gcd

ה־gcd של x, y הוא המספר השלם z הגדול ביותר כך ש־$z \mid x$ ו־$z \mid y$. האלגוריתם:

```
def gcd(x,y):
    if x<y:
        x,y = y,x  # Now y <= x
    while y>0:
        x,y = y,x%y
    return x
```

פעולת merge

עבור רשימות ממוינות באורך n ו־m, זה עולה $O(n+m)$ כדי לחבר אותן באופן ממוין.

```
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
    result += right[j:]
```

merge sort

```
def merge_sort(L):
    if len(L) < 2:
        return L[:]
    else:
        middle = int(len(L) / 2)
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:], compare)
        return merge(left, right)
```

quick sort

```
def quicksort(lst):
    if len(lst) <= 1:
        return lst
    else:
        pivot = lst[0] #for a deterministic quicksort
        smaller = [elem for elem in lst if elem < pivot]
        equal = [elem for elem in lst if elem == pivot]
        greater = [elem for elem in lst if elem > pivot]
        return quicksort(smaller) + equal + quicksort(greater)
```

חיפוש בינארי רקורסיבי

```
def rec_binary_search(lst, key, left, right):
    """passing lower and upper boundaries"""
    if left > right:
        return None
    middle = (left+right)//2
    if key == lst[middle]:
        return middle
    elif key < lst[middle]:
        return rec_binary_search(lst, key, left, middle-1)
    else:
        return rec_binary_search(lst, key, middle+1, right)
```

חיפוש בינארי רגיל

```
def binary_search(lst, key):
    n = len(lst)
    L = 0
    R = n-1
    while L <= R:
        mid = floor((L+R)/2)
        if lst[mid] < key:
            L = mid + 1
        elif lst[mid] > key:
            R = mid - 1
        else:
            return mid
    return -1
```

minimax

עבור משחק בלי תיקו, השלבים של minimax הם:

1. מהם המהלכים החוקיים?
2. עבור כל מהלך אפשרי, נבנה את הלוח כאילו בחרנו בו. ע"י רקורסיה נבדוק האם הוא מנצח.
3. אם קיימת בחירה שמובילה למצב מפסיד (עבור היריב), אנחנו במצב מנצח. אחרת, במצב מפסיד.

```
def win(n, m, hlst):
    if sum(hlst)==0:
        return True
    for i in range(m):
        for j in range(hlst[i]):
            move_hlst = [n]*i+[j]*(m-i)
            new_hlst = [min(hlst[i],move_hlst[i]) for i in range(m)]
            if not win(n, m, new_hlst):
                return True
    return False
```

Diffie Hellman

דרך לתאם מפתח כך שגם אם מישהו מצוטט לשיחה, המפתח יהיה רק בידי שני האנשים שמתקשרים אחד עם השני.

Alice and Bob boht make random numbers, a and b respectively.

Alice sends $g^a \mod p$.

Bob sends $g^b \mod p$.

Alice calculates $(g^b)^a \mod p$, Bob calculates $(g^a)^b \mod p$. This is the key.

קארפ־רבין

אלגוריתם למציאת pattern, $P=[0,\dots,m-1]$ בתוך טקסט $T=[0,\dots,n-1]$. האלגוריתם עובד בכך שהוא מחשב finger-print ל־M התווים הראשונים, ומשתמש ב־fingerprint הקודם כדי ליצור את fingerprint הבא. ה־fingerprint משתמש בייצוג מספרי של תווים המייצג מחרוזת כסכום של התו האחרון, התו הלפני אחרון כפול בסיס (לרוב $b=2^{16}$), התו הלפני לפני אחרון כפול בסיס בריבוע, וכך הלאה, ואז עושים modulo כדי לשמור את זה במקום של מספר יחיד. כדי לעבור מ־fingerprint אחד לבא, מחסירים את הערך של התו שירד כפול b^{m-1}, מכפילים את כל ה־fingerprint פי b, ואז מוסיפים את הערך של התו שנוסף.

לכן יכולים להיות false positives שבהם נמצא דברים שלא מתאימים ל־pattern, אבל זה נדיר.

```python
def fingerprint(string, basis, r):
    s = 0
    for ch in string:
        s = (s*basis + ord(ch)) % r
    return s

def text_fingerprint(string,length,basis=2**16,r=2**32-3):
    """ used to compute karp-rabin fingerprint of the text """
    f=[]
    b_power=pow(basis,length-1,r)
    list.append(f, fingerprint(string[0:length], basis,r))
    # f[0] equals first text fingerprint
    for s in range(1,len(string)-length+1):
        new_fingerprint=((f[s-1]-ord(string[s-1])*b_power)*basis + ord(string[s+length-1])) % r
            # compute f[s], based on f[s-1]
            list.append(f,new_fingerprint)# append f[s] to existing f
            return f
def find_matches_KR(pattern,text,basis=2**16,r=2**32-3):
    if len(pattern) > len(text):
        return []
    p=fingerprint(pattern,basis,r)
    f=text_fingerprint(text,len(pattern),basis,r)
    matches = [s for s, f_s in enumerate(f) if f_s == p]
    return matches
```

קוד huffman

דרך להשיג את הקידוד הכי יעיל עבור טקסט מסוים, התווים שמשתמשים בהם יותר יקחו פחות מקום בזיכרון לאחסן. כל תו הוא שורש של עץ מסוים, שהערך שלו הוא כמות ההופעות של התו הזה. בכל שלב, נחבר את שני העצים המינימליים לעץ שערכו הוא סכום הערכים, עד שיש עץ אחד. העץ הזה הוא הקידוד, כאשר ללכת ימינה זה 1 וללכת שמאלה זה 0.

למפל־זיו

דרך נוספת לכווץ טקסט היא בשיטת LZW, אין הנחות על ההתפלגויות של התווים, אלא מניחים שיש חזרות בטקסט. אם יש חזרה באורך K במקום J, במקום לכתוב שוב את אותו הדבר, נכתוב כמה צריך לחזור כדי למצוא את ההופעה הראשונה ואת אורך החזרה. לרוב מסתכלים רק על 4095 התווים האחרונים, אם כך מספיקים לנו 21 ביטים כדי לייצג את ה־offset. לרוב K הוא 5 ביטים. אם כך יעיל לייצג חזרות רק אם אורך החזרה גדול מ־2. לפני כל תו חדש או חזרה מוסיפים ביט יחיד של תווים ושל זוגות m,k. בנוסף, דחיסה כזו שנכנסת לשאומר האם זה תו חדש או חזרה. בשלב הראשון של הדחיסה, נהפוך חזרות לזוגות. זוגות שנכנסים לתוך עצמם לגיטימיים:

$$abcabcabcabc \to [a,b,c,(3,6)]$$

```python
def maxmatch(T, p, W=2**12-1, max_length=2**5-1):
    n = len(T)
    maxmatch = 0
    offset = 0
    for m in range(1, 1+min(p, W)):
        k = 0
        while k < min(n-p, max_length) and T[p-m+k] == T[p+k]:
            k += 1
         if k > maxmatch:
            maxmatch = k
            offset = m
    return offset, maxmatch

def LZW_compress(text, W=2**12-1, max_length=2**5-1):
    result = []
    n = len(text)
    p = 0
    while p < n:
        m, k = maxmatch(text, p, W, max_length)
        if k < 3:
            result.append(text[p])
            p += 1
        else:
            result.append([m,k])
            p += k
    return result
```

מרחק hamming

מרחק hamming מוגדר להיות המרחק המינימלי בין 2 קלטים חוקיים. עבור מרחק מינימלי d,

- אפשר לזהות עד $d-1$ שגיאות.
- אפשר לתקן עד $\lfloor\frac{d-1}{2}\rfloor$ שגיאות.

קוד שמפפה k תווים ל־n תווים ($n \ge k$) עם מרחק מינימלי d נקרא קוד (n,k,d).

מתקיים ה־volume bound, שהוא $2^k \cdot \sum_{h=0}^{\lfloor(d-1)/2\rfloor}\binom{n}{h} \le 2^n$. קוד עבורו יש שוויון בחסם volume bound נקרא קוד מושלם, לדוגמה קוד hamming 7,4,3.

Hamming 7,4,3

```python
def hamming_encode(x3,x5,x6,x7):
    """ Hamming encoding of the 4 bits input """
    x1= (x3+x5+x7) % 2
    x2= (x3+x6+x7) % 2
    x4= (x5+x6+x7) % 2
    return (x1,x2,x3,x4,x5,x6,x7)

def hamming_decode(y1,y2,y3,y4,y5,y6,y7):
    """ Hamming decoding of the 7 bits signal """
    b1 = (y1+y3+y5+y7) % 2
    b2 = (y2+y3+y6+y7) % 2
    b3 = (y4+y5+y6+y7) % 2
    b=4*b3+2*b2+b1  # the integer value
    if b==0:  # no error
        return (y3,y5,y6,y7)
    else:
        y=[y1,y2,y3,y4,y5,y6,y7]
        y[b-1]=(y[b-1]+1) % 2  # correct bit b
        return (y[2],y[4],y[5],y[6])
```

רשימה מקושרת

```
class Linked_list():
  def __init__(self):
    self.next = None
    self.len = 0
  def insert(self, val, loc):
    assert 0 <= loc <= len(self)
    p = self
    for i in range(0, loc):
      p = p.next
    tmp = p.next
    p.next = Node(val)
  def add_at_start(self, val):
    p = self
    tmp = p.next
    p.next = Node(val)
    p.next.next = tmp
    self.len += 1
  def delete(self, loc):
    assert 0 <= loc < len(self)
    p = self
    for i in range(0, loc):
      p = p.next
    # p is the element BEFORE loc
    p.next = p.next.next
    self.len -= 1
```

מילון hash table

כדי לשמור מידע, נמפה את העולם לעולם קטן יותר (רשימה). נשתמש ב־hash כדי להמיר מפתח לאינדקס, הבעיה העיקרית היא התנגשויות של ערכים, כי hash אינה פונקציה חח"ע, וכדי לפתור זאת אנו משתמשים ב־chaining, שזה לשמור רשימה בכל מקום. פקטור העומס α הוא האורך הממוצע של רשימה (אם n כמות האיברים ב־hash table ו־m גודל ה־hash table אז $\alpha = \frac{n}{m}$). אם α קטן אז החיפוש יעיל אך דורש הרבה זיכרון, ואם α גדול נצטרך גם לחפש בתוך רשימה בגודל α בממוצע.

גנרטורים

גנרטור חוקי הוא גנרטור שתמיד לוקח זמן סופי למצוא את האיבר הבא. לכן גנרטור שנותן רק דברים שהופיעו יותר מפעם אחת באיטרטור נתון לא אפשרי, כי יכול להיות ששום איבר לא יופיע פעמיים. דוגמה ל־merge של איטרטורים אינסופיים:

```
def merge(iter1, iter2):
  left = next(iter1)
  right = next(iter2)
  while True:
    if left < right:
      yield left
      left = next(iter1)
    else:
      yield right
      right = next(iter2)
```

מילת הקוד yield מחזירה איבר יחיד. כשנגמרים האיברים בגנרטור, הוא נותן תקלה מסוג StopIteration שאפשר לטפל בה באמצעות `try:, except StopIteration`.

עצי חיפוש בינאריים

מבנה נתונים לאוסף של איברים. ניתן להוסיף, להוריד, לחפש איברים. לכל איבר מפתח וערך, ומצביעים לתת־עצים הימניים והשמאליים. בעץ חיפוש, כל המפתחות משמאל קטנים, ומימין גדולים, מהמפתח של הצומת הנוכחית. פעולות הכנסה וחיפוש נעשים באופן רקורסיבי.

עיבוד תמונות

תמונה בגודל x,y (הרזולוציה) היא מטריצה בגודל x,y כאשר כל תא הוא פיקסל וסרטון זה מטריצה בגודל x,y,t כאשר t זה מימד הזמן. אם התמונה היא בשחור־לבן את כל פיקסל מתאר את חוזק הפיקסל (ערך פיקסל יותר נמוך זה יותר שחור ולהפך). וכאשר התמונה היא צבעונית נשמור שלישיית מספרים בגודל 8 ביט (בדרך כלל) כל אחד המתארים את חוזק הצבעים אדום ירוק וכחול בהתאמה. ולהלן פעולות של המחלקה Matrix:

פעולה	הסבר
`m = Matrix(n, m, c)`	אתחות מטריצה בגודל n,m (שורות זה N) עם ערך התחלתי c (פרמטר c לא חובה)
`m.dim()`	מחזיר את גודל המטריצה בפורמט n,m
`m1 == m2`	לבדוק אם שתי מטריצות שוות
`m.copy()`	להעתיק מטריצה למקום אחר בזיכרון
`not, +, -, *`	פעולות אריתמטיות על מטריצות
`m[i, j]`	האיבר במקום ה־i,j במטריצה (ניתן גם לקבל וגם לשנות אותו)

רעשים בתמונות

כאשר מצלמים תמונה לכל פיקסל מתווסף ערך שנקרא הרעש והוא קורה בגלל הגבלות טכנולוגיית הצילום. מטרה של אלגוריתם להפחתת רעש היא להיפטר מהרעש שנגרם בזמן הצילום. יש 2 דרכים להוספת רעש בצורה דיגיטלית, **רעש גאוסי** כאשר משתמשים בפונקציית הסתברות כדי להוסיף רעש ו**שיטת המלח והפלפל** שבה מוסיפים פיקסלים שחורים ולבנים בצורה אקראית. כדי להיפטר מרעש אפשר להשתמש בממוצע או בחציון של הפיקסל עצמו ושמונת הפיקסלים הסובבים אותו. החציון כמעט ולא מושפע מ־salt and pepper, ואילו הממוצע כן מושפע. הממוצע גם מורח גבולות, והחציון לא. הממוצע מנקה רעש גאוסיאני טוב באיזורים חלקים. החציון מעלים את כל הפרטים הקטנים.

```
def items(mat):
    ''' flatten mat elements into a list '''
    n,m = mat.dim()
    lst = [mat[i,j] for i in range(n) for j in 
range(m)]
    return lst

def local_operator(mat, op, k=1):
    ''' Apply op to every pixel.
        op is a local operator on a square 
neighbourhood
        of size 2k+1 X 2k+1 around a pixel '''
    n,m = mat.dim()
    res = mat.copy() # brand new copy of A
    for i in range(k,n-k):
        for j in range(k,m-k):
            res[i,j] = op(items(mat[i-k:i+k+1,j-k:j
+k+1]))
    return res
```

חוקי log

1. $\log(x \cdot y) = \log(x) + \log(y)$

2. $\log\left(\frac{x}{y}\right) = \log(x) - \log(y)$
3. $\log\left(x^y\right) = y \cdot \log(x)$
4. $a^{\log_a(x)} = x$
5. $\log_b\left(b^x\right) = x$
6. $\log_b(a) = \frac{\log_c(a)}{\log_c(b)}$

חוקי mod

1. $(a \mod n) \mod n = a \mod n$
2. $(a+b) \mod n = ((a \mod n) + (b \mod n)) \mod n$
3. $(ab) \mod n = ((a \mod n)(b \mod n)) \mod n$
4. $a^b \mod c = (a \mod c)^b \mod c$

פעולות של רשימות

`list.append(4)`	$[1,2,3] \to [1,2,3,4]$	$O(1)$ בממוצע
`list.insert(1, 4)`	$[1,2,3] \to [1,4,2,3]$	$O(n)$
`del list[i]`	$[1,2,3] \to [1,3]$ `del list[1]`	$O(n-i)$
`list.pop(i)`	כמו del אבל גם מחזיר את האיבר	$O(n-i)$

סיבוכיות

סיבוכיות מקום	average	worst	פעולה
$O(n+m)$	$O(n+m)$		merge של רשימות ממוינות
$O(n)$	$O(n\log n)$	$O(n^2)$	Quick sort
$O(n)$	$O(n\log n)$		Merge sort
$O(1)$ אם ממיינים במקום	$O(n^2)$		מיון בועות/הכנסה/בחירה
$O(1)$	$O(\log n)$		Binary search
$O(1)$	$O(n+m)$	משתנה	Karp-Rabin
$O(1)$	$O(i)$		גישה לאיבר ה־i ברשימה מקושרת
$O(1)$	$O(1)$ בהינתן node, אחרת $O(i)$		הוספת/מחיקת איבר ברשימה מקושרת
$O(1)$	$O(1)$	$O(n)$	הכנסת/הוצאת/חיפוש איבר בטבלת hash
$O(1)$	$O(\log n)$ אם די מאוזן	$O(n)$ לא מאוזן	חיפוש/הוספת/מחיקת איבר בעץ חיפוש בינארי

Iterated Squaring

```python
def modpower(a,b,c):
    result = 1
        while b>0:
            if b % 2 == 1:
                result = (result * a) % c
            a = (a*a) % c
            b = b//2
    return result
```

סיבוכיות פורמלית

$f(x) = O(g(x))$ אם"ם $\exists c, N. \forall n > N. |f(x)| \le c \cdot |g(x)|$.

הגדרה אסימפטוטית: $\lim_{x\to\infty} \frac{f(x)}{g(x)} < \infty$ כאשר f, g חיוביות.

ייצוג float בזיכרון

1 בשביל sign, 11 בשביל exp ו־52 בשביל info. סה"כ 64 ביטים.

$$2^{exp} \cdot info \cdot sign$$

טווחים ו־slicing

range(a,b) זה כל המספרים מ־a עד $b-1$ כולל, ו־lst$[a:b]$ זה כל האיברים של lst מ־a עד $b-1$ כולל.