנייה כל פעולת הציור נעשית בזמן "פרישת" הרקורסיה, כלומר תוך כדי הכניסה אליה. בשלב העלייה חזרה לא מתבצע דבר.

ניתוח תהליך הרקורסיה

- ברקורסיה ליניארית מבחינים לפיכך בין שני חלקים של הפונקציה:
- החלק **שלפני** הקריאה הרקורסיבית מתבצע תוך כדי פרישת הרקורסיה. זהו "ראש" הרקורסיה.
- החלק **שאחרי** הקריאה הרקורסיבית מתבצע בזמן הגלגול בחזרה של הרקורסיה. זהו "זנב" הרקורסיה.
- מה יקרה אם נשלב בין שני החלקים, כלומר נכתוב פונקציה שבה מציירים שורת כוכביות גם לפני וגם אחרי הקריאה הרקורסיבית?

ניתוח תהליך הרקורסיה

- בדוגמה הבאה מופיע הקוד של הפונקציה המשולבת. התוצאה היא מעין דגלון העשוי משני משולשים הפוכים.

```
void draw_flag(int n)
{
  int i;

  if (n <= 0) return;

  for (i=0; i<n; ++i) putchar('*');
  putchar('\n');

  draw_flag(n-1);

  for (i=0; i<n; ++i) putchar('*');
  putchar('\n');
}
```

קוד מקדים

קוד מאסף

ניתוח תהליך הרקורסיה

- אנו יכולים לתאר את הפונקציה שכתבנו במונחים של רקורסיה סטנדרטית. למעשה, האלגוריתם הרקורסיבי שמימשנו הוא:
  1. הדפס שורת כוכביות באורך **n**.
  2. צייר דגלון בגודל **n−1**.
  3. הדפס שורת כוכביות באורך **n**.
- הפלט של הפונקציה נראה כך:

```
*****
****
***
**
*
*
**
***
****
*****
```

- נסו בעצמכם! פונקציה רקורסיבית שמציירת פירמידה מכוכביות:

```
  *
 ***
*****
```

ניתוח תהליך הרקורסיה

- רק בשביל השעשוע... נשפר את מראה הדגלון בכך שנצייר את המשולש העליון רחב יותר, ואת המשולש התחתון קצר יותר:

```
void draw_nice_flag(int n)
{
  int i;
  if (n <= 0) return;
  for (i=0; i<n; ++i) printf("****");
  putchar('\n');
  draw_nice_flag(n-1);
  if (n % 2) return;
  for (i=0; i<n; ++i) printf("*");
  putchar('\n');
}
```

Run!

המשולש רחב פי 4

המשולש קצר פי 2

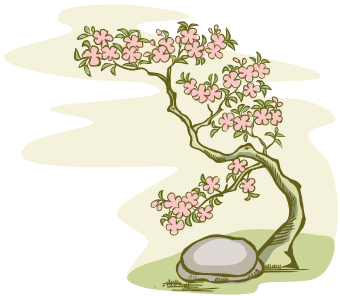
ניתוח תהליך הרקורסיה

- עבור n=10, הפלט של פונקציה זו נראה כך:

```
****************************************
********************************
****************************
************************
********************
****************
************
********
****
**
****
******
********
**********
```


עץ הקריאות של פונקציה רקורסיבית

- על מנת לחקור התנהגות של פונקציה רקורסיבית, נוח להתבונן **בעץ הקריאות** שלה.
- עץ הקריאות של פונקציה רקורסיבית הוא עץ שבו כל קודקוד מייצג את אחת הקריאות לפונקציה. השורש הוא הקריאה הראשונה לפונקציה (כלומר כשקוראים לפונקציה מחוץ לה), והבנים של כל קודקוד הם הקריאות הרקורסיביות שהפונקציה מבצעת במהלך ריצתה.

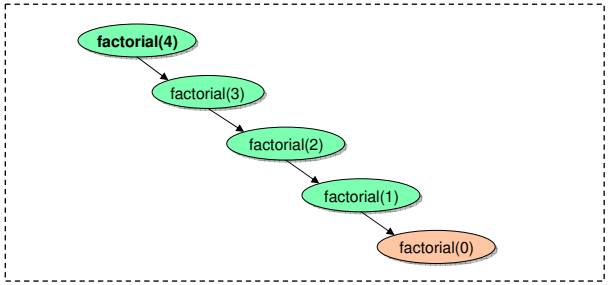


- העלים בעץ הקריאות מתאימים למקרי הבסיס של הרקורסיה, כיוון שהם מייצגים ריצות של הפונקציה שאינן מבצעות קריאות רקורסיביות.

דוגמאות לעצי קריאות

חישוב עצרת:

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

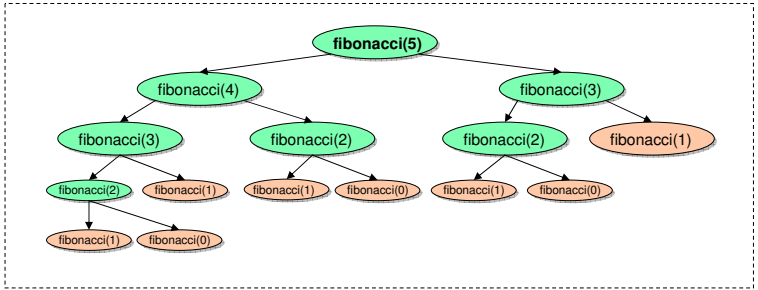


רקורסיה ליניארית!

דוגמאות לעצי קריאות

- פיבונאצ'י:

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



סיבוכיות של אלגוריתמים רקורסיביים

- **סיבוכיות זמן**: קשורה למספר הכולל של קריאות רקורסיביות.
- סיבוכיות הזמן היא סך כל הזמן הדרוש לפונקציה, והוא שווה לסכום הזמן שדורשות כל הקריאות הרקורסיביות יחד.
- במקרה הפשוט, כל קריאה רקורסיבית מתבצעת בזמן קבוע $\Theta(1)$. במקרה זה הזמן הכולל הוא פשוט (מס' הקריאות הרקורסיביות) Θ.
- במקרה הכללי צריך להתבונן בעץ הקריאות של הפונקציה הרקורסיבית ולסכום את הזמנים הדרושים לכל הקריאות בעץ.

סיבוכיות של אלגוריתמים רקורסיביים

- **סיבוכיות מקום**: קשורה לעומק המקסימאלי של הרקורסיה (המספר המקסימאלי של קריאות רקורסיביות שמתקיימות בו זמנית על המחסנית).
- סיבוכיות המקום היא כמות הזיכרון המקסימאלית שהפונקציה צורכת במהלך ריצתה. כל כניסה רקורסיבית דורשת הקצאת מקום נוסף במחסנית, ואילו כל יציאה מהרקורסיה מפנה זיכרון זה. לכן כמות הזיכרון המקסימאלית שדורשת הפונקציה מתקבלת בדרך כלל כאשר אנו נמצאים בעומק המקסימאלי של הרקורסיה.
- במקרה הפשוט, כל קריאה רקורסיבית דורשת זיכרון קבוע $\Theta(1)$. במקרה זה הזיכרון הכולל הוא פשוט (עומק הרקורסיה המקסימאלי)Θ.
- במקרה הכללי יש לבחון את עץ הקריאות, למצוא את הקריאה בעומק המקסימאלי, ולסכום את כמות הזיכרון שתופשות כל הקריאות הרקורסיביות מהשורש ועד עליה.

סיבוכיות של פיבונאצ'י

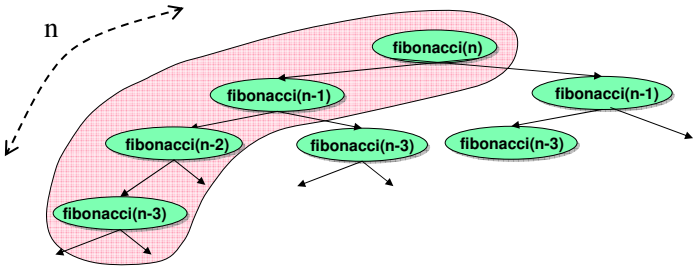
- **סיבוכיות זמן**: כל קריאה לפונקציה דורשת מספר קבוע של פעולות, ולכן זמן הריצה הכולל הוא פשוט (מספר הקריאות הרקורסיביות)Θ, שזה גם (מספר הקודקודים בעץ הקריאות)Θ.
- **סיבוכיות מקום**: כל קריאה רקורסיבית צורכת כמות קבועה של זיכרון. לכן, השלב בו הכי הרבה זיכרון תפוש מתקבל כאשר אנו נמצאים בעומק המקסימאלי של הרקורסיה – במצב זה יש הכי הרבה קריאות על המחסנית.

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

סיבוכיות של פיבונאצ'י

סיבוכיות מקום:

אם נתבונן בעץ הקריאות של הפונקציה, נבחין שהמסלול הארוך ביותר בעץ מגיע לעומק n. לפיכך, המספר המקסימאלי של קריאות רקורסיביות שמתקיימות בו-זמנית על המחסנית הוא n, וכיוון שכל קריאה רקורסיבית כזו תופשת כמות קבועה של זיכרון, אנו מקבלים שסיבוכיות הזיכרון הינה $\Theta(n)$.



סיבוכיות של פיבונאצ'י

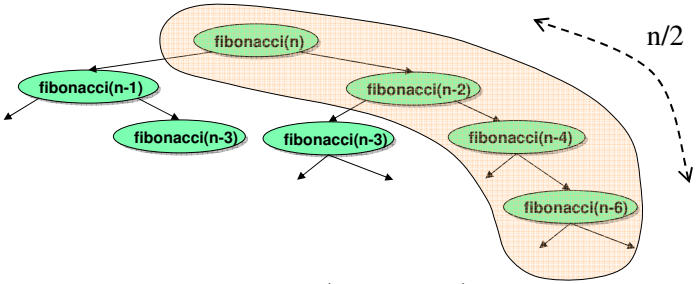
סיבוכיות זמן:

על מנת לקבל את סיבוכיות הזמן של פיבונאצ'י, עלינו לדעת את מספר הקודקודים בעץ הקריאות. הבעיה היא, שקשה לחשב את המספר המדויק של הקודקודים בעץ הזה. במקום זאת, נקבל חסם עליון וחסם תחתון על מספר הקודקודים בעץ.

חסם עליון: אנו יודעים שהעומק המקסימאלי של העץ הוא n. כעת, בעץ **מלא** בעומק n יש 2^n-1 קודקודים, ואילו העץ שלנו איננו מלא ולכן יש בו **לכל היותר** 2^n-1 קודקודים. מכאן שמספר הקריאות הרקורסיביות הוא לכל היותר 2^n-1, ולכן זמן הריצה חסום מלמעלה על ידי $T(n)=O(2^n)$.

סיבוכיות של פיבונאצ'י

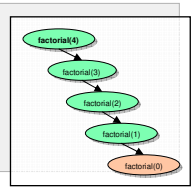
חסם תחתון: אם נתבונן שוב בעץ הקריאות, נראה שהעומק **המינימאלי** של העץ הוא n/2. עתה, בעץ מלא בעומק n/2 יש $2^{n/2}-1$ קודקודים, ואילו העץ שלנו מכיל בתוכו את כל העץ הזה, ולכן יש בו **לפחות** $2^{n/2}-1$ קודקודים. מכאן שמספר הקריאות הרקורסיביות הוא לפחות $2^{n/2}-1$, וזמן הריצה חסום מלמטה על ידי $T(n)=\Omega(2^{n/2})$.



סיבוכיות של `factorial()`

- פונקצית העצרת הינה רקורסיה ליניארית. נשים לב שכל קריאה רקורסיבית מקטינה את n באחת, ולכן עומק הרקורסיה הוא $\Theta(n)$.

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



- זמן ריצה: כל קריאה רקורסיבית דורשת מספר קבוע של פעולות, ומתבצעות סה"כ n קריאות רקורסיביות, ולכן זמן הריצה הוא $\Theta(n)$.
- זיכרון: כל קריאה רקורסיבית דורשת זיכרון בגודל קבוע (שאיננו תלוי ב-n), והמספר המקסימאלי של קריאות רקורסיביות שמתקיימות בו-זמנית הוא n. לכן סיבוכיות הזיכרון היא $\Theta(n)$ גם כן.

סיבוכיות של חיפוש בינארי

- חיפוש בינארי ניתן למימוש כרקורסיה ליניארית. בכל קריאה רקורסיבית מקטינים את תחום החיפוש פי 2 על פי האיבר האמצעי.

```
int binsearch(int a[], int n, int x)
{
  if (n <= 0) return -1;
  if (a[n/2] == x) return n/2;
  if (a[n/2] >  x)
    return binsearch(a,n/2,x);
  else {
    int pos = binsearch(a+n/2+1, n-n/2-1, x);
    if (pos == -1) return -1;
    return pos + n/2 + 1;
  }
}
```


סיבוכיות של חיפוש בינארי

- במקרה של חיפוש בינארי, כל קריאה רקורסיבית מקטינה את n פי 2, ולכן עומק הרקורסיה הוא $\Theta(\log(n))$.

- <u>זמן ריצה</u>: בכל קריאה רקורסיבית מבוצע מספר קבוע של פעולות, ולכן זמן הריצה הוא $\Theta(\log(n))$.
- <u>דרישות זיכרון</u>: כל קריאה רקורסיבית צורכת זיכרון בגודל קבוע, ולכן סיבוכיות הזיכרון אף היא $\Theta(\log(n))$.

ניתוח סיבוכיות באמצעות פיתוח טלסקופי

- שיטת הפיתוח הטלסקופי הינה שיטה נוחה לניתוח הסיבוכיות של רקורסיות רבות.
- באופן כללי, טכניקת הפיתוח הטלסקופי מאפשרת לנתח זמני ריצה על ידי סכימה שיטתית של זמני הריצה של כל הקריאות הרקורסיביות שמתבצעות. ניתן להשתמש גם בווריאציה של השיטה לשם ניתוח סיבוכיות מקום, אך זה לרוב פחות נוח (אנו נראה דוגמה אחת לכך).
- הטכניקה תוסבר דרך מספר דוגמאות. אנו נתחיל מהמקרה הפשוט ביותר: ננתח (שוב...) את פונקצית העצרת. שימו לב לשלבי העבודה – הם יחזרו על עצמם בהמשך.

פיתוח טלסקופי של `factorial()`

1. נתחיל מכתיבת ביטוי **לא מפורש** ל-$T(n)$, כזה שמסתמך על ידיעת ערכו של T עבור n-ים קטנים יותר. את הביטוי ניתן לקבל ישירות מהקוד של הפונקציה הרקורסיבית:

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

- במקרה שלנו, נקבל מקוד הפונקציה את הביטוי הבא, שמתאר את $T(n)$ באופן לא מפורש (כאשר C הוא קבוע המציין את מספר הפעולות שמתבצעות בפונקציה ללא הקריאה הרקורסיבית):

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

פיתוח טלסקופי של factorial()

2. כעת, נציב n-1 במקום n בביטוי שרשמנו. נקבל כי זמן הריצה עבור (n-1) מקיים:

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

- ערך זה ניתן להציב בחזרה בביטוי ל-T(n). נקבל:

$$\begin{aligned} T(n) &= T(n-1) + C = \\ &= \big(T(n-2) + C\big) + C = \\ &= T(n-2) + 2C \end{aligned}$$

- באופן דומה נוכל להמשיך ולהציב עבור n-2, n-3 וכן הלאה.

פיתוח טלסקופי של factorial()

3. נמשיך כך, ונקבל את **הפיתוח הטלסקופי** של T(n):

$$\begin{aligned} T(n) &= T(n-1) + C = \\ &= \big(T(n-2) + C\big) + C = \\ &= T(n-2) + 2C = \\ &= T(n-3) + 3C = \\ &\vdots \\ &= T(n-k) + k \cdot C \end{aligned}$$

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

כאשר השורה התחתונה היא השורה ה-k בפיתוח הטלסקופי.

פיתוח טלסקופי של factorial()

4. שרשרת הפיתוח נעצרת כאשר מגיעים למקרה הבסיס. שימו לב שעבור מקרה זה, הביטוי שרשמנו בתחילה ל- T(n) **איננו תקף**.

- במקרה של **`factorial()`**, מקרה הבסיס הוא n=0, ולכן כאשר נגיע ל-T(0) בפיתוח נעצור. ובכן, מתי נגיע ל-T(0) בפיתוח הטלסקופי? לשם כך נתבונן בביטוי שחישבנו עבור השורה ה-k בפיתוח:

$$T(n) = T(n-k) + k \cdot C$$

- אנו רואים כי עבור ההצבה $k=n$ (כלומר בשורה ה-n של הפיתוח) נקבל בדיוק T(0). נציב אם כך $k=n$ בביטוי זה ונקבל:

$$T(n) = T(0) + n \cdot C$$

פיתוח טלסקופי של factorial()

5. זהו, סיימנו! T(0) הוא **מספר קבוע**, כיוון שהוא איננו תלוי ב-n. לכן נוכל להחליפו בקבוע C_1 (זהו למעשה מספר הפעולות הדרושות עבור הקלט n=0), ואנו מקבלים את הביטוי המפורש ל-T(n) :

$$\begin{aligned} T(n) &= T(0) + n \cdot C = \\ &= C_1 + n \cdot C_2 = \\ &= \Theta(n) \end{aligned}$$

פיתוח טלסקופי של Binary Search

- נראה כעת כיצד ניתן לקבל את זמן הריצה של binary search גם כן באמצעות פיתוח טלסקופי.

1. בכל איטרציה של הפונקציה הרקורסיבית, מתבצע מספר קבוע C של פעולות, וכן מתבצעת קריאה רקורסיבית עם קלט בגודל n/2. לפיכך, הביטוי הלא מפורש ל-T(n) הוא:

$$T(n) = T\left(\frac{n}{2}\right) + C$$

2. הצבת n/2 בביטוי זה נותנת:

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + C$$

פיתוח טלסקופי של Binary Search

3. נמשיך כך ונקבל את הפיתוח הטלסקופי של T(n). אנו רוצים לקבל ביטוי לשורה ה-k בפיתוח, ולכן נרשום:

$$\begin{aligned} T(n) &= T\left(\tfrac{n}{2}\right) + C = \\ &= \left(T\left(\tfrac{n}{4}\right) + C\right) + C = \\ &= T\left(\tfrac{n}{4}\right) + 2C = \\ &= T\left(\tfrac{n}{8}\right) + 3C = \\ &\vdots \\ &= T\left(\tfrac{n}{2^k}\right) + k \cdot C \end{aligned}$$

פיתוח טלסקופי של Binary Search

4. במקרה שלנו מקרה הבסיס הוא עבור מערך בגודל 1.

- מתי נגיע ל-T(1) ?
- ובכן, ניתן לראות שזה מתרחש כאשר $n=2^k$, שזה אומר $k=\log(n)$. נציב זאת בביטוי ל-T(n), ונקבל (כפי שרצינו):

$$\begin{aligned} T(n) &= T(1) + \log(n) \cdot C = \\ &= C_1 + \log(n) \cdot C_2 = \\ &= \Theta(\log(n)) \end{aligned}$$

סיבוכיות של Bubble Sort

- ננתח כעת את הסיבוכיות של אלגוריתם המיון bubble sort שפגשנו בתחילת הפרק:

```
void bubble_sort(int A[], int N)
{
  int i=0;
  if (N <= 1) return;

  bubble_sort(A+1, N-1);
  while(A[i] > A[i+1]) {
    swap(A+i, A+i+1);
    if ((++i) == N-1) break;
  }
}
```

סיבוכיות של Bubble Sort

- כל קריאה רקורסיבית מקטינה את n ב-1, ולכן עומק הרקורסיה במקרה זה הוא $\Theta(n)$.
- <u>סיבוכיות זיכרון</u>: כל קריאה רקורסיבית דורשת זיכרון בגודל קבוע, ולכן סיבוכיות הזיכרון היא $\Theta(n)$.
- <u>זמן ריצה</u>: לשם כך נבצע פיתוח טלסקופי של זמן הריצה.
- נשים לב שבכל איטרציה של הפונקציה הרקורסיבית מתבצעת קריאה רקורסיבית עם מערך בגודל n-1, וכן מתבצעות (במקרה הגרוע) עוד n פעולות החלפה. לכן זמן הריצה מקיים:

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

פיתוח טלסקופי של Bubble Sort

- מהפיתוח הטלסקופי נקבל במפורש את זמן הריצה:

$$\begin{aligned}
T(n) &= T(n-1) + n = \\
&= T(n-2) + (n-1) + n = \\
&= T(n-3) + (n-2) + (n-1) + n = \\
&= T(n-k) + (n-(k-1)) + \cdots + (n-1) + n = \\
&\vdots \\
&= T(0) + 1 + 2 + \cdots + (n-1) + n = \\
&= C_1 + \frac{(n+1)n}{2} = \Theta(n^2)
\end{aligned}$$

ניתוח אלגוריתם Merge Sort

- ננתח כעת את הסיבוכיות של מיון merge sort, בגרסתו הרקורסיבית.
- נזכיר ראשית סקיצה של אלגוריתם merge sort הרקורסיבי:

```
merge_sort(A[N])
{
  if (N <= 1) return;
  allocate tmp[N];

  merge_sort( A[0..N/2] );
  merge_sort( A[N/2+1..N-1] );

  tmp = merge( A[0..N/2], A[N/2+1..N-1] );
  memcpy(A ← tmp);
}
```


סיבוכיות הזמן של Merge Sort

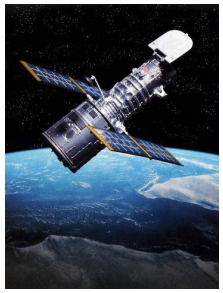
- נשים לב שבכל פעם שנכנסים לתוך קריאה רקורסיבית, אורך המערך קטן פי 2. לכן, העומק המקסימאלי של הרקורסיה הוא $\Theta(\log(n))$.
- <u>זמן ריצה</u>: נבצע כרגיל פיתוח טלסקופי עבור זמן הריצה.
- כפי שניתן להבחין, בכל איטרציה של `merge_sort()` מתבצעות **שתי קריאות** רקורסיביות, כל אחת עם מערך בגודל n/2. כמו כן מתבצעת גם פעולת merge, שדורשת עוד n פעולות.
- אנו מקבלים את הביטוי הבא:

$$T(n) = 2 \cdot T\left(\tfrac{n}{2}\right) + n$$

פיתוח טלסקופי של Merge Sort

- הפיתוח הטלסקופי של ביטוי זה הינו:

$$\begin{aligned} T(n) &= 2\cdot T\left(\tfrac{n}{2}\right)+n = \\ &= 2\cdot\left(2\cdot T\left(\tfrac{n}{4}\right)+\tfrac{n}{2}\right)+n = \\ &= 4\cdot T\left(\tfrac{n}{4}\right)+2n = \\ &= 8\cdot T\left(\tfrac{n}{8}\right)+3n = \\ &\vdots \\ &= 2^k\cdot T\left(\tfrac{n}{2^k}\right)+k\cdot n = \end{aligned}$$



פיתוח טלסקופי של Merge Sort

- על מנת להגיע למקרה הבסיס, עלינו להציב $2^k=n$, כלומר $k=\log(n)$. נקבל את התוצאה:

$$\begin{aligned} T(n) &= 2^k\cdot T\left(\tfrac{n}{2^k}\right)+k\cdot n = \\ &= n\cdot T(1)+\log(n)\cdot n = \\ &= n\cdot C_1+n\log(n) = \\ &= \Theta(n\log(n)) \end{aligned}$$

סיבוכיות הזיכרון של Merge Sort

- מה לגבי סיבוכיות הזיכרון של merge sort? ובכן, גם במקרה זה ניתן לבצע פיתוח טלסקופי, אולם הגישה מעט שונה.
- נזכיר שבמקרה של סיבוכיות זיכרון, אנו מעוניינים בכמות הזיכרון המקסימאלית הנדרשת לשם ביצוע הפונקציה.
- במהלך ריצת הפונקציה הרקורסיבית, כמות הזיכרון התפוס גדלה וקטנה כל העת כאשר אנו נכנסים ויוצאים מקריאות רקורסיביות. מה שעלינו לעשות הוא לזהות את השלב ברקורסיה שבו כמות הזיכרון התפוס היא הגדולה ביותר.
- כפי שמייד נראה, הפיתוח הטלסקופי עבור סיבוכיות זיכרון משקף את פעולת המקסימיזציה הזו, על ידי החלפת פעולות החיבור שראינו בחישובי סיבוכיות הזמן עם פעולות מקסימום.

סיבוכיות הזיכרון של Merge Sort

- כל קריאה רקורסיבית דורשת זיכרון להקצאת מערך **tmp** שגודלו n, וכן זיכרון נוסף לצורך ביצוע הקריאות הרקורסיביות.
- נשים לב שלמרות שישנן שתי קריאות רקורסיביות בפונקצית המיון, **הן אינן מתבצעות בו זמנית**, אלא קודם הראשונה מסתיימת – ומפנה את הזיכרון שהיא תפסה, ורק לאחר מכן השנייה מתחילה.
- לכן, הזיכרון המקסימאלי שיהיה תפוס במהלך ריצת הפונקציה הוא זה הדרוש לאחסון **tmp**, ועוד זה שנדרש על ידי הקריאה הרקורסיבית שתופסת יותר זיכרון מבין השתיים שמבוצעות.

סיבוכיות הזיכרון של Merge Sort

- נסמן את סיבוכיות הזיכרון עבור מערך באורך n ב-S(n). לפיכך, כל אחת מן הקריאות הרקורסיביות דורשת זיכרון בגודל S(n/2).
- נקבל את הביטוי הבא עבור S(n):

$$S(n) = n + \max\left(S\left(\tfrac{n}{2}\right), S\left(\tfrac{n}{2}\right)\right)$$

- במקרה שלנו שני הערכים בתוך ה-max שווים, ואנו מקבלים ביטוי פשוט ל-S(n). פיתוח טלסקופי של ביטוי זה נותן (בדקו!):

$$S(n) = n + S\left(\tfrac{n}{2}\right) = \Theta(n)$$

לסיום: היפוך מחרוזת

- נכתוב לסיום פונקציה **strflip()** שמקבלת מחרוזת ומחזירה אותה הפוכה (מהסוף להתחלה), אך מבצעת זאת רקורסיבית. נראה כאן שתי אלטרנטיבות, ונשווה את זמן הריצה שלהן:

```
void strflip1(char *str)
{
  if (*str == 0) return;

  strflip1(str+1);

  while (str[1] != 0) {
    swap(str,str+1);
    str++;
  }
}
```

$$T(n) = T(n-1) + n = \cdots = \Theta(n^2)$$

היפוך מחרוזת: אופציה שנייה

- הפתרון הקודם איננו יעיל במיוחד: הוא דורש זמן ריבועי ב-n, כאשר אנו יודעים שניתן לעשות אותה הפעולה בזמן ליניארי ב-n.
- הפתרון הבא יעיל יותר, ואולם הוא דורש שהפונקציה תקבל **שני מצביעים**: אחד לתו הראשון במחרוזת, והשני לתו האחרון בה (הכוונה לתו שלפני ה-null):

```
void strflip2(char *begin, char *end)
{
  if (begin >= end) return;

  swap(begin, end);
  strflip2(begin+1, end-1);
}
```


פונקציות מעטפת

- בעיה קטנה: הפתרון השני אמנם יעיל יותר, אך חתימת הפונקציה שכתבנו שונה מזו שאנו רוצים!
- ישנה טכניקה סטנדרטית לפתרון סוגיה זו: נכתוב **פונקצית מעטפת** ש"תעטוף" את הפונקציה הרקורסיבית שכתבנו. הרעיון הוא שהפונקציה שהמשתמש קורא לה בפועל תהיה פונקצית המעטפת, והיא תהייה בדיוק עם החתימה אותה אנו רוצים. במעשה, כל תפקידה של פונקציה זו הוא לקרוא לפונקציה הרקורסיבית, כאשר היא מספקת לה את כל הפרמטרים הנוספים אותם אנו רוצים "להסתיר" מהמשתמש.
- על מנת שהמשתמש לא יהיה מודע לכל זאת, ניתן לפונקצית המעטפת את השם **strflip2()**, ואת שם הפונקציה הרקורסיבית שכתבנו נשנה ל-**strflip2_aux()**.

היפוך מחרוזת: האופציה השנייה

- נקבל את צמד הפונקציות הבאות. שימו לב שלמעשה המשתמש יודע רק על קיומה של הפונקציה הראשונה (פונקצית המעטפת):

```
void strflip2(char *str)
{
  strflip2_aux(str, str + strlen(str)-1);
}
```

```
void strflip2_aux(char *begin, char *end)
{
  if (begin >= end) return;

  swap(begin, end);
  strflip2_aux(begin+1, end-1);
}
```

$$T(n) = T(n-2) + C = \cdots = \Theta(n)$$