פרק 14
Backtracking

בעיית הסכום

- נתון: מערך באורך **n** של מספרים שלמים חיוביים, ומספר **x**.
- הבעיה: לבדוק האם יש תת-קבוצה של מספרים במערך שסכומה **x** (ללא שימוש באותו איבר פעמיים).
- לדוגמה, נתון המערך הבא:

5	3	4	4	7

- עבור x=15 התשובה היא חיובית, כיוון ש- 5 + 7 + 3 = 15
- עבור x=16 התשובה היא חיובית, כיוון ש- 5 + 3 + 4 + 4 = 16
- עבור x=17 התשובה שלילית.

בעיית הסכום

- ננסה לתכנן **פתרון רקורסיבי** לבעיה.
- צעד המעבר: נשים לב שניתן להגיע לסכום **x** בשתי צורות:
  - אם משתמשים ב-**a[0]** (זה אפשרי כמובן רק כש- **a[0] ≤ x**), אזי יש פתרון לבעיה אם ניתן להגיע לסכום **x-a[0]** באמצעות יתר **n-1** האיברים במערך (**a[1]**...**a[n-1]**).
  - מצד שני, אם לא משתמשים ב-**a[0]** אזי ייתכן פתרון לבעיה אם ניתן להגיע לסכום המקורי **x** באמצעות **n-1** האיברים **a[1]**...**a[n-1]** בלבד.
- מקרה הבסיס: ישנם שני מקרים גם כן. אם **x==0**, אזי יש פתרון באופן טריוויאלי – שהרי סכום של 0 איברים מהמערך הוא 0. לעומת זאת אם **n==0** (אבל **x≠0**) אז ברור שאין פתרון.

בעיית הסכום

- נקבל את הפונקציה הרקורסיבית הבאה:

```
int subsum1(unsigned int a[], int n,
            unsigned int x)
{
  if (x==0) return 1;
  if (n<=0) return 0;

  if (a[0]<=x && subsum1(a+1,n-1,x-a[0]))
    return 1;

  return subsum1(a+1,n-1,x);
}
```

בעיית הסכום

- נרצה כעת לשפר את הפונקציה על ידי קבלת מעט יותר אינפורמציה: במקום להחזיר 0/1, נרצה שהיא תחזיר את מספר האיברים בקבוצה שסכומה **x**, או -1 אם אין כזו תת-קבוצה.
- מימוש הפונקציה ישתנה כך:

- נניח שהקריאה הרקורסיבית החזירה תשובה שיש בזנב המערך (מגודל **n-1**) תת קבוצה כלשהי בגודל **k** שסכומה **x-a[0]**. במקרה זה, הוספת **a[0]** לתת הקבוצה זו תיתן את הסכום **x**, ולכן אנו נחזיר **k+1** בתור גודל תת-הקבוצה שסכומה **x**.
- אם הקריאה הראשונה לא צלחה, ננסה בקריאה רקורסיבית נוספת לבדוק אם אולי יש בזנב המערך תת קבוצה שסכומה **x** בדיוק. במידה וקריאה זו תצליח נחזיר את גודל תת-הקבוצה שהיא מצאה בתור התשובה, ואם לא, אזי בהכרח אין פתרון ואנו נחזיר -1.

בעיית הסכום

- הפונקציה המעודכנת תראה כך:

```
int subsum2(unsigned int a[], int n,
            unsigned int x)
{
  if (x==0) return 0;
  if (n<=0) return -1;

  if (a[0] <= x) {
    int k = subsum2(a+1,n-1,x-a[0]);
    if (k > -1) return k+1;
  }
  return subsum2(a+1,n-1,x);
}
```


בעיית הסכום

- כעת, נרצה שהפונקציה גם תחזיר את הפתרון עצמו.
- לשם כך נוסיף לפונקציה פרמטר נוסף – מערך פלט **b[]**, שלתוכו היא תכתוב את הפתרון (נניח כרגיל ש-**b[]** הוקצה מחוץ לפונקציה, ושהוא באורך מספק).
- נרצה שבסוף ריצת הפונקציה, המערך **b[]** שהעברנו יכיל את האינדקסים של האיברים ב-**a[]** שהשתתפו בסכימה.
- קוד הפונקציה מופיע בשקף הבא. יש רק בעיה אחת... שימו לב שבמקרה שאנו בוחרים להוסיף את **a[0]** לתת-הקבוצה, עלינו לכתוב את האינדקס שלו למערך **b[]**; ואולם, איננו יודעים את האינדקס האמיתי שלו במערך **a[]** המקורי! לכן, זמנית רשמנו את קטע הקוד המתאים בכתב נטוי, ומייד ניגש לפתור בעיה זו.

בעיית הסכום

```
int subsum3(unsigned int a[], int n,
            unsigned int x, unsigned int b[])
{
  if (x==0) return 0;
  if (n<=0) return -1;

  if (a[0] <= x) {
    int k = subsum3(a+1,n-1,x-a[0],b+1);
    if (k > -1) {
      b[0] = index of a[0] ;
      return k+1;
    }
  }
  return subsum3(a+1,n-1,x,b);
}
```

בעיית הסכום

- על מנת לדעת את האינדקס של **a[0]** במערך המקורי, אין לנו ברירה אלא להוסיף לפונקציה פרמטר נוסף (ואחרון!).
- נוסיף פרמטר עזר בשם **index**, שיציין את האינדקס של **a[0]** במערך המקורי. בכל קריאה רקורסיבית, נעדכן את **index** בהתאם.
- החיסרון בפתרון זה הוא שבקריאה הראשונה לפונקציה, אנו חייבים לקרוא לפונקציה עם הפרמטר **index** שווה ל-0, אחרת הפונקציה תכשל. כיוון שכך, לא היינו רוצים כלל ש-**index** יופיע כפרמטר של הפונקציה! פרמטר זה הוא פרמטר עזר שאנו הוספנו, שאינו מעניין את המשתמש מבחוץ.
- לכן, כפי שראינו ברקורסיות בעבר, נשתמש כאן בפונקציה מעטפת. את שם הפונקציה הרקורסיבית נשנה ל-**subsum3_aux()**, ונעטוף אותה בפונקציה **subsum3()** שתקרא לה עם **index==0**.

בעיית הסכום

```
int subsum3_aux(unsigned int a[], int n,
     unsigned int x, unsigned int b[], int index)
{
  if (x==0) return 0;
  if (n<=0) return -1;

  if (a[0] <= x) {
    int k = subsum3_aux(a+1,n-1,x-a[0],b+1,index+1);
    if (k > -1) {
      b[0] = index;
      return k+1;
    }
  }
  return subsum3_aux(a+1,n-1,x,b,index+1);
}
```


בעיית הסכום

- פונקצית המעטפת תראה כך:

```
int subsum3(unsigned int a[], int n,
            unsigned int x, unsigned int b[])
{
  return subsum3_aux(a,n,x,b,0);
}
```

- נשים לב שפונקציה זו מחזירה פתרון **כלשהו** של הבעיה. במקרה הכללי, כמובן שייתכנו כמה תתי-קבוצות שונות שסכומן הוא **x**; בהמשך נראה דוגמה שבה אנו מחזירים דווקא פתרון אחד מסוים.

בעיית הסכום

- דוגמה לפונקצית **main()** המשתמשת בפונקציה הנ"ל:

```
void main()
{
  unsigned int a[] = {5,3,4,4,7}, b[5], x;
  int i, k;

  printf("Enter x: "); scanf("%u", &x);
  k = subsum3(a,sizeof(a)/sizeof(int),x,b);

  if (k==-1)
    printf("No solution for ");
  for(i=0; i<k; ++i) {
    printf("%u %c ", a[b[i]], i==k-1 ? '=':'+');
  }
  printf("%u\n", x);
}
```

בעיית הסכום

- דוגמאות לריצת הפונקציה:

```
Enter x: 12
5 + 3 + 4 = 12

Enter x: 13
5 + 4 + 4 = 13

Enter x: 16
5 + 3 + 4 + 4 = 16

Enter x: 17
No solution for 17

Enter x: 7
3 + 4 = 7
```


Backtracking

- התוכנית שזה-עתה ראינו מדגימה יישום של טכניקת backtracking לפתרון בעיות. Backtracking הינה טכניקה המשמשת לפתרון מגוון בעיות "קשות" שבהן אין לנו בנמצא אלגוריתם יעיל יותר, ואנו נאלצים לבצע חיפוש מייגע של הפתרון בתוך אוסף רחב של אפשרויות.

- ב-backtracking מצוי בידינו בכל שלב פתרון חלקי של הבעיה, ואנו רוצים לקבוע אם ניתן להרחיב אותו לפתרון מלא שלה או לא.

- הצרה היא שבאופן כללי יש לנו כמה דרכים אפשריות להרחיב את הפתרון, ומראש אין לנו כל דרך לדעת איזו מהן אמנם מובילה לפתרון! (אם בכלל...). לכן, אנו נאלצים לנסות כל אחת מהאפשרויות בזו-אחר-זו, באופן רקורסיבי.

Backtracking

- חלק מרכזי בכל אלגוריתם backtracking הינו **תנאי העצירה**.

- תנאי עצירה אחד הוא ברור – אם הפתרון החלקי שקיבלנו פותר את הבעיה המלאה, הרי אנו יכולים לעצור ולהחזיר פתרון זה.

- תנאי עצירה אחר, אולי חשוב ממנו, הוא המקרה בו אנו מבחינים כי הפתרון החלקי שקיבלנו אינו ניתן להרחבה לפתרון מלא. במקרה זה נרצה לקטוע את הרקורסיה, ולהחזיר תשובה שלילית.

- תנאי העצירה מן הסוג השני הוא חשוב ביותר, כיוון שהוא מסייע בצמצום מרחב החיפוש שלנו, שמלכתחילה הוא עצום ורב. באופן כללי, ככל שנשכיל לזהות טוב יותר פתרונות חלקיים שלא ניתן להמשיך מהם, ונזהה אותם בשלב מוקדם יותר, כך ה-backtracking יעשה יעיל ומהיר יותר.

Backtracking

- בקורס זה עקרונית לא ניתקל במקרים כאלו, ואולם כאן המקום לציין כי באלגוריתמי backtracking מורכבים יותר, ניתן לראות מאזן עדין בין כמות העבודה שאנו משקיעים בזיהוי פתרונות חלקיים "תקועים", ובין כמות העבודה שאנו חוסכים בכך שאנו מזהים את הפתרונות הללו ומונעים חיפוש מיותר.

- ברור שככל שנשקיע יותר עבודה בזיהוי פתרונות חלקיים שגויים, כך נזהה יותר מהמקרים האלו ונקטע את הרקורסיה מוקדם יותר. ואולם, אם נגזים בכך אנו עשויים לשלם על כך מחיר יקר, שכן עלינו לבצע את אוסף הבדיקות הזה בתחילת כל קריאה רקורסיבית על הפתרון החלקי שנתון לנו (מתוך תקווה לזהות שהוא שגוי) וזו עשויה להצטבר אף ליותר עבודה מזו שחסכנו מלכתחילה.

Backtracking

- ננתח כעת את פתרון בעיית הסכום בהקשר של backtracking:
- **פתרון חלקי** במקרה שלנו הוא תת-קבוצה שסכומה קטן מ-**x**.
- **הרחבת הפתרון** נעשית על ידי הוספת איברים לתת-הקבוצה.
- **תנאי עצירה מסוג ראשון** הינו התנאי **x==0**, כיוון שזה מציין שמצאנו פתרון חוקי לבעיה.
- **תנאי עצירה מסוג שני** הינו התנאי **n==0**, כיוון שזה מציין שהבחירות שעשינו היו שגויות ולכן קיבלנו פתרון חלקי שאינו ניתן להרחבה לפתרון מלא. ברור שזה אינו תנאי מחוכם במיוחד, ולמעשה זהו תנאי טריוויאלי מסוג זה.

Backtracking

- חומר למחשבה: היינו יכולים להחליף את תנאי העצירה השני בתנאי חכם יותר; למשל, אפשר לעצור את החיפוש במידה וסכום כל האיברים שנותרו במערך קטן מ-**x** (ברור שבמצב זה הפתרון החלקי שבנינו אינו ניתן להרחבה לפתרון מלא). תנאי זה מהווה למעשה הכללה של התנאי המקורי **n==0**.
- שימו לב שכעת אנו משקיעים עבודה רבה יותר בכל קריאה רקורסיבית לצורך זיהוי פתרונות חלקיים שגויים; ואולם, אנו עשויים לחסוך באופן זה קריאות רקורסיביות מיותרות רבות, ולכן זהו תנאי שכדאי לשקול. על פי רוב לא נעסוק בקורס זה בתנאי עצירה מורכבים שכאלה.

Backtracking

- כפי שראינו בבעיית הסכום, כאשר מתכננים את התוכנית חשוב לשים לב לסוג הפתרון הדרוש. כמה אפשרויות נפוצות הן:
  - בדיקה **האם יש פתרון**, בלא החזרת הפתרון עצמו.
    לדוגמה: בדיקה האם קיימת תת-קבוצה כלשהי שסכומה **x**.
  - מציאת **פתרון כלשהו**, ללא כל העדפה.
    לדוגמה: מציאת תת-קבוצה כלשהי שסכומה **x**.
  - מציאת **כל הפתרונות** לבעיה.
    לדוגמה: מציאת כל תתי הקבוצות שסכומן **x**.
  - מציאת הפתרון **הטוב ביותר** (במובן כלשהו שאנו בוחרים).
    לדוגמה: מציאת תת-הקבוצה הקטנה ביותר שסכומה **x**.

בעיית הצריבה

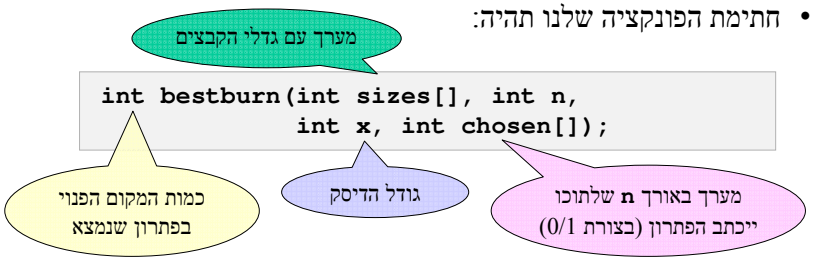
- נתון: אוסף של **n** קבצים בגדלים שונים (גדולים מ-0), ודיסק בגודל **x**.
- הבעיה: למצוא את אוסף הקבצים שגודלו מקסימאלי ונכנס כולו בדיסק.
- לדוגמה, נתון אוסף קבצים בגדלים הבאים:

270	400	220	210	360	540	490

- עבור x=500 התשובה היא 270 + 220 = 490
- עבור x=600 התשובה היא 220 + 360 = 580
- עבור x=700 התשובה היא 270 + 220 + 210 = 700

בעיית הצריבה

- בבעיה זו נציג גישה נפוצה אחרת לייצוג הפתרון: במקום להחזיר את האינדקסים של הקבצים שנבחרו, נחזיר מערך של 0 ו-1, כאשר 1 במקום ה-**i** פירושו שבחרנו את הקובץ ה-**i**, ואילו 0 פירושו שלא בחרנו בקובץ זה.
- נעיר שבאותה מידה היינו יכולים גם להחזיר מערך של אינדקסים; הבחירה במערך של 0/1 מטרתה פשוט להציג גישה שונה.
- חתימת הפונקציה שלנו תהיה:



בעיית הצריבה

- <u>צעד המעבר</u> יהיה כזה:
- ראשית, נניח שהקובץ **sizes[0] לא נצרב**. במקרה זה הפתרון הטוב ביותר הוא פשוט למלא את הדיסק ככל האפשר באמצעות יתר **n–1** הקבצים. אופציה זו מניבה לנו פתרון אפשרי אחד; נסמן את כמות הזיכרון שנותרת פנויה בפתרון זה על ידי **free1**.
- שנית, נניח שהקובץ **sizes[0] כן נצרב** (ניתן להניח זאת רק אם **x≥sizes[0]** כמובן). במקרה זה, הפתרון הטוב ביותר הוא למלא ככל האפשר את יתר הדיסק, שגודלו **x–sizes[0]**, באמצעות יתר **n–1** הקבצים. זה נותן לנו פתרון אפשרי שני – נסמן את כמות הזיכרון הפנוי בפתרון זה על ידי **free2**.
- לאחר שבידינו שני הפתרונות, נחזיר את הטוב מביניהם.

בעיית הצריבה

- נתחיל ממקרי הבסיס – אם הדיסק בגודל 0 הרי שאין אפשרות להוסיף קבצים נוספים; לכן, נסמן 0-ים עבור כל הקבצים שנותרו, ונחזיר גם 0 ככמות הזיכרון שנותרה בדיסק. לחילופין, אם **n==0** אזי לא נותרו קבצים לצרוב, ולכן נחזיר את גודל הדיסק **x** ככמות הזיכרון הפנוי שנותר.

```
int bestburn(int sizes[], int n, int x, int chosen[])
{
  int free1, free2, i;

  if (x==0) {
    for (i=0; i<n; ++i) chosen[i] = 0;
    return 0;
  }
  if (n==0) return x;
  …
```


בעיית הצריבה

- בהנחה שאיננו באחד ממקרי הבסיס, נעבור ללב האלגוריתם.
- נניח תחילה כי איננו צורבים את הקובץ **sizes[0]**. נסמן במערך **chosen[]** את הערך 0 במקום המתאים, ונקרא רקורסיבית לפונקציה. בתום הקריאה הרקורסיבית, יהיה בידינו במערך **chosen[]** את הפתרון האופטימאלי ללא הקובץ **sizes[0]**, וכן את כמות הזיכרון שנותרת פנויה בדיסק במקרה זה, **free1**.

```
  chosen[0]=0;
  free1 = bestburn(sizes+1, n-1, x, chosen+1);
  …
```

- כעת משבידינו פתרון זה, נחשב גם את הפתרון השני (זה שעושה שימוש בקובץ **sizes[0]**); נעשה זאת כמובן רק אם **x≥sizes[0]**.

בעיית הצריבה

- סוגיה קטנה שנוצרת כאן היא היכן לאחסן את הפתרון השני – הרי **chosen[]** כבר מכיל פתרון אחד. לכן עבור הפתרון השני עלינו להקצות מערך זמני נוסף **chosen2[]** שישמש לאחסון הפתרון השני; בהמשך הפונקציה, במידה והפתרון השני יתגלה כטוב יותר, נכתוב ל-**chosen[]** פתרון זה.

```
if (sizes[0] <= x)
{
  int *chosen2 = (int*)malloc(n*sizeof(int));

  chosen2[0]=1;
  free2 = bestburn(sizes+1, n-1, x-sizes[0],
                   chosen2+1);
  …
```


בעיית הצריבה

- זהו, סיימנו! כעת עלינו רק להחזיר את הטוב מבין הפתרונות. אם **free2** קטן יותר, נכתוב ל-**chosen[]** את הפתרון השני ונחזיר אותו. אם לא, או אם כלל לא נכנסנו ל-**if** , אזי **chosen[]** כבר מכיל את הפתרון הנכון ולכן נחזיר את **free1**.

```
  if (sizes[0] <= x) {
    …
    if (free2 < free1) {
      for (i=0; i<n; ++i) chosen[i] = chosen2[i];
      free(chosen2);
      return free2;
    }
    free(chosen2);
  }
  return free1;
}
```


בעיית הצריבה

- הקוד המלא של הפונקציה:

```
int bestburn(int sizes[], int n, int x,
             int chosen[])
{
  int free1, free2, i;

  if (x==0) {
    for (i=0; i<n; ++i) chosen[i] = 0;
    return 0;
  }
  if (n==0) return x;

  chosen[0]=0;
  free1 = bestburn(sizes+1, n-1, x, chosen+1);
  …
```


בעיית הצריבה

```
  …
  if (sizes[0] <= x) {
    int *chosen2 = (int*)malloc(n*sizeof(int));

    chosen2[0]=1;
    free2 = bestburn(sizes+1, n-1,
                     x-sizes[0], chosen2+1);

    if (free2 < free1) {
      for (i=0; i<n; ++i) chosen[i] = chosen2[i];
      free(chosen2);
      return free2;
    }
    free(chosen2);
  }
  return free1;
}
```

בעיית הצריבה

- והנה פונקצית **main()** לדוגמה:

```
int main()
{
  int sizes[] = {270,400,220,210,360,540,490};
  int x, i, free, n, chosen[sizeof(sizes)/sizeof(int)];

  n = sizeof(sizes)/sizeof(int);
  printf("\nEnter disc size (MB): "); scanf("%d", &x);

  free = bestburn(sizes,n,x,chosen);
  for(i=0; i<n; ++i) {
    if (chosen[i])
      printf("File %03d:   %3dMB\n", i, sizes[i]);
  }
  printf("Total used: %3dMB\n", x-free);
  printf("Free space: %3dMB\n", free);
  return 0;
}
```


בעיית הצריבה

- דוגמאות הרצה:

```
Enter disc size (MB): 600
File 002:   220MB
File 004:   360MB
Total used: 580MB
Free space:  20MB

Enter disc size (MB): 700
File 003:   210MB
File 006:   490MB
Total used: 700MB
Free space:   0MB

Enter disc size (MB): 800
File 002:   220MB
File 003:   210MB
File 004:   360MB
Total used: 790MB
Free space:  10MB
```


בעיית המבוך

- נתון: מבוך בגודל NxN, המיוצג כמערך דו-ממדי של 0 ו-1. מותר ללכת רק על משבצות שבהן יש 1, ולהתקדם רק במאונך או במאוזן (לא באלכסון).
- הבעיה: נרצה לקבוע אם קיים מסלול מהנקודה (0,0) לנקודה (N-1,N-1).

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

- הרעיון: נתחיל מהנקודה (0,0), ונחפש מסלול על ידי ניסיון שיטתי של כל האפשרויות (בדומה לאופן שבו פותרים מבוך ידנית).

בעיית המבוך

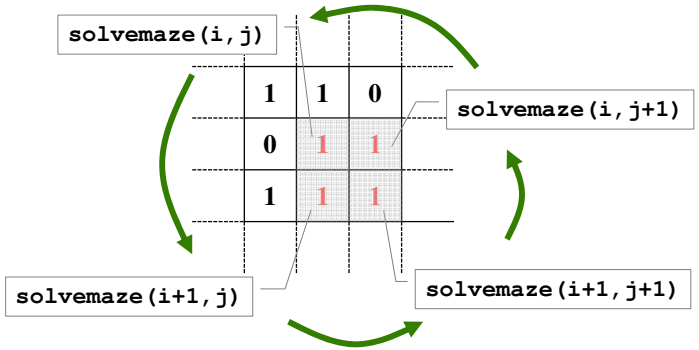
- נציג גרסה ראשונה של האלגוריתם. **maze[N][N]** הוא מערך דו-ממדי המכיל את המבוך; הפונקציה **solvemaze()** מחזירה אם קיים מסלול מהמשבצת (**i**, **j**) למשבצת (**N-1**, **N-1**) במבוך.

```
int solvemaze(maze[N][N], i, j)
{
  if (maze[i][j] == 0) return 0;
  if (i==N-1 && j==N-1) return 1;

  if (solvemaze(maze,i+1,j) || solvemaze(maze,i-1,j) ||
      solvemaze(maze,i,j+1) || solvemaze(maze,i,j-1) ) {
    return 1;
  }
  return 0;
}
```

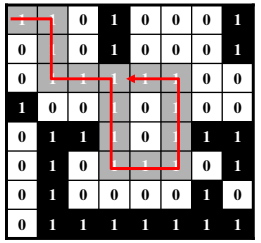
בעיית המבוך

- רגע... הרעיון טוב, אבל הפונקציה נכנסת ללולאה אינסופית!
- הסיבה היא שאם יש במבוך שלנו מעגל, הפונקציה עלולה "לטייל" לאורך המעגל שוב ושוב עד אינסוף.



בעיית המבוך

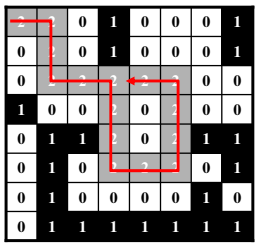
- אנחנו רוצים להימנע מלסגור מעגלים במסלול. נשים לב שמעגל נוצר כאשר במהלך הכניסה לרקורסיה אנו מבקרים במשבצת שכבר שייכת למסלול הנוכחי (כלומר משבצת שממנה הגענו ברקורסיה למשבצת הנוכחית).



- כיצד נדע אם משבצת מסוימת שייכת למסלול הנוכחי שלנו?
- ובכן, נוסיף לשם כך סימון מיוחד במערך **`maze[][]`**, שיציין את כל המשבצות השייכות למסלול הנוכחי שלנו.

בעיית המבוך

- נבצע את הסימון כך: ברגע שאנו מבקרים במשבצת כלשהי במבוך, נחליף את הסימון 1 במערך **`maze[][]`** בסימון מיוחד, נאמר 2, שיסמן שהמשבצת הינה כעת חלק מן המסלול הנוכחי.



- בתחילת כל קריאה רקורסיבית, נבדוק שלא מסומן 2 במשבצת שנכנסו אליה (באותו האופן בו אנו מוודאים כי לא מסומן בה 0), ואם כן – נקטע את התהליך כיוון שהשלמנו מעגל.
- ביציאה מן המשבצת, נחזיר את הסימון 1 למערך.

בעיית המבוך

```
int solvemaze(int maze[N][N], int i, int j)
{
  if (maze[i][j] == 0) return 0;
  if (maze[i][j] == 2) return 0;

  maze[i][j] = 2;
  if (i==N-1 && j==N-1) return 1;

  if ( ((i < N-1) && solvemaze(maze,i+1,j)) ||
       ((i > 0)   && solvemaze(maze,i-1,j)) ||
       ((j < N-1) && solvemaze(maze,i,j+1)) ||
       ((j > 0)   && solvemaze(maze,i,j-1)) ) {
    return 1;
  }
  maze[i][j] = 1;
  return 0;
}
```

בעיית המבוך

- נשים לב שבמקרה וקריאה רקורסיבית כלשהי מחזירה תשובה חיובית, הרי שבמצב זה אנו מחזירים 1 ללא שינוי סימון המשבצת שלנו בחזרה מ-2 חזרה ל-1.
- לפיכך, מה שקורה הוא שבמידה ונמצא מסלול שפותר את המבוך, הרי שאנו מותירים את הסימון '2' לכל אורך מסלול זה כאשר אנו עולים בחזרה ברקורסיה. לפיכך, בתום ריצת הפונקציה (ובהנחה שהיא החזירה 1) אזי לא רק שנדע שאכן יש פתרון למבוך, אלא אף נדע את המסלול עצמו כיוון שהוא מסומן ע"י ערכי ה-2 בתוך המערך **maze[][]**.
- כמו כן נשים לב שאם הפונקציה איננה מוצאת אף מסלול, היא מותירה את המערך **maze[][]** ללא שינוי.

בעיית המבוך

- אגב, אילו היינו רוצים שהפונקציה תחזיר את **maze[][]** ללא שינוי בכל מקרה, היינו יכולים לעשות כן בכך שתמיד נחזיר את הסימון 2 ל-1 במערך לפני היציאה מן הקריאה הרקורסיבית:

```
int solvemaze(int maze[N][N], int i, int j)
{
  …
  if ( ((i < N-1) && solvemaze(maze,i+1,j)) ||
       ((i > 0)   && solvemaze(maze,i-1,j)) ||
       ((j < N-1) && solvemaze(maze,i,j+1)) ||
       ((j > 0)   && solvemaze(maze,i,j-1)) ) {
    maze[i][j] = 1;
    return 1;
  }
  maze[i][j] = 1;
  return 0;
}
```


בעיית המבוך

- דוגמה לפונקצית **main()** העושה שימוש בפונקציה שכתבנו:

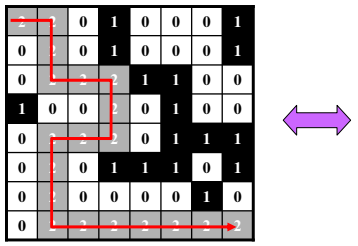
```
void main()
{
  int solved, maze[N][N] = { {1,1,0,1,0,0,0,1},
                             {0,1,0,1,0,0,0,1},
                             {0,1,1,1,1,1,0,0},
                             {1,0,0,1,0,1,0,0},
                             {0,1,1,1,0,1,1,1},
                             {0,1,0,1,1,1,0,1},
                             {0,1,0,0,0,0,1,0},
                             {0,1,1,1,1,1,1,1} };

  solved = solvemaze(maze,0,0);

  if (solved) printmaze(maze);
  else printf("No solution found!\n");
}
```


בעיית המבוך

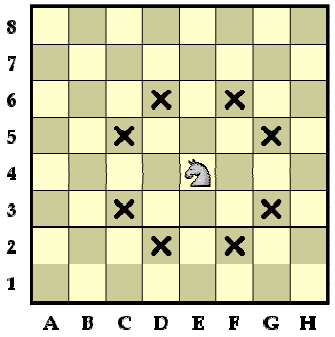
- והנה הפלט של תוכנית זו:



```
2 2 0 1 0 0 0 1
0 2 0 1 0 0 0 1
0 2 2 2 1 1 0 0
1 0 0 2 0 1 0 0
0 2 2 2 0 1 1 1
0 2 0 1 1 1 0 1
0 2 0 0 0 0 1 0
0 2 2 2 2 2 2 2
```

בעיית הפרש

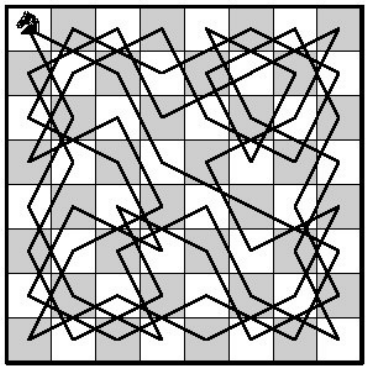
- נתון: נקודת התחלה על לוח שחמט NxN.
- הבעיה: למצוא מסלול של צעדי פרש המתחיל בנקודה זו ומבקר בכל משבצות הלוח פעם אחת בדיוק.



- צעד של פרש מורכב משלושה צעדים: שניים לפנים בכיוון כלשהו (מעלה/מטה/ימינה/שמאלה) ואחד נוסף בניצב לכיוון זה.
- בשרטוט ניתן לראות את המשבצות אליהן יכול לנוע פרש הממוקם במרכז הלוח.

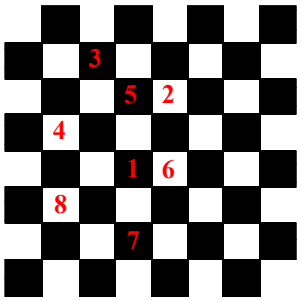
בעיית הפרש

- דוגמה לפיתרון עבור לוח 8x8 (כאן הפרש אף חוזר לנקודת ההתחלה בסוף המסלול):



בעיית הפרש

- נעביר לפונקציה מערך דו-ממדי **board[N][N]** שלתוכו היא תכתוב את המסלול. מסלול ייוצג במערך זה בתור רצף מספרים עוקבים (1,2,3...) בתאים שבהם המסלול עובר, ואלו יציינו את סדר הביקור במשבצות.



- נוכל לייצג במערך **board[][]** גם מסלולים חלקיים, על ידי כך שנרשום 0 בכל משבצת שעדיין לא ביקרנו בה.

בעיית הפרש

צעד המעבר יהיה כזה:

- נניח כי המערך מכיל כבר מסלול חלקי של צעדי פרש, ושהמשבצת שאנו מבקרים בה כעת היא (i,j). כעת:

1. נסמן במשבצת (i,j) את מספרו הסידורי של הצעד הנוכחי.
2. אם מספרו של הצעד הוא N^2 – זהו הצעד האחרון ומצאנו פתרון.
3. אחרת, ננסה להוסיף צעד נוסף למסלול. לכל צעד פרש חוקי שניתן לבצע מהמשבצת הנוכחית, נבדוק שעוד לא בקרנו במשבצת היעד (כלומר שהיא מסומנת באפס), ואם אכן זה כך, ננסה להתקדם למשבצת זו ולהמשיך את החיפוש רקורסיבית מהמשבצת החדשה.
4. במידה ונסיים לנסות את כל הצעדים החוקיים ואף מסלול לא נמצא, סימן שהמסלול החלקי שקיבלנו הוא שגוי. לפיכך, נחזור אחורנית ברקורסיה, תוך שאנו מסמנים 0 במשבצת ממנה אנו יוצאים.

בעיית הפרש

- נתחיל עם מספר פונקציות עזר. הפונקציה **inboard()** בודקת אם משבצת מסוימת היא בתוך גבולות הלוח; הפונקציה **islegal()** בודקת אם צעד מסוים הוא חוקי (כלומר שהמשבצת היא בתוך גבולות הלוח, ועדיין לא ביקרנו בה).

```
#define N 7
#define KNIGHT_MOVE_NUM 8

int inboard(int i, int j) {
  if (i>=0 && i<N && j>=0 && j<N)
    return 1;
  return 0;
}

int islegal(int i, int j, int board[N][N]) {
  return (inboard(i,j) && board[i][j]==0);
}
```


בעיית הפרש

- וכעת לפונקציה הרקורסיבית שפותרת את בעיית הפרש!

```
int solve_knights(int board[N][N],
                  int i, int j, int path_len)
```

- הפונקציה מקבלת את הקואורדינאטות של המשבצת הבאה שאנו מבקרים בה, (**i, j**), וכן את אורך המסלול עד כה, **path_len** (על מנת שנוכל לסמן את מספרו הסידורי של הצעד הבא בלוח).
- בזמן הקריאה לפונקציה, אנו מניחים שהלוח כבר מכיל מסלול חלקי שאורכו **path_len**. הפונקציה תנסה להשלים מסלול זה למסלול מלא, ותחזיר אם היא מצאה פתרון או לא. במידה וכן, היא תשאיר את הלוח עם הפתרון שהיא מצאה; אך במידה ולא – עליה להחזיר את הלוח למצב שקיבלה אותו, על מנת שניתן יהיה לנסות אפשרויות נוספות.

בעיית הפרש

- ראשית נבקר במשבצת (**i,j**) על ידי כך שנרשום בה את מספר הצעד הנוכחי; כיוון שעד כה ביקרנו ב-**path_len** משבצות, הרי שמספרו הסידורי של הצעד הבא הוא **path_len+1**. כעת, במידה והצעד שביצענו מספרו N^2 – סיימנו את המסלול, ואנו מחזירים 1.

```
int solve_knights(int board[N][N],
                  int i, int j, int path_len)
{
  int move_id;
  static int knight_moves[][2] = {
    {-2, -1}, {-2, +1}, {+2, -1}, {+2, +1},
    {-1, -2}, {-1, +2}, {+1, -2}, {+1, +2} };

  board[i][j] = path_len+1;
  if (board[i][j]==N*N) return 1;
  …
```


בעיית הפרש

- במידה ולא השלמנו מסלול מלא, עלינו לנסות ולהרחיב אותו על ידי ביצוע צעד פרש נוסף. נעבור בזו-אחר-זו על כל אחת משמונה האופציות; בכל איטרציה, המשבצת הבאה לביקור תהיה (**next_i,next_j**). במידה ומשבצת זו חוקית (כלומר בתוך הלוח ועדיין לא ביקרנו בה), ננסה להמשיך את המסלול על ידי ביקור בה.
- במידה ואחת מן הקריאות הרקורסיביות מצליחה, נעצור את החיפוש ונחזיר 1.
- במידה ואף אחת מן הקריאות לא מצליחה למצוא פתרון, הרי שהמסלול החלקי שהיה בלוח שגוי, ולכן נחזיר 0. נזכור שעלינו לאפס את המשבצת שהיינו בה במערך **board[][]**, כיוון שאנו יוצאים ממנה בלא לבקר בה.

בעיית הפרש

```
  for(move_id=0; i<KNIGHT_MOVE_NUM; ++move_id)
  {
    int next_i = i + knight_moves[move_id][0];
    int next_j = j + knight_moves[move_id][1];

    if (!islegal(next_i, next_j, board))
      continue;
    if (solve_knights(board, next_i, next_j,
                      path_len+1))
    {
      return 1;
    }
  }

  board[i][j] = 0;
  return 0;
}
```


בעיית הפרש

- נזכיר שאם הפונקציה מחזירה 1, היינו רוצים שכאשר היא מסיימת היא תותיר את הלוח עם הפתרון שמצאה. נוודא שאכן כך קורה:

א. כשאנו במקרה הבסיס (**board[i][j]==N*N**), הרי שהמערך **board[][]** מכיל בתוכו את מסלול הפרש כולו; כיוון שאיננו משנים את הלוח אלא פשוט מחזירים 1, הרי ברור שהלוח שאנו מחזירים עדיין מכיל את הפתרון המלא.

ב. כאשר איננו במקרה הבסיס, אם אנו מחזירים 1 סימן שאחת הקריאות הרקורסיביות שביצענו החזירה 1. במקרה זה, אנו יודעים מתנאי הרקורסיה כי הקריאה הרקורסיבית החזירה את **board[][]** עם הפתרון המלא; כיוון שאנו מחזירים מייד 1 בלא לשנות את תוכן **board[][]**, הרי שאנו מותירים את הלוח עם הפתרון המלא בתוכו, כפי שרצינו.

בעיית הפרש

- לסיום, כפי שעשינו בבעיית הסכום, גם כאן נעטוף את הפונקציה שכתבנו עם פונקצית מעטפת לשם שימוש נוח ע"י משתמש חיצוני.
- ראשית, נשים לב שהפרמטר האחרון (**path_len**) של הפונקציה צריך להיות תמיד 0 בקריאה הראשונה לה; זהו למעשה פרמטר עזר של הפונקציה שאינו רלוונטי למשתמש החיצוני, ולכן לא היינו רוצים אותו בחתימת הפונקציה.
- שנית, נזכור שהפונקציה שלנו מניחה ש-**board[][]** מאותחל בתחילת הריצה לאפסים. במקום לסמוך על המשתמש שיעשה זאת, נבצע אתחול של הלוח לאפסים בפונקצית המעטפת.

בעיית הפרש

- נקבל את פונקצית המעטפת הבאה (שימו לב שהיא מקבלת כפרמטר את המשבצת הראשונה של המסלול, והיא מעבירה משבצת זו לפונקציה הרקורסיבית בתור המשבצת הבאה שיש לבקר בה):

```
int solve_knight_prob(int board[N][N],
                      int i_begin, int j_begin)
{
  int i, j;

  for (i=0; i<N; ++i) {
    for (j=0; j<N; ++j)
      board[i][j] = 0;
  }

  return solve_knights(board,i_begin,j_begin,0);
}
```

בעיית הפרש

- הנה פונקצית **main()** פשוטה שמדגימה את השימוש בפונקציה:

```
int main()
{
  int i_begin = 0, j_begin = 0, solved;
  int board[N][N];

  solved =
    solve_knight_prob(board, i_begin, j_begin);

  if (solved) print_board(board);
  else printf("No solution found!\n");

  return 0;
}
```


בעיית הפרש

- הנה פלט התוכנית עבור **N**-ים שונים; שימו לב לקצב שבו מספר האטרציות גדל עם ההגדלה של **N**:

N=5

```
76497 iterations

 1  12   3  18  21
 4  17  20  13   8
11   2   7  22  19
16   5  24   9  14
25  10  15   6  23
```

N=7

```
8947880 iterations

 1  14   3  38   5  34   7
12  39  10  33   8  37  26
15   2  13   4  25   6  35
40  11  32   9  36  27  44
19  16  21  24  45  48  29
22  41  18  31  28  43  46
17  20  23  42  47  30  49
```