אלגוריתמים מהרצאות:

מציאת המכנה המשותף הגדול ביותר (GCD):

תיאור: בהינתן שני שלמים חיוביים n ו-m, יחזיר את השלם הגדול ביותר שמחלק את שניהם.

אלגוריתם: מספר יחלק את m ו-n אם ורק אם הוא יחלק את n ואת $\boxed{r = m\%n}$. כלומר:

$\gcd(m,n) = \gcd(n, m\%n)$ - היפוך הסדר נובע מכך שבהכרח: $m\%n < n$. ממשים זאת ע"י לולאה שממשיכה לרוץ כל-עוד המספר הקטן מבין השניים (n) הוא גדול מאפס. בכל איטרציה, הערך של n הופך להיות הגדול (m) ואילו m%n הופך להיות הקטן (n).

מימוש:

```
unsigned int m, n, temp;

scanf("%u%u", &n, &m);
while(n != 0) {
	temp = n;
	n = m % n;
	m = temp;
}
if(m != 0)
printf("The gcd is %d\n", m);
```

מימוש רקורסיבי:

```
int gcd(int a, int b)
{
	if(a == 0)
		return b;
	return gcd(b%a,a);
}
```

- ניתן לעשות gcd ל-3 מספרים כך שקודם עושים על-2 ואז על עושים על התוצאה שלהם עם השלישי.
- למציאת מכנה משותף הטוב ביותר: $b_1 \cdot b_2 / \gcd(b_1, b_2)$ (כאשר רוצים לחבר שברים).

מציאת שורשי משוואה בשיטת ניוטון רפסון:

תיאור: מציאת ערך x המקיים: $f(x) = 0$ עבור פונקציה נתונה.

אלגוריתם: לולאה שרצה ע"פ תנאי מסוים,יכול להיות מספר איטרציות מסוים, ערך הפונקציה שקרוב ל0 יותר מקבוע $|f(x_{i+1})| < \varepsilon$, הפרש בין שני הפתרונות האחרונים קטן מקבוע $|x_{i+1} - x_i| < \delta$.

בתוך הלולאה ע"פ ערך x קודם (מתחילה בx_0) מחשבים את הx הבא ע"י: $\boxed{x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}}$

מימוש: (עבור כל פונקציה נתונה צריך לממש את $f(x_i)$ ו-$f'(x_i)$ באופן שונה)

המימוש הנ"ל, עבור: $f(x) = x^2 - 9$ כאשר תנאי העצירה הוא מספר איטרציות מסוים.

```
double a, fa, fda;
int i;

scanf("%lf", &a);
fa = a*a - 9;
for(i = 0; i < MAX_ITERATIONS && (fa != 0); ++i) {
	fa = a*a - 9;
	fda = 2*a; /* what if fda == 0 ? */
	a = a - fa/fda;
	printf("Iteration: %d, Solution: %.12f\n", i, a);
```


©ענת עציון

```
}
printf("Solution is: %.12f\n", a);
```

אלגוריתם לבדיקת ראשוניות:

תיאור: בהינתן n שלם חיובי, יש לקבוע האם הוא ראשוני.
אלגוריתם: אם n הוא 1 או מספר שאינו 2 אבל מתחלק ב-2, אז הוא אינו ראשוני. לאחר בדיקה זו רצים בלולאה על כל המספרים האי-זוגיים עד לשורש של n. עבור כל מספר, אם שארית החלוקה של n בו היא אפס, אז המספר אינו ראשוני ואפשר להפסיק לבדוק.
מימוש:

```
int i, n, sqrt_n, is_prime = 1;

if(n == 1 || (n!= 2 && n%2 == 0))
	is_prime = 0;
else {
	sqrt_n = (int)sqrt(n);
	for (i = 3; i <= sqrt_n; i+=2)
		if (n%i == 0) {
			is_prime = 0;
			break;
		}
}
if (is_prime)
	printf("%d is a prime\n", n);
else
	printf("%d is not a prime\n", n);
```

מבוא לסטטיסטיקה: ממוצע וחציון

תיאור: חישוב ממוצע של איברי מערך והחזרתו. בנוסף, החזרת מספר האיברים שמעליו.
אלגוריתם: קליטת איברים לתוך מערך ותוך כדי- חישוב הסכום שלהם. הממוצע הינו הסכום/מספר האיברים. ריצה נוספת עם לולאה על המערך וספירת האיברים שגבוהים מהממוצע.

ממוצע- ממזער את השגיאה הריבועית $\arg\min \sum_{i=1}^{n} (x_i - y)^2$

חציון- ממזער את השגיאה בערך מוחלט: $\arg\min \sum |x_i - y|$

מימוש:

```
double salaries[EMPLOYEES_NUM], sum = 0.0, average;
int i, above_average = 0;

for(i = 0; i < EMPLOYEES_NUM; ++i) {
	scanf("%lf", &salaries[i]);
	sum += salaries[i];
}
average = sum/EMPLOYEES_NUM;
for(i = 0; i < EMPLOYEES_NUM; ++i)
	above_average += (salaries[i] > average);
printf("The average is: %f\n", average);
printf("There are %d salaries above the average\n", above_average);
```

- כאשר הנתונים הם בתוך תחום מוגדר, ניתן למצוא את החציון ע"י בניית היסטוגרמה (גרף שכיחויות) ואז מעבר על המספרים שבמערך השכיחויות מספר פעמים השווה לחצי מהאיברים, והערך שנעצור עליו יהיה החציון (כי חצי מהמספרים קטנים ממנו).

©ענת עציון

מציאת שטח משולש" נוסחת הרון:

תיאור: קבלת קואורדינאטות של משולשים, הדפסת השטח של כל משולש והדפסת השטח הגדול ביותר.

אלגוריתם: התוכנית קולטת את הקואורדינאטות של המשולשים, מחשבת את אורך הצלעות ע"י הנוסחה: $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$ (מעין פיתגורס על הקואורדינאטות שכאילו יוצרות משולש ישר-זווית). באורכי 3 הצלעות משתמשים לחישוב שטח המשולש ע"י נוסחת הרון:

$area=\sqrt{p(p-a)(p-b)(p-c)}$ כאשר a,b,c הם אורכי צלעות המשולש ו p היא נקודה פנימית במשולש המוגדרת ע"י: $p=\frac{a+b+c}{2}$.

מימוש:

```
int main ( void )
{
	double x1, x2, x3, y1, y2, y3, max_area = 0;

	printf("\nEnter coordinates of triangle vertices: ");
	while (6 == scanf("%lf%lf%lf%lf%lf%lf", &x1, &y1, &x2, &y2,
	&x3, &y3)) {
		double area = tri_area(x1, y1, x2, y2, x3, y3);
		if (area > max_area)
		max_area = area;
		printf("\nThe area of the triangle is %lf\n", area);
		printf("\nEnter coordinates of triangle vertices: ");
		}printf("\nThe maximum triangle area is %lf", max_area);
return 0;
}

double distance(double x1, double y1, double x2, double y2)
{
	return sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
}

double heron(double a, double b, double c)
{
	double p = (a + b + c) / 2;

	return sqrt(p * (p-a) * (p-b) * (p-c));
}

double tri_area (double x1, double y1, double x2, double y2,
double x3, double y3)
{
	double d12, d13, d23;

	d12 = distance(x1, y1, x2, y2);
	d13 = distance(x1, y1, x3, y3);
	d23 = distance(x2, y2, x3, y3);

  return heron(d12, d23, d13);
}
```


©ענת עציון

הנפה של ארטוסתנס:

תיאור: מציאת כל המספרים הראשוניים בין 2 למספר N נתון כלשהו.
אלגוריתם: בונים מערך עזר בוליאני המאותחל לTRUE, כלומר: כל מספר הוא ראשוני עד שיוכח אחרת. עוברים בלולאה על כל המספרים מ-2 עד ל $\sqrt{N}$. עבור כל מספר, אם הוא מסומן כראשוני, נסמן את כל הכפולות שלו כלא ראשוניים (עדכון ערך המערך לFALSE). לבסוף נקבל מערך שבו כל ערך המסומן בTRUE, אינדקס המערך הוא מספר ראשוני.
מימוש:

```
for (i = 0; i < N+1; ++i){
       sieve[i] = TRUE;
}
for (p = 2; p <= sqrt_N; ++p){
       if (sieve[p]) {
              for (i = 2; i <= N/p; ++i){
                     sieve[i*p] = FALSE;
              }
       }
}
printf("The prime integers between 2 and %d are\n", N);
for (i = 2; i <= N; ++i){
       if (sieve[i]) {
              printf("%d\n", i);
       }
}
```

חיפוש בינארי במערך ממויין:

תיאור: בדיקה האם מספר מסוים נמצא במערך ממויין ואם כן אז באיזה אינדקס במערך.
אלגוריתם: נשווה את x עם אמצע המערך, אם הם שווים נחזיר את אינדקס מרכז המערך, אם x גדול ממרכז המערך נחפש בחלק הימני, אם x קטן יותר נחפש בחלק המערך השמאלי.
סה"כ נחזיק 3 מצביעים:
low האינדקס הנמוך, high האינדקס הגבוה, mid האינדקס האמצעי (high+low)/2
במידה ואיטרציה הבאה מחפשים בחצי הימני low מקבל את mid+1
במידה ואיטרציה הבאה מחפשים בחצי השמאלי HIGH מקבל את mid-1.
סיבוכיות: $\boxed{O(\log n)}$
מימוש:

```
int binary_search(int a[], int n, int x)
{
       int low, mid, high;

       low = 0; high = n - 1;
       while(low <= high) {
              printf("[Low=%d, High=%d]\n",low,high);
              mid = (low + high) / 2;
              if (x < a[mid])
                     high = mid - 1;
              else if (x > a[mid])
                     low = mid + 1;
              else
                     return mid;
       }
       printf("[Low=%d, High=%d]\n",low,high);
       return -1;
}
```


©ענת עציון

מימוש רקורסיבי:
(מחזיר רק האם האיבר במערך ולא את המיקום שלו).

```
boolean exists(int a[], int n, int x)
{
	if (n == 0)
		return FALSE;
	int mid = n/2;

	if (a[mid] == x)
		return TRUE;
	if (a[mid] > x)
		return exists(a, mid, x);
	else
		return exists(a + mid + 1, n-mid, x);
}
```

טענות:

- אם x במערך אז קיים אינדקס i כך ש $low \le i \le high$ כך שמתקיים: $a[i] == x$
- אם $high < low$ x אינו במערך.
- חיפוש בינארי הוא היעיל ביותר לצורך חיפוש במערך ממויין

מגדלי הנוי:

תיאור: בהינתן 3 מגדלים, יש להעביר n חישוקים המסודרים קטן על גדול ממגדל מקור למגדל יעד כך שגם עליו הם יהיו באותו סדר וגם במהלך המעבר לא ניתן להניח חישוק גדול מעל קטן.

אלגוריתם:
כאשר לא נותרו עוד חישוקים להעביר: מסיימים את הרקורסיה.
מאתחלים משתנה via להיות העמוד עזר שהוא אינו עמוד from או to.
צעד הרקורסיה הינו להעביר n-1 חישוקים לעמוד העזר, ואז מעבירים את החישוק האחרון שנשאר במוט מקור אל מקומו התחתון במוט יעד. ואז קוראים לרקורסיה שוב עם n-1 החישוקים כשהפעם הם עוברים מהמוט עזר למוט יעד.

סיבוכיות: זמן: $\boxed{O(2^n)}$ מודפסות $2^n - 1$ פקודות print לפעולות הזזה.

מקום: $\boxed{O(n)}$, בזמן נתון יש לכל היותר $n+1$ קריאות לפונקציה על מחסנית הקריאות.

מימוש:
(האלגוריתם מדפיס "הוראות לנזיר" כיצד להעביר את החישוקים ממוט למוט).

```
typedef enum{A,B,C} tower_t;

void hanoi(int n, tower_t from, tower_t to)
{
	tower_t via = (A + B + C) - from - to;
	if (n == 0) return;
	hanoi(n-1, from, via);
	printf("Move disc %d from %c to %c\n",n, 'A' + from, 'A' + to);
	hanoi(n-1, via, to);
}
```


©ענת עציון

אלגוריתמי מיון:

Max-Sort:

תיאור: בכל איטרציה נמצא את האיבר המקסימאלי ומעביר אותו לסוף המערך שהולך וקטן .
אלגוריתם: נעבור על המערך ב $n-1$ איטרציות, בכל אחת: נאתר את האיבר הגדול ביותר בין $a[0]...a[n-k]$ ונחליף בינו לבין $a[n-k]$. במצב זה k האיברים הגדולים ביותר כבר יושבים במקומותיהם הסופיים ועל-כן המערך שנעבור עליו לאיתור האיבר הגדול ביותר, הולך וקטן. נמשיך באותה צורה לאיטרציה ה $n-1$ שבסופה המערך ממוין.
איטרציות: (1) $a[0]...a[n-1]$, (2) $a[0]...a[n-2]$ ... עד $a[0]...a[n-k]$

סיבוכיות: $\boxed{O(n^2)}$.

מימוש:

```
int index_of_max(int a[], int n)
{
	int i, i_max = 0;
	for(i = 1; i < n; i++)
	if(a[i] > a[i_max])
		i_max = i;
	return i_max;
}
void max_sort(int a[], int n)
{
	int length;
	for(length = n ; length > 1; length--) {
	int i_max = index_of_max(a, length);
	swap(&a[length-1], &a[i_max]);
	}
}
```

Bubble-Sort:

תיאור: ע"י swap בין צמדים יבעבע האיבר הגדול ביותר לסוף המערך שהולך וקטן.
אלגוריתם: נעבור על המערך ב $n-1$ איטרציות, בכל אחת: אם האיבר ב $a[i-1]$ גדול מהאיבר במקום ה $a[i]$ אז נחליף ביניהם, וכך נבעבע את האיבר הגדול ביותר בין $a[0]...a[n-k]$ למקום ה- $a[n-k]$ במערך. ייתכן שהמערך יהיה מסודר עוד לפני $n-1$ ולכן נסיים כאשר ביצענו $n-1$ או כאשר המעבר האחרון על המערך לא הצריך שום החלפות.
איטרציות: (1) $a[0]...a[n-1]$, (2) $a[0]...a[n-2]$ ... עד $a[0]...a[n-k]$

סיבוכיות: $\boxed{O(n^2)}$.

מימוש:

```
int bubble(int a[], int n)
{
	int i, swapped = 0;
	for(i = 1; i < n; i++)
	if(a[i-1] > a[i]) {
		swap(&a[i], &a[i-1]);
		swapped = 1;
	}
	return swapped;
}
```


©ענת עציון

```
void bubble_sort(int a[], int n)
{
	int not_sorted = 1;
	while( (n > 1) && not_sorted )
	not_sorted = bubble(a, n--);
}
```

Insertion-Sort:

תיאור: בכל פעם נעבור על המערך מסוף הסדרה ועד לאיבר ציר ונביא למצב שממנו והלאה המערך ממוין.

אלגוריתם: ניקח את האיבר ה $pivot = a[n-k-1]$ ונעבירו למיקומו הנכון ביחס ל $a[n-k]...a[n-1]$ אשר כבר מסודרים ביניהם. בתהליך הinsertion נחזיק את הpivot במשתנה זמני, ונבצע העתקה למטה של האיברים הגדולים ממנו, אחד-אחד. ובסיום נטען למיקום הפנוי את הערך ששמרנו במשתנה הזמני (כך ההעברה נעשית ביעילות ולא כמקבץ סתמי של החלפות סמוכות כב-bubble sort).

איטרציות: $pivot = a[n-2]$, $pivot = a[n-3]$ ... $pivot = a[n-k-1]$ ... עד $pivot = a[0]$.

סיבוכיות: $\boxed{O(n^2)}$

מימוש:

```
void insertion_step(int a[], int k, int n)
{
	int temp = a[k]; // the pivot
	int j = k+1;

	while((j < n) && (temp > a[j])){
		a[j-1] = a[j];
		j = j+1;
	}
	a[j-1] = temp;
}
void insertion_sort(int a[], int n)
{
	int i;

	for(i=n-2; i>=0; i--)
	insertion_step(a, i, n);
}
```

השוואה בין אלגוריתמי המיון:

- שלושתם מבצעים אותה כמות של פניה לאיברי המערך והשוואה ביניהם במהלך הסידור.
- אלגוריתמי ה-Bubble-sort וה-insertion-sort מבצעים (הרבה) יותר החלפות במהלך הסידור.
- תוצאות חלקיות של אלגוריתמים אלו טובות יותר משל הMax-sort וזאת בשל הסידורים החלקיים המושגים תוך כדי ריצה.
- אם מהערך ממויין מראש Bubble-sort יגלה זאת אך שני האחרים יבצעו את כל n-1 האיטרציות.

©ענת עציון

מיון באמצעות מיזוג מערכים:

Merge:

תיאור: מיזוג שני מערכים ממוינים לכדי מערך אחד ארוך ממוין המורכב מאיברי המערכים.
אלגוריתם: נחזיק 3 אינדקסים, אחד לכל מערך מקורי ואחד למערך הגדול שנחזיר כפלט. נשווה את איברי שני המערכים ואת הקטן מביניהם נעתיק למערך הממוזג. נקדם את אינדקס המערך שממנו העתקנו ואת אינדקס המערך הממוזג. כשנגיע לסוף אחד המערכים, נעתיק את שארית המערך האחר למערך הממוזג.

סיבוכיות: $\boxed{O(na+nb)=O(n)}$

מימוש:

```
void merge(int a[], int na, int b[], int nb, int c[])
{
	int ia, ib, ic;

	for(ia = ib = ic = 0; (ia < na) && (ib < nb); ic++) {
		if(a[ia] < b[ib]) {
			c[ic] = a[ia];
		}
		else {
			c[ic] = b[ib];
			ib++;
		}
	}
	for(;ia < na; ia++, ic++) c[ic] = a[ia];
	for(;ib < nb; ib++, ic++) c[ic] = b[ib];
}
```

Merge-Sort:

תיאור: נתון מערך המכיל $n=2^k$ איברים. נחלק את המערך לזוגות ונמיין כל זוג. נמזג כל שני זוגות סמוכים לרביעייה ממויינת וכך הלאה עד שנמזג שתי רשימות ממויינות באורך 2^{k-1} לרשימה ממויינת אחת באורך 2^k.
אלגוריתם: מקצים דינאמית מערך עזר באורך n.
בלולאת for חיצונית רצים מ-1 עד n-1 (בכפולות של-2) כאשר האינדקס שומר את אורך הרשימות אותן ממזגים.
בלולאת for פנימית האינדקס מחזיק כל-פעם את תחילת רשימה שצריך למיין ועושים Merge עליה ועל הרשימה שבאה אחריה לתוך מערך העזר. את מערך העזר הממויין מעתיקים בחזרה למקום המקורי של-2 הרשימות במערך הנתון. באיטרציה הבאה האינדקס ידלג שתי רשימות קדימה ויצביע על רשימה נוספת שצריך למזג עם זו שאחריה.

סיבוכיות: $\boxed{O(n\log n)}$

מימוש:

```
int merge_sort(int ar[], int n)
{
	int len;
	int *temp_array, *base;
	temp_array = (int*)malloc(sizeof(int)*n);
	if(temp_array == NULL) {
		printf("Dynamic Allocation Error in merge_sort");
		return FAILURE;
	}
	for (len = 1; len < n; len *= 2) {
		for (base = ar; base < ar + n; base += 2 * len) {
```


©ענת עציון

```
                    merge(base, len, base + len, len, temp_array);
                    memcpy(base, temp_array, 2*len*sizeof(int));
                }
        }
        free(temp_array);
        return SUCCESS;
}
```

Merge-Sort רקורסיבי:

תיאור: מימוש רקורסיבי של מיון מערך בעזרת מיזוג של תתי-מערכים.
אלגוריתם: מקצים דינאמית מערך עזר מחוץ לרקורסיה, באורך n.
שומרים ע"י משתנים את אורכי המערכים ושולחים כל חצי מערך לרקורסיה כך שתחזיר את החצי הזה ממויין. היא ממיינת את החצי בעזרת מערך עזר ואז מעתיקה את הערכים למערך המקורי.
לבסוף לאחר ששני החצאים ממויינים, עושים merge בינהם ואז המערך כולו ממויין.
הסתכלות נוספת: מחלקים את המערך לחצאים וכל אחד מהם גם לחצי עד שנגיע לתאים בודדים, ואז תחל פעולת המיזוג (בחזרה מהרקורסיה) של כל שני חצאים למערך יחיד עד שנגיע למערך המקורי ממויין.
הערות: הפונקציה ממיינת את המערך גם כאשר אינו חזקה של 2.
ניתן לשנות את הקוד כך שישתמש במערך זמני שגודלו $n/2$ - גודל הגדול מכך נדרש רק בפעולת המיזוג האחרונה. בפעולה זו נמלא את המערך הזמני (עם האיברים הקטנים למשל) ולאחר שהתמלא, מעבירים את מה שלא ממויין לחצי אחד של המקורי ומעתיקים לחצי האחר את המערך הממויין השמור במערך הזמני. ואז ממשיכים למיין את החצי הלא ממויין בעזרת שימוש במערך הזמני באותו האופן.
סיבוכיות: זמן: $\boxed{O(n\log n)}$ מקום: $\boxed{O(n)}$ (מערך העזר-n + קריאות- $\log_2 n$).

מימוש:

```
void merge_sort(int a[], int n)
{
        int *tmp_array = malloc(sizeof(int) * n);

        internal_msort(a, n, tmp_array);
        free(tmp_array);
}

void internal_msort(int a[], int n, int helper_array[])
{
        int left = n / 2, right = n - left;

        if (n < 2)
        return;
        internal_msort(a, left, helper_array);
        internal_msort(a + left, right, helper_array);
        merge(a, left, a + left, right, helper_array);
        memcpy(a, helper_array, n * sizeof(int));
}
```


©ענת עציון

Quick-Sort- מיון רקורסיבי יעיל:

תיאור: מיון מערך ע"י קביעת איבר ציר וארגון שאר איברי המערך בשני חלקים, אלה שקטנים ממנו ואלה שגדולים ממנו והעברתו למקום בין שני החלקים (שזה גם יהיה מקומו הסופי במערך הממוין). ואז ביצוע אותה הפעולה רקורסיבית על כל אחד מהחלקים.

אלגוריתם: נהוג לבחור את איבר הציר באקראי (ואז להחליפו עם $a[0]$).

לאחר שנבחר הציר, סורקים את המערך בשני הכיוונים (שני מצביעים- אחד מההתחלה $a[1]$ ואחד מהסוף). כל עוד האינדקס של הסוף מצביע על איברים הגדולים מהציר, ממשיכים להתקדם וכך גם האינדקס של ההתחלה מתקדם כל עוד הוא מצביע על איברים שקטנים מהציר. אם אחד מהם מצביע על איבר שאינו במקומו, האינדקס נעצר, עד אשר גם האינדקס השני נעצר ואז מחליפים ביניהם ערכים (ומקדמים את שני האינדקסים לכיוון המרכז). לאחר ההחלפה הם ממשיכים בסריקה. כאשר האינדקסים חולפים זה על-פני זה- הסריקה הסתיימה והחלוקה הסתיימה.

מעבירים את איבר הציר למקום עליו מצביע האינדקס מהסוף (יצביע על איבר שקטן מהציר). ואז ממיינים ברקורסיה את כל אחד מהחלקים, משמאל ומימין לאיבר הציר שכבר נמצא במקומו הסופי.

סיבוכיות: ערכי הpivot קובעים את זמן הריצה ועומק הרקורסיה.

במקרה הטוב- זמן: $\boxed{O(n\log n)}$ מקום: $\boxed{O(\log n)}$

במקרה הרע- זמן: $\boxed{O(n^2)}$ מקום: $\boxed{O(n)}$ (עומק מחסנית הקריאות, אין שימוש במערך עזר).

האלגוריתם נחשב יעיל כיוון שבממוצע איברי הציר הם טובים ואז הסיבוכיות היא המקרה הטוב.

מימוש:

```
void qsort(int a[], int n)
{
	int p, b = 1, t = n - 1;

	if (n < 2)
	return;
	swap(&a[0], &a[n/2]); /*אמצע המערך נבחר באופן שרירותי*/
	p = a[0];
	while(b <= t) {
		while(t >= b && a[t] >= p )
			t--;
		while(b <= t && a[b] < p)
		b++;
		if ( b < t)
		swap(&a[b++], &a[t--]);
	}
	swap(&a[0], &a[t]);
	qsort(a, t);
	qsort(&a[b], n - b);
}
```


©ענת עציון

סיבוכיות:

חסם עליון: נאמר ש(g(n חסם עליון של (f(n אם:

$$f(n)=O\big(g(n)\big):\exists C>0,\forall n>N,f(n)<C\cdot g(n)$$

חסם תחתון: נאמר ש(g(n חסם תחתון של (f(n אם:

$$f(n)=\Omega\big(g(n)\big):\exists C>0,\forall n>N,f(n)>C\cdot g(n)$$

חסם הדוק ("מפגש החסמים"): נאמר ש(g(n חסם הדוק של (f(n אם:

$$f(n)=\Theta\big(g(n)\big):\exists C>c>0,\forall n>N,C\cdot g(n)>f(n)>c\cdot g(n)$$

נוסחאות שימושיות לחישוב סיבוכיות:

$$\sum_{i=1}^{n}\frac{1}{i}=\Theta\big(\log n\big)$$

$$\sum_{i=1}^{n}i=\frac{n(n+1)}{2}=\Theta\big(n^2\big)$$

$$\sum_{i=1}^{n}i^2=\frac{n(n+1)(2n+1)}{6}=\Theta\big(n^3\big)$$

$$\sum_{i=1}^{n}i^k=\Theta\big(n^{k+1}\big)$$

דוגמאות להתנהגות אסימפטוטית:

$$T(n)=\log_2 n+\log_5 n=\theta\big(\log n\big)$$

$$T(n)=n+n\log n=\theta\big(n\log n\big)$$

$$T(n)=n^2\log n+n^3=\theta\big(n^3\big)$$

$$T(n)=\log(n^3)=\theta\big(\log n\big)$$ *3 זה קבוע שיוצא החוצה מה-log*

$$T(n)=3^{2n}=\theta\big(9^n\big)$$

$$T(n)=2^n+(\log n)^n=\theta\big((\log n)^n\big)$$

זיהוי סיבוכיות לוגוריתמית:

כאשר ישנה לולאה שרצה עבור ערך מסויים והערך קטן כל פעם פי מספר כלשהו (מחלקים את האינדקס בתוך הלולאה בקבוע). כאשר המספר שנשלח לולאה הינו n אז הסיבוכיות: $\theta(\log n)$ אך כאשר המספר שנשלח ללולאה הינו n^n אז הסיבוכיות: $\theta(n\log n)$

```
void f3(int n)
{
	int i, m=1;

	for(i = 0; i < n; i++)
		m *= n;
	while (m > 6)
		m /= 3;
}
```

$$\frac{n^n}{3^k}<6\longrightarrow n^n<6\cdot 3^k\longrightarrow n\cdot\log_3 n<6k\longrightarrow\boxed{\theta(n\log n)}$$

©ענת עציון

הערות טכניות:

אתחולי מערכים

```
int grades[5] = {100, 97, 79, 0, 0};
שקול ל:
int grades[] = {100, 97, 79, 0, 0};
שקול ל:
int grades[5] = {100, 97, 79}
```

הקצאת זיכרון דינאמית:

הפקודה malloc מחזירה void*- מצביע כללי שמצביע על כתובת התחלת שטח הזיכרון שהוקצה (או NULL אם ההקצאה נכשלה).
בדרך-כלל צריך לעשות casting לטיפוס של מצביע כלשהו (בדוגמא זו לא חייבים כי p כבר הוגדר כמצביע מטיפוס כלשהו ולכן ההמרה תתבצע אוטומאטית).

```
double *p;
int n;

p = (double*) malloc(n * sizeof(double));
if (p == NULL)
       return  1;
```

מערכים ומצביעים:

- הפעלת sizeof: מערך- מספר הבתים שתופס כל המערך, מצביע- מספר הבתים שתופס משתנה מסוג מצביע.
- שמו של מערך הוא קבוע, לא ניתן לשנות את הכתובת אליה הוא מצביע.

```
int a[6] = {0};
a = &c; /* שגיאת קומפילציה */
```

- מצביע מאחסן כתובת ולכן גם שמור בזיכרון. ניתן לקבל את הכתובת שבה הכתובת שלו שמורה ע"י & .

```
int x;
int* ptr1 = &x;
int** ptr2 = &ptr1;
```

- אם מצביע מאותחל ל0 (כתובת לא חוקית) אז לא ניתן לשים בו ערכים.
- שתי הצורות הבאות שקולות
  - החזרת מצביעים לאברי המערך: $a+1 \leftrightarrow \&\, a[1]$
  - גישה לאיברי המערך: $*(a+1) \leftrightarrow a[1]$
- יצירת מערך מקצה זיכרון לאיברי המערך, מצביע זקוק לזיכרון שמוקצה ע"י גורם חיצוני

```
int *ptr;
ptr[0] = 100; /* שגיאת זמן ריצה (בהנחה שכתובת הזבל שהמצביע מכיל אינה
חוקית)*/
```

- בהעברת מערך דו-מימדי לפונקציה יש לציין מפורשות את אורך השורה של המטריצה

```
int func(int matrix[][4]);
```

הפונקציה שהוגדרה יכולה לקבל אך-רק מערכים באורך שורה כזה.

©ענת עציון

סוגי משתנים:

משתנים (לוקאליים) אוטומטיים: מוגדרים ומוקצים בזכרון בתחילת בלוק, מתים בסוף הבלוק.

```
{
        int var;
        …
}
```

מוגדרים גם כשנקראת פונקציה

```
int myFunc(int var1, double var2)
{
        …
}
```

משתנים (לוקאליים) סטטיים: מוגדרים רק פעם אחת בתחילת ריצת התוכנית (ללא קשר אם נקרא הבלוק בו הם מוגדרים). מתים רק בסוף התוכנית – מאותחלים בד"כ לאפס (כי חייב להיות להם ערך התחלתי מוגדר) והערך שלהם נשמר לכל אורך התוכנית,

```
int myFunc(int var1, double var2)
{
        static int calls = 0;
        calls++;
        …
}
```

המשתנה calls סופר את מספר הפעמים שהפונקציה myFunc נקראת (שימוש קלאסי במשתנים סטטיים).

משתנים גלובאליים (תכנות רע, או אילוצים לשימושים של מערכת ההפעלה) מוגדרים מחוץ לכל הבלוקים (כולל main) אם אין אתחול מפורש אז הערך הוא אפס (כי חייב להיות להם ערך התחלתי מוגדר) חיים לאורך כל התוכנית ומוכרים ע"י כל בלוק בתוכנית.

```
int myGlob = 0;
int main ()
{
        myGlob = 4;
        …
}
int myFunc(int var1, double var2)
{
        myGlob++
        …
}
```

המרת Float וLong:

אפשרי לאבד מידע בשני המקרים:

float → long	אם הערך שלם, ונמצא בטווח של long, לא נאבד דבר. אם הערך בטווח של long אבל מכיל שבר, נאבד את השבר: 5,731.78 → 5,731 אם הערך מחוץ לטווח, התוצאה לא מוגדרת.
long → float	כיוון של-float רק 6 ספרות דיוק, נקבל את הערך של ה-long אבל **במקורב**, כלומר אנו נאבד דיוק: 760,553,411,987 → 7.60553e+11

©ענת עציון

מחרוזות:

- קבוע מחרוזת בתוכנית ("שלום") מוגדר במקום לקריאה בלבד בניגוד למערך רגיל. עם תחילת ריצת התוכנית, כל קבועי המחרוזת נכתבים לאזור קבועים מיוחד בזיכרון.
במהלך ריצת התוכנית, כל מחרוזת קבועה בקוד מוחלפת אוטומאטית במצביעים מטיפוס char* המציין את מיקומה של המחרוזת בזיכרון הקבועים.

- כאשר מאתחלים מחרוזת לתוך מערך, מתבצעת פעולת העתקה של תוכן המחרוזת מזיכרון הקבועים אל תוך המערך (שנמצא במחסנית- מקום שניתן לכתוב אליו)

```
char* sptr = "I love camels";
```

sptr הינו מצביע למחרוזת באיזור הקבועים.

- זיכרון הקבועים מיועד לקריאה בלבד וכתיבה אליו יגרור שגיאת זמן ריצה:

```
sptr[0] = 'u';
```


©ענת עציון

פונקציות שימושיות:

היפוך סדר איברים במערך/מחרוזת:

```
void reverse(char *s, int len)
{
	int i=0;

	while (i<(len-1)) {
		char temp = s[i];
		s[i] = s[len-1];
		s[j] = temp;
		i++;
		j++;
	}
}
```

החלפת משתנים רגילה (swap):

```
void swap(int* p, int* q)
{
	int tmp = *p;
	*p = *q;
	*q = tmp;
}
```

*משתנים נשלחים לפונקציה עם &: `swap(&a[i],&a[j])`

©ענת עציון

שגיאות נפוצות:

שגיאות קומפילציה:

```
int i = 0;
int *j = *i;
```

הסבר: הפעלת האופרטור * על משתנה שאיננו מצביע.

```
int *i, j, k;
i = &(j+k);
```

הסבר: הפעלת האופרטור & על ביטוי ולא על משתנה.

```
int a[10] = {0,1,2,3,4,5};
int *p = a;
a = p+2;
```

הסבר: ניסיון לשנות את הכתובת אליה מצביע המערך a.

```
char s[] = "Moed";
s[4] = "A";
```

הסבר: הטיפוסים לא מתאימים – "A" הינו טיפוס מסוג *char ואילו s[4] מטיפוס char.

```
int a;
int* b = &a;
void* c = b;
*c = 3;
```

הסבר: אי-אפשר להפעיל אופרטור * על מצביע מטיפוס *void

```
int a=1; *b=&a, **c=&b;
a = (c==&&a) ? 2 : **c;
```

הסבר: האופרטור & נותן כתובת של משתנה אבל a&& אינו כתיבה חוקית (אין כזה דבר "כתובת של כתובת", כתובת חייבת להיות של משתנה!) למעשה && הוא אופרטור ב-C (AND לוגי) והשימוש בו כאן אינו נכון.

שגיאות זמן ריצה:

```
int *j;
int *i = j;
*i = 5;
```

הסבר: כתיבה לזיכרון (i*) שלא הוקצה.

```
char* s = "Hello World!";
while (*s) {
	*s = *s + 1;
```

הסבר: כתיבה לזיכרון שמוגדר כקריאה בלבד (אזור הקבועים).

```
int *i = 0;
int j = *i;
```

הסבר: גישה באמצעות האופרטור * לתוכן של מצביע לnull.

```
int a;
int** b = 0;
*b = &a;
```


©ענת עציון

הסבר: ניסיון לכתוב לכתובת 0.

```
char s[] = "Hello World!";
int i = 0;
while (s[i++]);
s[i] = '\0';
```

הסבר: ניסיון לבצע כתיבה לזיכרון שלא הוקצה (אחרי סוף המערך s).

```
void f(int i) {
	i = i - 1;
}
int r(int i) {
	if(i == 0) return 1;
	f(i);
	return r(i);
}
int main() {
	r(10);
	return 0;
}
```

הסבר: ערכו של i לא משתנה בתוך הפונקציה r ולכן תהיה רקורסיה אינסופית שתגרום לקריסת המחסנית. לולאה אינסופית רגילה (לא רקורסיבית) לא הייתה נחשבת לשגיאה.

```
int a[10] = {10};
a[sizeof(a)-1] = 3;
```

הסבר: sizeof(a) יחזיר את גודל המערך a בבתים שזה יותר מ10, ולכן תהיה כתיבה מחוץ לגבולות המערך.

```
char c;
scanf("%d", &c);
c++;
```

הסבר: scanf תכתוב נתון בגודל של int (בלי קשר למה שהמשתמש הקליד) ולכן תהיה דריסת זיכרון כי גודלו של c הינו בית בודד.

```
char s[5] = {'H','e','l','l','o'};
int d = 5.9999;
s[d] = '0';
```

הסבר: d יאותחל ל5, אבל האינדקס 5 אינו חוקי במערך s שבו יש 5 איברים (0..4).

ללא שגיאות:

```
int i[8] = {0,1,2,3,4,5};
*(i + 2) = 8;

int *p, *q, a, b, d;
p = &a;
q = &b;
d = (p-q);
```

הסבר: הפרש בין שני מצביעים מאותו הטיפוס הוא פעולה חוקית בחשבון מצביעים.

```
float *p, x;
p = &x;
x = *p = 4.5;
```

הסבר: הקוד תקין. ההשמות בשורה השלישית מתבצעות מימין לשמאל.

```
int x=0, y=5;
```


©ענת עציון

```
int b = (0<=y<=3) ? 1 : 1/x;
```

הסבר: התנאי תמיד מתקיים (ללא תלות בערכו של y) ולכן 1/x לא יתבצע

```
void f(double a) {
	a/=0;
}
int main() {
	double b=5;
	return f(b);
}
```

```
char *p = "%dx, %c%c";
printf(p, 10, 'c', 'u');
```

הסבר: printf מצפה לקבל מצביע מטיפוס *char וזה אכן מה שנותנים לו.

©ענת עציון