

## Question 1 :

1. הטרידים רצים בתוך תהליך כלשהו , וגם מאפשרים לבצע מספר דברים שונים במקביל , זה שמאפשר לנהל לבצע מספר דברים ופעולות בו זמנית בצורה קלה יותר מאשר עם מספר תהליכים , גם ה קרניל עובד באמצעות טרידים, וגם בתהליכים אשר לכל תהליך מבצעים שכפול ויש לכל תהליך זיכרון משלו אבל בטרידים כל המידע של הזיכרון לא מצריך שכפול ועדכון כל פעם שעוברים בין הטרידים .

2.

Application that is 30% sequential and 40% parallel for

④ 2 processing cores  

$$\text{speedup} \leq \frac{1}{\frac{0.3 + 0.4}{2}} \Rightarrow 1.538 \text{ m/s}$$

⑥ 4 processing cores  

$$\text{speedup} \leq \frac{1}{\frac{0.3 + 0.4}{4}} \Rightarrow 2.105 \text{ m/s}$$

⑧ 8 processing cores  

$$\text{speedup} \leq \frac{1}{\frac{0.3 + 0.4}{8}} \Rightarrow 2.5806 \text{ m/s}$$

⑩ 64 processing cores  

$$\text{speedup} \leq \frac{1}{\frac{0.3 + 0.4}{64}} \Rightarrow 3.216 \text{ m/s}$$

Application with 8% sequential and 92% parallel:-

⑨ 2 processing cores:-      ⑩ 8 processing cores:-  

$$\text{speedup} \leq \frac{1}{\frac{0.08 + 0.92}{2}} = 1.88 \text{ m/s} \quad \text{speedup} \leq \frac{1}{\frac{0.08 + 0.92}{8}} \Rightarrow 5.12 \text{ m/s}$$

⑪ 4 processing cores:-      ⑫ 64 processing cores:-  

$$\text{speedup} \leq \frac{1}{\frac{0.08 + 0.92}{4}} \Rightarrow 3.22 \text{ m/s} \quad \text{speedup} \leq \frac{1}{\frac{0.08 + 0.92}{64}} \Rightarrow 10.8 \text{ m/s}$$

3. THREAD POOLS הוא בעצם מאגר של טרידים מוכנים על מנת לחסוך בזמן היצירה של ה טריד בכל פעם , וזה עוזר למערכת להשתמש בהם ולבצע משימות שונות בעזרתם , למשל ה טרייד פול שימושי עבור תוכניות שכל הזמן מבצעות TASKS שונים .

### Question 3:

A:

```
P_counting( int count )
    P( countLock )    // Acquire lock to count: countLock <- 0
    count--
    if( count <= 0 )    // If no more threads allowed into critical
section
        P( sectionLock ) // Resource full => Acquire section lock:
sectionLock <- 0
        V( countLock )    // Release lock to count: countLock <- 1
    else
        V( countLock)

V_counting( int count )
    P( countLock )
    count++
    if( count > 0)    // Release sectionLock if resource is freed up
        V(sectionLock)    // countLock released after sectionLock so
that waiting
        V(countLock)    // threads do not jump in when before
resource is available
    else
        V(countLock)
```

B:

```
bool flag[2] = {false, false};  
bool flag[2] = {false, false};  
int turn;
```

```
while (!flag[0]) {
```

```
    p0: flag[0] = True
```

```
    turn = 1;
```

```
    while (flag[1] == true & turn == 1)
```

```
    {  
        busy with
```

```
        critical section
```

```
    } end
```

```
    flag[0] = false
```

```
    p1: flag[1] = True
```

```
    turn = 0;
```

```
    while (flag[0] == true & turn == 0)
```

122 sold

```
    flag[1] = false;
```

```
    flag[0] = true;
```

```
while (!flag[1]) {
```

```
    p2: flag[0] = True;
```

```
    turn = 1;
```

```
    flag[0] = false
```

```
    p3: flag[1] = True
```

```
    turn = 0;
```

```
    flag[1] = false;
```

```
    flag[0] = true;
```



C:

$S_1 = 1, S_2 = S_3 = S_4 = S_5 = 0;$

Wait ( $S_1$ )

Signal ( $S_2$ )

Wait ( $S_2$ )

Signal ( $S_3$ )

Wait ( $S_3$ )

Signal ( $S_4$ )

Wait ( $S_4$ )

Signal ( $S_5$ )

Wait ( $S_5$ )

D:

DEAD LOCK הוא מצב בו שניים או יותר טרדים מחכים בלולאת WAIT אינסופית שאומר שהטרדים מחכים שמישהו ישחרר אותם והמישהו זה גם הוא

טרד שמחכה , לכן מערכת ההפעלה או אנחנו בתור המתכנתים צריכים לדאוג ל  
LIVNESS של המערכת כך שלא תתכנס ל DEAD LOCK למשל :

```
public class TestThread {  
    public static Object Lock1 = new Object();  
    public static Object Lock2 = new Object();  
  
    public static void main(String args[]) {  
        ThreadDemo1 T1 = new ThreadDemo1();  
        ThreadDemo2 T2 = new ThreadDemo2();  
        T1.start();  
        T2.start();  
    }  
  
    private static class ThreadDemo1 extends Thread {  
        public void run() {  
            synchronized (Lock1) {  
                System.out.println("Thread 1: Holding lock 1...");  
  
                try { Thread.sleep(10); }  
                catch (InterruptedException e) {}  
                System.out.println("Thread 1: Waiting for lock 2...");  
  
                synchronized (Lock2) {  
                    System.out.println("Thread 1: Holding lock 1 & 2...");  
                }  
            }  
        }  
    }  
  
    private static class ThreadDemo2 extends Thread {  
        public void run() {
```

```

synchronized (Lock2) {
    System.out.println("Thread 2: Holding lock 2...");

    try { Thread.sleep(10); }
    catch (InterruptedException e) {}
    System.out.println("Thread 2: Waiting for lock 1...");

    synchronized (Lock1) {
        System.out.println("Thread 2: Holding lock 1 & 2...");
    }
}
}
}
}
}

```

E:

האלגוריתם של דקר מספק הדרה הדדית, חוסר האפשרות להתרחש מבוי סתום או הרעבה. (מבוי סתום - מצב בו ימתינו מספר תהליכים במצב של משאבים אינסופיים הנכבשים על ידי תהליכים אלו (עצמם); הרעבה - תהליכי ניתוק זה).

אחד היתרונות של אלגוריתם זה הוא שהוא אינו דורש הוראות בדיקה והגדרה מיוחדות (פעולת קריאה אטומית, שינוי וכתיבה), כך שהוא נייד בקלות לשפות תכנות שונות וארכיטקטורת מחשבים. אפשר לקרוא לחסרונות ישימות שלו במקרה של שני תהליכים בלבד ושימוש ב-Busy waiting במקום השעיית התהליך (שימוש בהמתנה עסוקה מעיד על כך שהתהליכים צריכים לבלות מינימום זמן בקטע הקריטי).

לפיכך, האלגוריתם Dekker שימושי במקרים שבהם אתה משתמש במספר מוגבל של תהליכים, כאשר הגידול במספר התהליכים יגדל ביחס למספר הפריטים בשימוש (Pr1 ..., Prn), כמו גם למספר של קטעי קוד להרשאה להיכנס לקטע קריטי עבור כל אחד מהתהליכים.

כדי לפשט את השימוש באלגוריתם של דקר במקרה של תהליכים מרובים בשנת 1981, הוצע אלגוריתם קנייני של פיטרסון, המאפשר לארגן את הכניסה לפי סדר לקטע הקריטי של מספר בלתי מוגבל של תהליכים.

האלגוריתם של פיטרסון - אלגוריתם תוכנה להדרה הדדית זורם ללא איסור הפרעה. הוא הוצע בשנת 1981 על ידי הארי פיטרסון מאוניברסיטת רוצ'סטר (ארה"ב). האלגוריתם של פיטרסון מבוסס על אלגוריתם שדקר השתמש בו. במקור, האלגוריתם נוסח עבור מקרה in-line-2, אך ניתן להכליל אותו לכל מספר שרשרים. האלגוריתם אינו מבוסס על שימוש בהוראות מעבד האוסרות על נעילת פסיקה של אפיק הזיכרון, יש רק אפיקי זיכרון משותפים ולולאת המתנה לקלט בחלק הקריטי של הקוד, מה שנקרא תוכנת אלגוריתם מותנה. האלגוריתם של פיטרסון לוקח בחשבון את היעדר אטומיות בפעולות הקריאה והכתיבה של משתנים וניתן להשתמש בו ללא שימוש בפקודות בקרת הפסקה.

האלגוריתם פועל באופן הבא: לכל תהליך יש דגל משתנה משלו [i] ואת הסיבוב הכולל של המשתנה. שמירת כל המשתנים מתרחשת בזיכרון משותף. עובדת לכידת המשאב מאוחסנת בדגל משתנה, תור משתנה - מספר תהליך לכידת המשאב.

כאשר מבוצע הפרולוג קטע קריטי, תהליך  $P_i$  מצהיר על נכונותו ליישם את האזור הקריטי ומציע מיד לתהליך אחר להמשיך ליישמו. במקרה בו שני התהליכים מגיעים לפרולוג בו זמנית, שניהם מצהירים על נכונותם להציע זה לזה ולרוץ. בנוסף, כל הצעה צריכה להיות ברורה זו לזו. כך, העבודה באזור הקריטי תמשיך להתבצע עד לביצוע המשפט האחרון.

...

ראשית, התהליך מגדיר את הדגל העסוק, ולאחר מכן - תהליך שכן מספר. לאחר שלבים אלה, כל אחד מהתהליכים הכלולים במחזור ההמתנה יציאה מהלולאה מתרחשת אם דגל העסוק מוגדר ומספר התהליך מתאים לשכן.



...

כאשר אתה מנסה לגשת למשאב קריטי, תהליך `enter_region` קורא לפונקציה ומעביר אותה בחדר שלך. אם משאב קריטי כבר תפוס, הפונקציה תיכנס ללולאה המכונה "הדוקה" בהמתנה עד שהמשאב לא ישוחרר. שחרור הפונקציה שיוצרה המשאב `leave_region`. אלגוריתם זה מבוסס על הרעיון של מה שנקרא המתנה אקטיבית, כלומר מצב קבוע של נעילת משתנה סקר עצמי תוך כדי מחזור "מתוח". חוסר היעילות של האלגוריתם הראשי הוא שבילה זמן רב של מעבד למצב המתנה פעיל תוך העסקת משאבים מתהליכים אחרים. לאלגוריתם של פיטרסון יש לחץ קפדני יותר בכניסה לקטע הקריטי, כאשר האלגוריתם של דקר הוא יחסית רך יותר ופחות אגרסיבי.

#### Question 4 : Thread-safe Binary Search Tree

עשינו מחלקת `NODE` שבתוכה יש את קודקוד אב ושני בנים, ובמחלקת `BINARY TREE` יש משתנה מסוג `NODE` וגם יש לנו `MUTEX LOCK` שאתחלנו אותו בבנאי של המחלקה הזו ומימשנו את הפונקציות שביקשו ובתחילת כל פונקציה עשינו ל `MUTEXLOCK` פעולת ה `WAITONE()` ואחרי שנסיים הפעולות שלנו בפונקציה נעשה לו פעולת `RELEASEMUTEX` () זה בכללי מה שעשינו .

#### Question 5 :

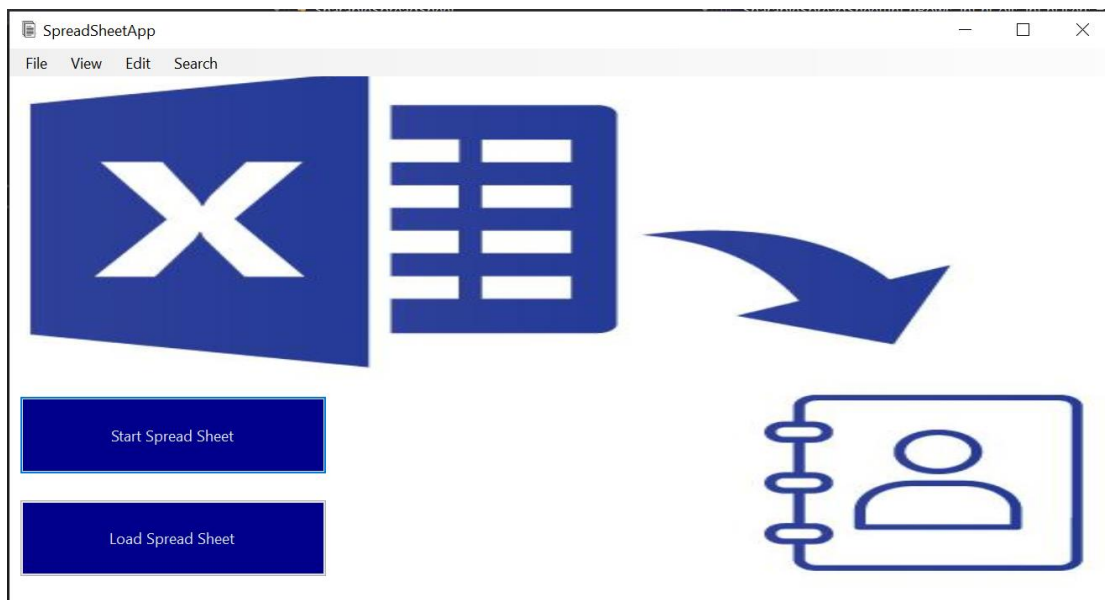
עשינו מחלקת `SHARABLESPREADSHEET` שיש בתוכה 2 מספרים ומערך של `STRING` אלה לייצוג הסופי של הלוח שמייצרים עם הגדלים שלו , הגדרנו גם שני מערכים מסוג `MUTEX` אחד בשביל ה `ROWS` והשני בשביל ה `COLS`

הגדרנו 3 `SEMAPHORES` ושני מספרים אטומים אחד היה נתון והשני למימוש ה `SEMAOHORES` , הגדרנו 2 פונקציות עזר הראשונה משתמשת

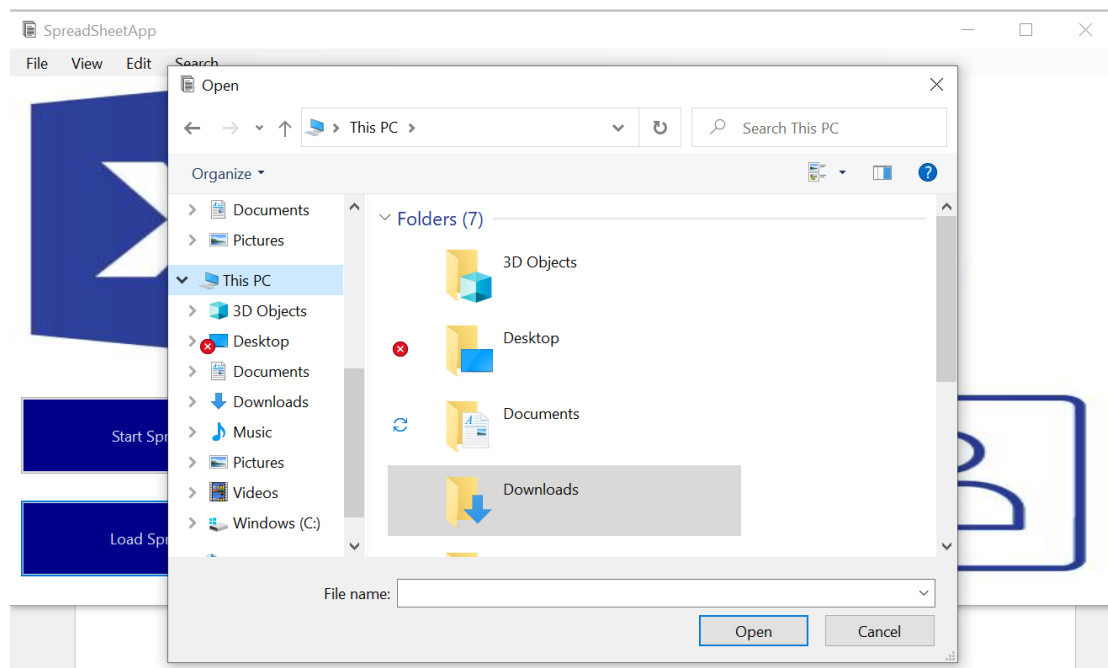
בכל ה-Semaphores שלנו בשביל להפעיל את SEMAPKORE שקשור למשאב שלנו והשנייה ממש משחררת ה-Semaphores שלא שחררו בראשונה ומחזירה ה ה-Semaphores שעשינו להם INCREASE למה שהיו באמצעות DECREASE וכמעט ברוב הפונקציות שעשינו היינו עושים קריאה לפונקציות תלוי אם זו פעולת כתיבה /קריאה , וגם היינו משתמשים בשני מערכי ה MUTEX עושים להם פעולת WAITONE עושים הפעולה שלנו ועושים להם RELEASE בכל הפונקציות זה מה שעשינו בכללי , 2 הפונקציות האחרונות SAVE LOAD היה שימוש ב STREAMREADER למימוש עם קבצים כמו שהיה מבוקש במימוש הפונקציות האלה .

### Question 5 GUI :

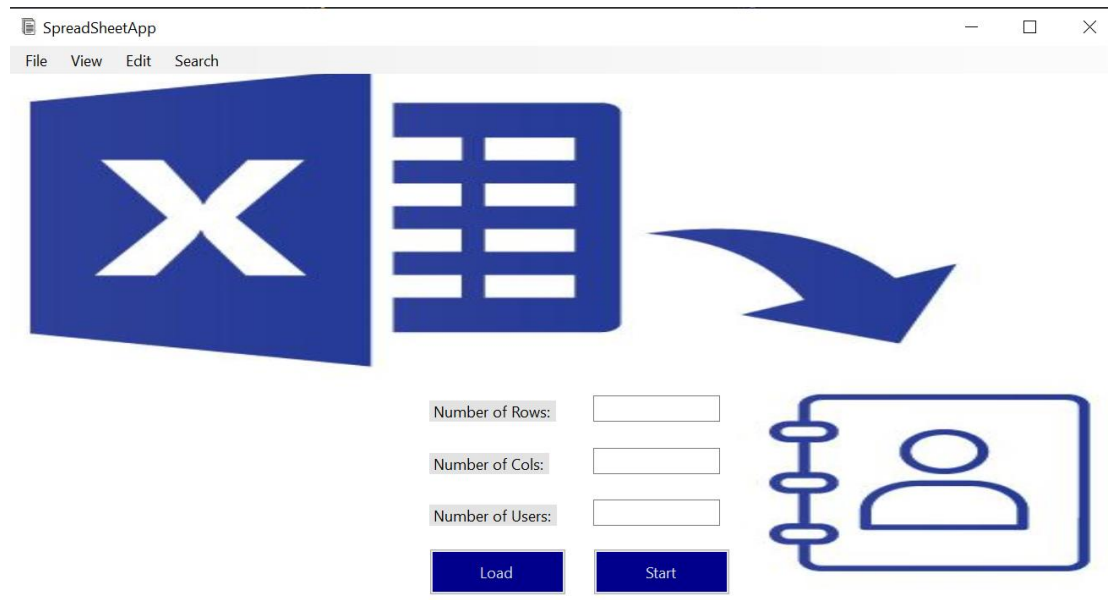
העמוד הראשי:



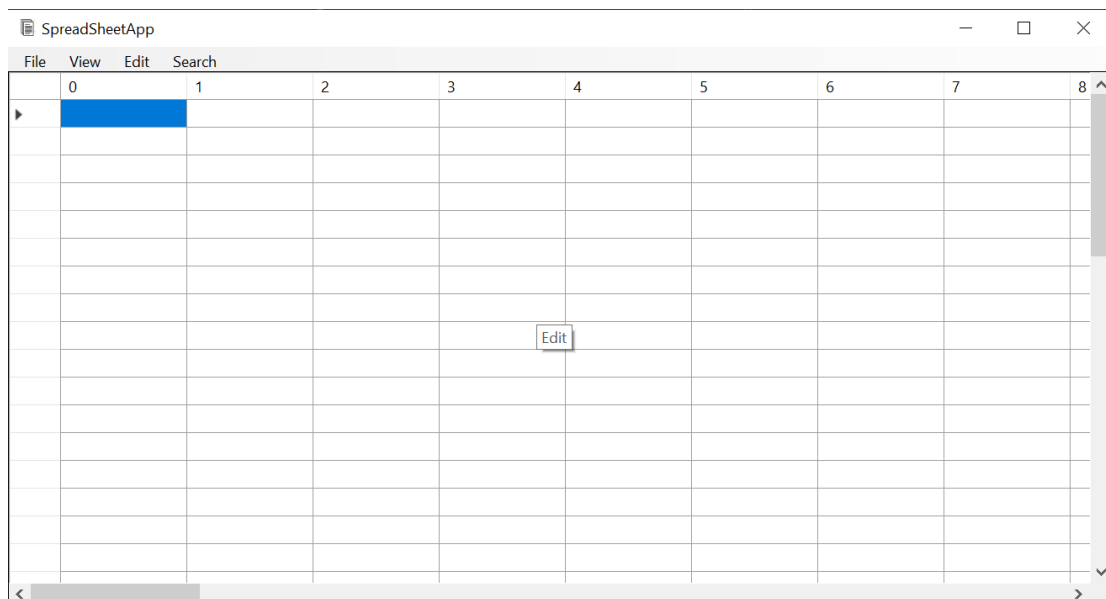
אם המשתמש המשתמש לוחץ על "Load Spread Sheet" יקפוץ חלון חדש של תקיות כדי לטעון אחת חייבת להיות טקסט כדי לטעון:



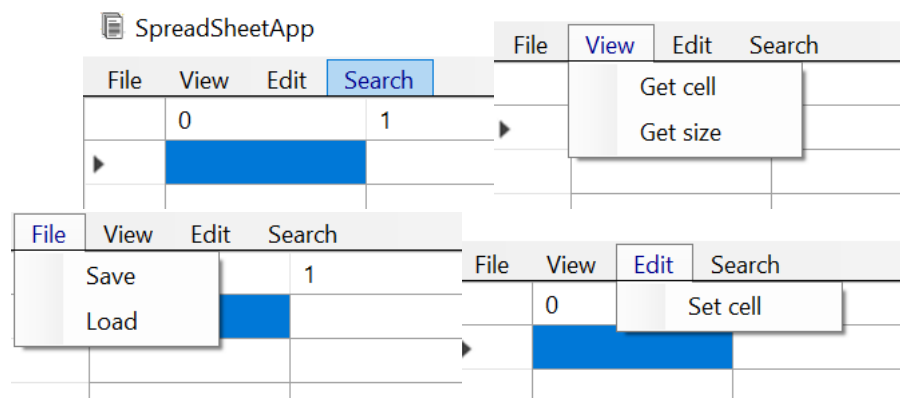
אם המשתמש המשתמש לוחץ על "Start Spread Sheet" נבקש ממנו להכניס את מספר השורות והעמודות ומספר הUsers אם הוא רוצה:



יצרנו Spread Sheet בגודל של 50 50 :



השתמשו ב menu stip כדי להשתמש בפעולות של ה spread sheet :



פעולת ה getcell :

getCell

Enter a row and column to get the string in this position:

Row:

Column:

Ok

אחרי שהמשתמש נתן עמודה ושורה ( למשל 1,1 ):

The cell: [1,1] is empty!

OK

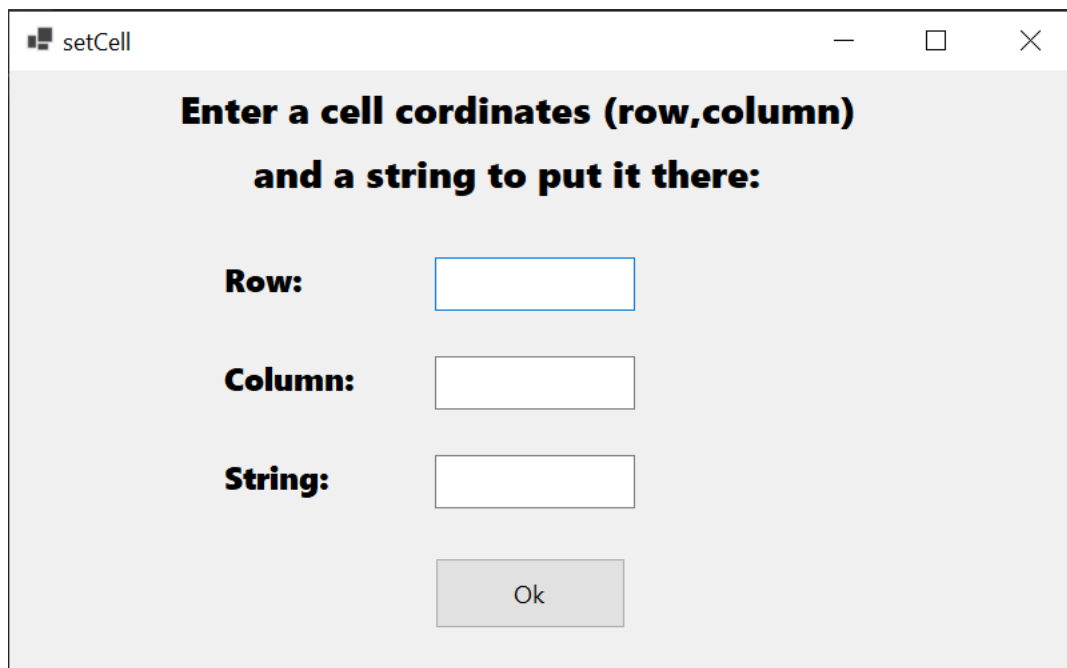
פעולת get Size:

The shareable spread sheet size is: Rows = 50, Columns = 50

OK

פעולת setCell :





setCell

**Enter a cell coordinates (row,column)  
and a string to put it there:**

**Row:**

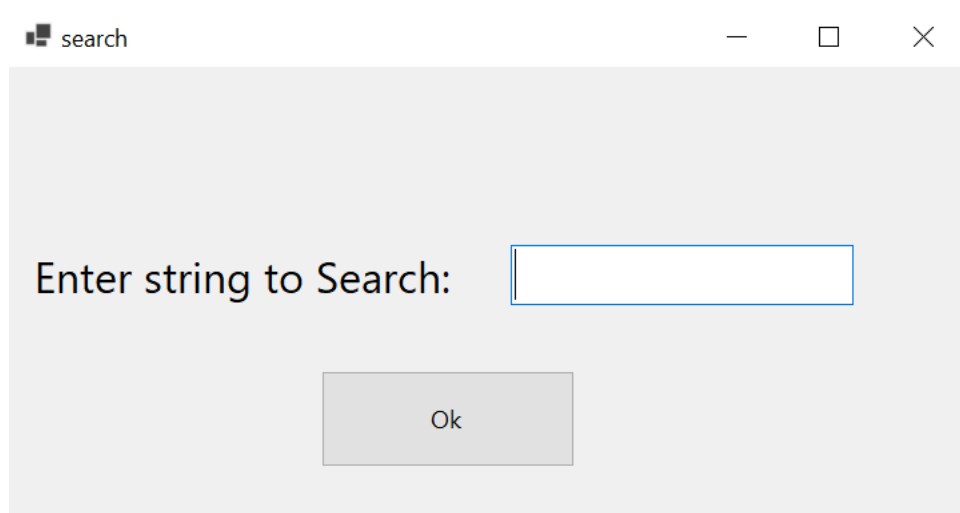
**Column:**

**String:**

Ok

אחרי שהשתמש נתן את העמודה והשורה ומחרוזת המחרוזת תתעדכן  
באפלקציה (spread sheet)

פעולת search:



search

Enter string to Search:

Ok

אחרי ששתמש נתן מחרוזת לחיפוש האפלקציה תקפיץ הודעה אם נמצאת או  
לא, אם כן תכתוב באיזה מקום.

אם המשתמש ירצה לצאת מהמערכת ( במקש על X ):  
המערכת תשאל אם הוא ירצה לשמור את הקובץ או לא:

