סיכום הקורס מבוא למבני נתונים ואלגוריתמים 044268
ע"פ הרצאותיה של דר' ליאן לוית-איתן, חורף 2010
הוכן ע"י איליה מלמד

מבני נתונים בסיסיים:

תור

יישום תור בעזרת מערך :

(כל הפעולות לוקחות $O(1)$)

Insert (x) מכניס איבר חדש
```
A[tail] ← x
tail ← (tail+1) mod N
```

Remove () מוציא איבר
```
  x ← A[head]
  head ← (head+1) mod N
return x
```

Top () מראה את האיבר בראש התור
```
  x ← A[head]
```

IsEmpty () האם התור ריק
```
  if tail==head
    return True
  else
    return FALSE
```

IsFull () האם התור מלא
```
  if (tail+1) mod N ==head
    return TRUE
  else
    return FALSE
```

מחסנית

יישום מחסנית בעזרת רשימה מקושרת :

(כל הפעולות לוקחות $O(1)$)

Push (x) מכניס איבר חדש
```
  tmp ← new LinkedItem
  tmp.data ← x
  tmp.next ← top
  top ← tmp
```

Pop () מוציא איבר
```
  if IsEmpty()
    stop
  tmp ← top
  top ← top.next
  return tmp.data
```

Top () מראה את האיבר בראש המחסנית
```
  return top.data
```

IsEmpty () האם המחסנית ריקה
```
  if top==NIL
    return TRUE
  else
    return FALSE
```

מיונים

Radix Sort	Bucket Sort	Counting Sort	Bubble Sort	Insertion Sort	Heap Sort	Merge Sort	Quick Sort	
$\Theta(d(n+k))$	$\Theta(n)$	$\Theta(n+k)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	Best Case
$\Theta(d(n+k))$	$\Theta(n)$	$\Theta(n+k)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	Avg. Case
$\Theta(d(n+k))$	$\Theta(n^2)$	$\Theta(n+k)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n^2)$	Worst Case
מס' הספרות- טווח- הספרות	האיברים- מפוזרים אוניפורמית טווח איברי- הקלט	טווח איברי הקלט	-	-	-	-	-	מידע נוסף
$\Theta(n+k)$	$\Theta(n)$	$\Theta(n+k)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	זכרון נוסף
דורש מיון יציב עבור כל ספרה d-מס' הספרות k-טווח הספרות	מתאים רק כאשר האיברים מפוזרים אוניפורמית	מיון יציב	מיון יציב				הכי מהיר מבחינת קבועים. Randomized $\Theta(n\log n)$	הערות

יישום מיונים:

CountingSort (A, B, k)

```
for i ← 1 to k
   Count[i] ← 0
for j ← 1 to length[A]
   Count[A[j]]++
for i ← 2 to k
   Count[i]←Count[i]+Count[i-1]
for j ← length[A] downto 1
   B[Count[A[j]] ← A[j]
   Count[A[j]]--
```

HeapSort (A)

```
BuildHeap (A)
for 1 ← heapsize[A] downto 2
   swap A[1] ↔ A[i]
   heapsize[A] ← heapsize[A]-1
   Heapify (A, 1)
```

QuickSort (A, left, right)

```
if left < right
   p ← Partition (A, left, right)
   QuickSort (A, left, p)
   QuickSort (A, p+1, right)
```

Partition (A, left, right)

```
e ← A[left]
L ← left-1
R ← right+1
while TRUE
   repeat R-- until A[R]≤e
   repeat L++ until A[L]≥e
   if L<R
      swap A[L] ↔ A[R]
   else
      return R
```

Rand_QuickSort (A, left, right)

```
if left < right
   p ←Rand_ Partition (A, left, right)
   Rand_QuickSort (A, left, p)
   Rand_QuickSort (A, right, p)
```

Rand_Partition (A, left, right)

```
q ← Random (left, right)
swap A[left] ↔ A[q]
return Partition (A, left, right)
```

MergeSort (A, p, r)

```
if p<r
   q ← ⌊(p+r)/2⌋
   MergeSort (A, p, q)
   MergeSort (A, q+1, r)
   Merge (A, p, q, r)
```

InsertionSort (A)

```
for i ← 2 to length[A]
   key ← A[i]
   j ← i-1
   while j>0 and A[j]>key
      A[j+1] ← A[j]
      j ← j-1
   A[j+1] ← key
```

BubbleSort(A)

```
for i ←1 to length[A]-1
   count ← 0
   for j ← length[A] downto i+1
      if A[j]<A[j-1]
         swap A[j] ↔ A[j-1]
         count++
      if count == 0
         stop
```

RadixSort (A, d, n)

```
for i ← 1 to d
   do a stable sort of A, acording to digit i
```

BucketSort (A)

```
n ← length[A]
for i ← 1to n
   B[⌊nA[i]/m⌋] ← A[i]
for i ← 1 to n-1
   *InsertionSort (B[i])
concatenate the lists {B[0]..B[m]} by order.
```

*-other sorts can be used

- חסם תחתון של בעיית המיון (ללא מידע נוסף): $\Omega(n\log n)$

ערימה - Heap

ערימה היא עץ בינארי כמעט מלא שבו לכל צומת מוגדר ערך "מפתח". ערך של מפתח גדול או שווה לכל ערכי המפתחות של הבנים שלו. הערך בראש הערימה הוא תמיד האיבר הגדול ביותר במערך.
ניתן להגדיר גם ערימת מינימום ע"י שינוי ההגדרה מגדול לקטן, ובכל המימושים ע"י שינוי הסימן ≤ ל-≥. בערמת מינימום האיבר בראש הערמה הוא הקטן ביותר.
Heapify - ממקם את האיבר ה-i במקומו, בהנחה שהוא אינו במקום. מניח שכל הצאצאים של i תקינים. סיבוכיות $O(\log n)$.
Build Heap - יוצר ערמה תקנית ממערך A. סיבוכיות $O(n)$.
Heap Extract Max - מוציא את האיבר המקסי' מהערימה ושומר על מבנה תקין שלה. סיבוכיות $O(\log n)$
Heap Insert- מכניס איבר לערימה למקומו הנכון. סיבוכיות $O(\log n)$
Priority Que Sort - ממיין את המערך ע"י שימוש בטור קדימויות. סיבוכיות $O(n \log n)$

יישום של ערמה בעזרת מערך:

```
Parent (i) =⌊i/2⌋
Left (i) = 2·i
Right (i) = 2·i+1
```

Heapify (A, i)

```
  left ← Left(i)
  right ← Right(i)
  if left ≤ heap_size and A[left] > A[i]
    largest ← left
  else
    largest ← i
  if right ≤ heap_size and A[left] > A[largest]
    largest ← right
  if largest ≠ i
    swap A[i] ↔ A[largest]
    Heapify (A, largest)
```

BuildHeap (A)

```
  heap_size[A] ← length [A]
  for i ← length[A]/2 downto 1
    heapify (A, i)
```

HeapExtractMax (A)

```
  if heap_size[A] < 1
    error "heap underflow"
  max ← A[1]
  A[1] ← A[heap_size[A]]
  heap_size[A]--
  Heapify (A, 1)
  return max
```

HeapInsert(A, key)

```
  heap_size[A]++
  i ← heap_size[A]
  while i>0 and A[Parent(i)] < key
    A[i] ← A[Parent(i)]
    i ← Parent(i)
  A[i] ← key
```

PQSort (A)

```
  S ← Ø
  for i ← 1 to n
    HeapInsert(S, A[i])
  for i ← n downto 1
    SortedA[i] ← HeapExtractMax(S)
```

בעיית הבחירה - Order Statistics

Rand Select - פתרון רנדומלי לבעיית הבחירה ב$O(n)$ - במקרה הממוצע ו-$O(n^2)$ במקרה הגרוע.
Select - פתרון דטרמיניסטי שפותר את בעיית הבחירה ב-$O(n)$ בכל מקרה. *האלגוריתם לא נלמד בקורס, אך צריך לדעת כי הוא קיים

RandSelect (A, p, r, i)

```
  if p==r
    return A[p]
  q ← RandPartition (A,p,r)
  k ← q-p+1
  if (i==k)
    return A[q]
  if i < k
    return RandSelect (A, p, q-1, i)
  else
    return RandSelect (A, q+1, r, i-k)
```

Select (A, p, r, i)

```
  if (r-p+1)≤5
    sort (A[p..r])
    return A[i]
  n ← r-p+1
  Medians[1..n/5] ←A[(l+2):5:(r-2)]
  x=Select (Medians, 1, n/5, n/10)
  A=Partition (A, l, r, x)
  k= find x in A[l..r]
  if i ≤ k
    return Select (A, l, k, i)
  else
    return Select (A, k, r, i-k+1)
```

ע"מ למצוא את החציון נריץ: Select (A, 1, n, $\frac{n+1}{2}$).

עצי חיפוש בינארי-BST

עץ חיפוש בינארי הוא עץ שלכל צומת יש שני בנים. הבן השמאלי קטן מהאב והימני גדול ממנו.

סיורי עץ - Tree walks

PreOrder_TreeWalk (x)
```
If x ≠ NIL
  Print (key[x])
  PreOrder_TreeWalk (Left[x])
  PreOrder_TreeWalk (Right[x])
```

PostOrder_TreeWalk (x)
```
If x ≠ NIL
  PostOrder_TreeWalk (Left[x])
  PostOrder_TreeWalk (Right[x])
  Print (key[x])
```

InOrder_TreeWalk (x)
```
If x ≠ NIL
  InOrder_TreeWalk (Left[x])
  Print (key[x])
  InOrder_TreeWalk (Right[x])
```

הרצת In order תדפיס את העץ בצורה ממויינת, סיבוכיות של כל הסיורים $O(n)$

TreeMax (x) מחזיר את המקסי'
```
while Right[x]≠NIL
  x ← Right[x]
return x
```

TreeMin (x) מחזיר את המיני'
```
while Left[x]≠NIL
  x ← Left[x]
return x
```

TreeDelete (T,z)
```
if Left[z] == NIL or Right[z] == NIL
  y ← z
else
  y ← TreeSuccesor (z)
if Left[y] ≠ NIL
  x ← Left[y]
else
  x ← Right[y]
if x ≠ NIL
  Parent[x] ← Parent[y]
if Parent[y]==NIL
  root[t] ← x
else
  if y == Left[Parent[y]]
    Left[Parent[y]] ← x
  else
    Right[Parent[y]] ← y
if y ≠ z
  key [z] ← key[y]
return y
```

TreeSuccesor (x) מוצא את האיבר הבא בסדר
```
If Right[x] ≠ NIL
  Return TreeMin(Righr[x])
y ← Parent[x]
while y ≠ NIL and x == Right[y]
  x ← y
  y ← Parent[y]
return y
```

TreeInsert (T, z) מכניס איבר לעץ
```
y ← NIL
x ← root[T]
while x ≠NIL
  y ← x
  if key[z] < key[x]
    x ← Left[x]
  else
    x ← Right[x]
parent[z] ← y
if y == NIL
  root[T] ← z
else
  if key[z] < key[y]
    Left[y] ← z
  else
    Right[y] ← z
```

TreeSearch (x, k) מחזיר מצביע למיקום של האיבר
```
if x = NIL or k=key[x]
  return x
if k < key[x]
  return TreeSearch (Left[x], k)
else
  return TreeSearch (Right[x], k)
```

זמן ריצה של כל האלגוריתמים הנ"ל $O(h)$, כאשר h הוא גובה העץ.

אם העץ הוא עץ <u>מאוזן</u> (Balanced) אז גובה העץ שווה ל-$\log n$.

תחזוקת שוטפת של עץ מאוזן תוודא כי כל פעולה תעלה רק $O(\log n)$. ארגון מחדש של BST שלא תוחזק תעלה $O(n \log n)$.

טבלאות ערבול/גיבוב – Hashing

משמשות כאשר רוצים להחזיק פעולות כתיבה, מחיקה וחיפוש ב-$O(1)$, ובסיבוכיות מקום $O(n)$.
נסמן :

- n- מסי המפתחות בטבלה
- m- מסי המקומות בטבלה
- $\alpha = n/m$ - מקדם העומס
- h: {keys}→[1..m] - פונקי ערבול, לוקחת מפתח ונותנת לו ערך מתאים בטבלה.

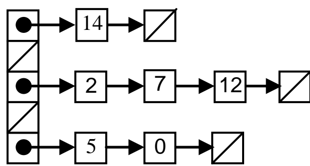
דרישות מפוני ערבול "טובה" :

- סיבוכיות זמן $O(1)$
- לא יוצרת קבוצות לאחר עירבול, מפזרת את המפתחות בצורה אוניפורמית בטבלת העירבול

דוגמאות לפונקי ערבול :

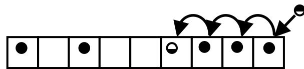
- h(k)=k mod m, יש לבחור עבור m ערכים ראשוניים שלא קרובים לחזקה מדוייקת של 2.
- $h(k)=\lfloor kA \bmod 1\rfloor$ כאשר $0\le A\le 1$ קבוע. עובד בצורה יעילה עבור $A=\frac{\sqrt{5}-1}{2}$

פתרונות עבור התנגשויות של פוני ערבול :



Chaining – כל תא בטבלת העירבול הוא רשימה שבה מאוחסנים כל האיברים בעלי אותו מפתח ערבול. חיפוש ומחיקה יתבצעו ב-$O(h)$, h אורך הרשימה, במקרה הגרוע ביותר מקבלים $O(n)$.
הוספה תתבצע ב-$O(1)$.
ניתן לממש את הרשימה בעזרת BST. הדבר ישפר את המקרה הגרוע ביותר ל- O(logn), אך זה מעלה את זמן ההוספה ל-$O(\log n)$ ומגדיל את הסרבול.

Open Addressing – כל מקום בטבלה מכיל איבר אחד ,ואם פונקי הערבול עלתה על מקום תפוס ,היא תמשיך לחפש עד שתמצא מקום פנוי .החיפוש של המקום הפנוי הבא צריך להיות תלוי רק במפתח ,לכן מרחיבים את פונקי הערבול כדי שתהיה תלויה גם במספר החיפוש.



שיטות לבדיקה :

- $h(k,i)=(h'(k)+i) \bmod m$ – Linear Probing
- $h(k,i)=(h'(k)+c_1 i+c_2 i^2) \bmod m$ – Quadratic Probing
- $h(k)=(h_1(k)+ih_2(k)) \bmod m$ – Double Hashing, שיטה זאת היא היעילה ביותר.

Rehashing – כאשר מגיעים ל-α קרוב ל-1 נרצה להגדיל את גודל הטבלה שלנו . נעשה זאת ע"י פעולת Rehashing אשר מעבירה את כל האיברים לטבלה גדולה יותר , וממקמת אותם במקומם החדש . כל פעם שעושים Rehashing יש צורך לשנות את פונק ' העירבול בהתאם.
למרות שכל פעולת Rehashing לוקחת $O(n)$, בגלל שהן נדירות, נשמרת סיבוכיות ממוצעת $O(1)$ לכל פעולה.

Union Find

טיפוס נתונים מופשט שמקיים שלוש פעולות :

- Union (x, y) – מאחד בין שתי הקבוצות x ו-y לקבוצה אחת.
- FindSet(x) – מחזיר את שם הקבוצה ש-x שייך אליה.
- MakeSet(x) – יוצר קבוצה בעלת איבר יחיד, x.

Union ו-MakeSet תמיד לוקחים $O(1)$. Find לוקח $O(n)$ במקרה הגרוע. ניתן לשפר את האלגוריתם והמבנה נתונים ל- Weighted Union, מה שמוריד את הזמן של Find ל-$O(\log n)$. ניתן גם לשפר את האלגוריתם של Find כך שבזמן מעבר על הערכים האלגוריתם מבצע Path Compretion, הדבר מוריד את הזמן <u>הממוצע</u> של פעולת חיפוש ל-$O(\log^* n)$, לכל ערך מציאותי ניתן להגיד כי $\log^* n\le 5$, ולכן ניתן להגיד שכל פעולת Find לוקחת <u>בממוצע</u> $O(1)$.

V – צומת

E – קשת

גרפים

סיווג גרפים:

- פשוט – לא קיימות קשתות עצמיות או מקבילות
- מכוון – גרף שבו יש חשיבות למוצא והכניסה של קשת (u,v)≠(v,u), בניגוד לגרף מכוון ש בו אין חשיבות ל חילוף בין כניסה ליציאה (v,u)=(u,v).
- קשיר – ניתן להגיע מכל צומת לכל צומת אחר בגרף, |E|≥|V|-1.
- יער – אוסף של עצים, |E|≤|V|-1.

עצים: שלושה תנאים, שאם שנים מהם מתקיימים, השלישי גם:

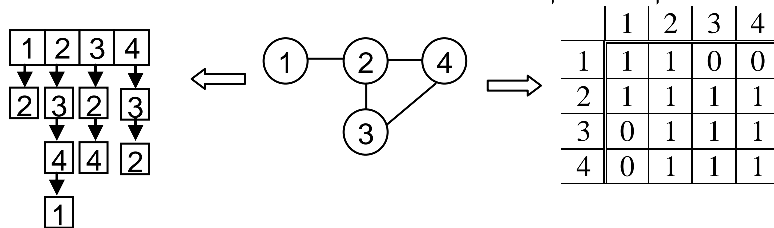
- גרף קשיר
- חסר מעגלים
- |E|=|V|-1

רכיבי קשירות:

- רכיב קשירות – תת גרף שכל הצמתים שלו מחוברים.
- רכיב קשיר היטב – רכיב שניתן להגיע מכל צומת שלו לכל צומת אחר, רלוונטי רק לגרפים מכוונים. (כל צומת בפני עצמו הוא רכיב קשיר היטב)

ייצוג של גרפים:

- רשימת סמיכויות Adjacency List – לכל אחד מהצמתים בגרף יש רשימה מקושרת שמכילה את כל הקשתות שלו.
- מטריצת סמיכויות Adjacency Matrix – כל תא במטריצה הוא 1 אם הקשת בין שני הצמתים בעלי האינדקסים המתאימים קיימת, 0 אחרת. מטריצה של גרף לא מכוון תהיה סימטרית

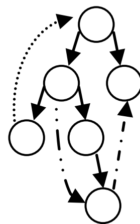


	1	2	3	4
1	1	1	0	0
2	1	1	1	1
3	0	1	1	1
4	0	1	1	1

סיווג קשתות (לאחר הרצת חיפוש):

- קשת עץ – קשת שנכללת בתוך העץ. (——)
- קשת קדמית – קשת שיוצאת מאב קדמון ומגיעה לצאצא. (-··-→)
- קשת אחורית – קשת שיוצאת מצאצא ומגיעה לאב קדמון. (-·-→)
- קשת cross – כל קשת אחרת. (········)

*בגרף בלתי מכוון קשת קדמית היא גם קשת אחורית



חיפוש לרוחב BFS-Breadth First Search

החיפוש בוחן בשיטתיות את כל הצמתים שיוצאים מצומת המוצא S, ורק לאחר שסיים את כל הצמתים שמחוברים ישירות ל-S עובר לרמה הבאה. מחשב את המרחק קצר ביותר של כל צומת לצומת המקור.

```
BFS (G, s)
   Unmark all vertices
   visit (s)
   Enqueue (Q,s)
   BFS_Tree ← Ø
   while Q≠Ø
      v ← Dequeue (Q)
         for every w in Adj[v]
            if w not visited
               visit (w)
               Enque (Q, w)
               BFS_Tree ← BFS_Tree∪{(v,w)}
```

```
BFS_ShortestPaths (G,s)
   for each v∈V
      d[v] ← ∞
   d[s] ← 0
   Enqueue (Q,s)
   while Q≠Ø
      v ← Dequeue (Q)
      for each w in Adj[v]
         if d[w]==∞
            d[w] ← d[v]+1
      Enqueue (Q,w)
```

בדיקת קשירות: לאחר הרצה יחידה של BFS מצומת כלשהו, לבדוק האם קיימים צמתים בלתי מסומנים; אם קיימים, לא קשיר.

סיבוכיות של BFS: $O(|E|+|V|)$.

<u>לאחר הרצה של BFS לא יהיו קשתות קדמיות או קשתות cross בין צמתים שהפרש המרחקים שלהם מצומת השורש גדול מ-2.</u>

חיפוש לעומק DFS-Depth First Search

החיפוש הולך "לעומק" עד שהוא לא גומר לבדוק את כל הקשתות במסלול מסוים שיוצא מ-S הוא לא נסוג. בחיפוש זה מסמנים (לפי הצורך) זמן גילוי וסיום לכל צומת.

DFS_TimeStamping (G)

```
time ← 0
for each v∈V
  d[v] ← unseen
for each v∈V
  if d[v] == unseen
    DFS(G,v)
```

DFS(G, s)

```
  d[s] ← time++
  visit (s)
  for each v∈Adj[s]
    if d[v] == unseen
      DFS(G, v)
  f[s] ← time++
```

GraphDFS (G)

```
Unmark all vertices
for each v∈V
  if v not marked
    DFS (G, v)
```

DFS (G, s)

```
visit (s)
for each v∈Adj[s]
  if v not marked
    DFS (G, v)
    DFS_Tree ← DFS_Tree∪{(s,v)}
```

סיבוכיות של DFS: $O(|E|+|V|)$

מיון קשתות לפי זמני גילוי וסיום: לקשת (v,u)

- קשת עץ: d[v]+1=d[u] or f[v]=f[u]+1 אחרת
- קשת קדמית: d[v]<d[u] and f[v]>f[u] אחרת
- קשת אחורית: d[v]>d[u] and f[v]<f[u] אחרת
- קשת cross.

לאחר הרצת DFS קיימות קשתות cross ⇔ קיימים מעגלים בגרף

מיון טופולוגי

מיון של הצמתים כך שכל צומת תלוי רק בצמתים שלפניו ולא אחריו.

תנאי הכרחי ומספיק לקיום מיון טופולוגי: יהי G גרף קשיר חסר מעגלים ⇔ קיים מיון טופולוגי לגרף G.

מיון טופולוגי הוא לא בהכרח יחיד.

אלגוריתם למיון טופולוגי על בסיס Time Stamping. סיבוכיות: $O(|E|+|V|)$

TopSort (G, s)

```
visit (s)
for each v∈Adj[s]
  if v is unmarked
    TopSort (G,v)
add s to front of TopSort list
```

TopologicalSort (G)

```
for each v∈V
  unmark (v)
for each v∈V
  if v is unmarked
    TopSort (G, v)
```

<u>גרפים ממושקלים</u>

גרפים שבהם לכל קשת יש משקל שונה. משקל יכול להיות שלילי. גרף לא ממושקל ⇐ לכל הקשתות יש אותו משקל

עץ פורש מינימאלי MST

מוגדר רק עבור גרפים לא מכוונים קשירים.

אם לא קיימות קשתות בעלות משקל שווה אז קיים MST יחיד.

MST הוא עץ אשר מחבר בין כל הצמתים בגרף וסכום המשקלים של הקשתות שלו מינימאלי.

לֶמה: יהי G=(E,V), X⊂V, e∈E קשת בעלת משקל מינימאלי שמחברת את X ו-V/{X}, אזי e תהיה בחלק מה-MST.
אם לא קיימות קשתות נוספות בעלות משקל זהה שמחברות בין X ו-V/{X}, אזי e תופיע בכל MST.

שני אלגוריתמים למציאת MST, יכולים לתת תוצאות שונות אם קיימים כמה MST.

Kruskal – ממיין את כל הקשתות לפי משקל , כל פעם מוסיף את הקשת בעלת המשקל המינימאלי אם שני הצמתים שהיא מחברת אינם באותו רכיב קשירות. סיבוכיות: $O(E\log E)=O(E\log V)$

Prim – מתחיל מצומת אקראי s ובכל פעם מוסיף את הקשת המינימאלית שמחברת צומת חדש לרכיב הקשירות של s. סיבוכיות:
$O\big((V+E)\log V\big) = O(E\log V)$

```
MST_Prim (G, wt)
    Q ←V[G] with cost[u]←∞ for all u
    choose random start vertex s
    DecreaseKey (s,0)
    while Q≠Ø
      u ← ExtractMin(Q)
      if u≠s
        T ← T∪{(u,closest[u])}
      for each v∈Adj[u]
        if v∈Q and wt (u,v)<cost[v]
          closest[v] ← u
          DecreaseKey (v, wt(u, v))
    return T
```

```
MST_Kruskal (G, wt)
    T ← (V, Ø)
    for each v∈V[G]
      MakeSet (v)
    sort E by non-decreasing wt
    for each (u,v) ∈E[G] (in sorted order)
      if FindSet(u) ≠ FindSet(v)
        T ← T∪{(u,v)}
        Union (FindSet(u), FindSet(v))
    return T
```

<u>מסלולים קצרים ביותר ממסלול יחיד SSSP</u>

אלגוריתם BFS פותר בעיה זו עבור גרפים שאינם משוקללים. נרצה למצוא פתרון גם עבור גרפים משוקללים.

SSSP קיים רק אם לא קיימים מעגלים שליליים (שסך המשקל שלהם שלילי) בגרף.

<u>אלגוריתם Dijkstra</u>
האלגוריתם הטוב ביותר עבור גרפים <u>ללא</u> משקלים שליליים. בוחר כל פעם בצומת בעלת המשקל הנמוך ביותר, ולא משנה יותר מסלולים. סיבוכיות $O\big((V+E)\log V\big) = O(E\log V)$

<u>אלגוריתם Belman-Ford</u>
יודע לעבוד עם גרפים בעלי משקל שלילי. אם מפעילים אינטרציה נוספת בסוף הריצה שתעבור על כל הקשתות ותבדוק אם יש שינויים נוכל לאתר מעגלים שליליים. סיבוכיות $O(VE)$

```
SSSP_Belman-Ford (G, s, wt)
   for m ← 0 to |V|-1
     d_m ← ∞
     d_m[s] ← 0
     π ← 0
     for m ← 1 to |V|-1
       D_m ← d_{m-1}
     for all (u,v) ∈E
       if d_m[v] > d_{m-1}[u]+wt(u,v)
         d_m[v] ← d_{m-1}[u]+wt(u,v)
         π[v] ← u
output: d_{|v|-1}, π
```

```
SSSP_Dijkstra (G, s, wt)
   S← Ø
   d[s] ← 0
   for all v in V[G]-s
     d[v] ← ∞
     π[v] ← NIL
   Q ← V
   while Q ≠ Ø
     u ← ExtractMin (Q)
     s ← S∪{u}
     for each v∈(Adj[u]-S)
       if d[v] > d[u]+wt(u,v)
         d[v] ← d[u]+wt(u,v)
         DecreaseKey (v, d[v])
         π[v] ← u
```

אלגוריתם למציאת רכיבי קשירות חזקים בגרף מכוון(SCC):

- קריאה ל-DFS(G) לחישוב זמני סיום (f[u]) לכל u.
- חשב את G^T.
- קריאה ל-$DFS(G^T)$ אבל בחירת הצמתים לפי סדר יורד של f[u].
- הוצא את הצמתים של כל עץ מהרצת DFS בפעם השניה כ-SCC.

G^T הוא גרף שבו כיווני כל הקשתות הפוך לכיוונן ב-G: $e(u,v)\in G \Leftrightarrow e(v,u)\in G^T$

שיטות לחישוב נוסחאות רקורסיה

נוסחת רקורסיה: נוסחה המתארת סיבוכיות זמן של אלגוריתם רקורסיבי, לדוגמא: $T(n) = T\left({}^{n}/_{2}\right) + O(n)$

שיטת חזרה (Iteration): לפתוח את הנוסחא ע"מ למצוא ביטוי מתמטי ניתן לחישוב.
דוגמא:

$$T(n) = T(n/2) + n = T(n/4) + n + n/2 = T(n/8) + n + n/2 + n/4 = \ldots$$

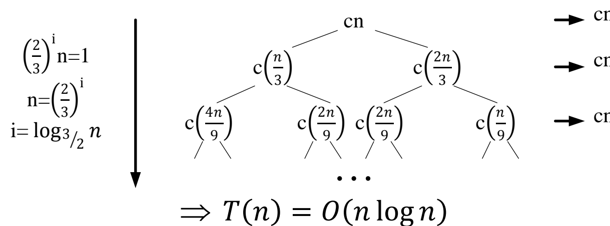
$$\ldots = T(n/n) + n(1 + 1/2 + 1/4 + \ldots + 1/n) = n\sum_{i=0}^{\log n}\frac{1}{2^i} = n\frac{2n-1}{n} = 2n - 1 = \Theta(n)$$

שיטת הצבה (Substitution): טובה במקרה ואנו יודעים איך הפתרון יראה.
ניחוש צורת הפתרון הכללית
שימוש באינדוקציה מתמטית למציאת הקבועים והוכחת נכונות הפתרון.
דוגמא:
$T(n) = 2T\left({}^{n}/_{2}\right) + n$, ננחש פתרון מהצורה $T(n) = O(n\log n)$:

$$T(n) = 2c(n/2)\log(n/2) + n \le cn\log(n/2) + n = cn\log n - cn\log 2 + n = cn\log n - cn + n \le cn\log n\ ,\ c \ge 1$$

חשוב: יש לוודא כי הקבוע לא משתנה וכי אי השיוויון מתקיים, לדוגמא: $T(n) = \frac{c}{3}n^3 \le cn^3$, אבל $T(n) = \left(\frac{c}{3} + 1\right)n^3 \nleq cn^3$
אי קיום של שיטת ההצבה לא מהווה הוכחה לחוסר נכונות, לדוגמא: ייתכן כי T(n)=O(n3), אבל בשיטת ההצבה לא נצליח להוכיח זאת.

שיטת עץ רקורסיה: שימושי במיוחד עבור אלגוריתמי "הפרד ומשול".
בונים עץ רקורסיה (ע"פ חלוקת הרקורסיה) .בכל צומת רושמים את סיבוכיות הפעולות ללא סיבוכיות תתי הרקורסיות. לבסוף סוכמים את הסיבוכיות של כל הצמתים, זאת הסיבוכיות של כל הרקורסיה.דוגמא: $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n)$



$\Rightarrow T(n) = O(n\log n)$

שיטת ה-Master: עבור נוסחאות מהצורה: $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$, (וזהה עבור $\left\lceil\frac{n}{b}\right\rceil$ ו-$\left\lfloor\frac{n}{b}\right\rfloor$)

א. קיים $\varepsilon > 0$ קבוע כך ש: $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ אזי: $T(n) = \Theta\left(n^{\log_b a}\right)$

ב. $f(n) = \Theta\left(n^{\log_b a}\right)$ אזי $T(n) = \Theta\left(n^{\log_b a} \cdot \log n\right)$

ג. קיים $\varepsilon > 0$ קבוע כך ש: $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ וגם $a \cdot f\left(\frac{n}{b}\right) \le c \cdot f(n)$ עבור $c>1$ קבוע וכל-n גדול מספיק אזי: $T(n) = \Theta(f(n))$