הפקולטה להנדסה

רשתות בחירה
מציאת האיבר הK

אורי גלאם

ברק בן אקון

פרויקט לקורס חומרה מכילה מטסטביליות

מנחה אקדמי: דר' מוטי מדינה

24/4/24

תוכן עניינים

הסבר תיאורטי

רשתות מיון הן מבנה נתונים המיועד לבצע מיון באופן מקבילי ויעיל. הרעיון הבסיסי מאחורי רשתות מיון הוא להשתמש בסדרת פעולות השוואה-החלפה קבועה, שבה כל פעולה משווה בין זוג אלמנטים נתונים ומחליפה אותם במידת הצורך. רשת המיון מורכבת משכבות, כאשר כל שכבה יכולה לבצע סדר של השוואות במקביל. הגודל של רשת המיון מתייחס למספר ההשוואות הכולל שמבוצע במהלך הפעולה, והעומק של הרשת מתייחס למספר השכבות, כלומר, למספר הפעולות המקביליות המרבי שנדרשות לביצוע המיון.
רשתות מיון מאפשרות מיון מקבילי ויעיל של נתונים באמצעות רשת של מקביליות. הגודל של רשת המיון מתייחס למספר ההשוואות הכולל שנעשה והעומק מתייחס למספר השכבות. כל שכבה ברשת מבצעת קבוצה של השוואות במקביל, מה שמאפשר עיבוד מהיר ויעיל של הנתונים.

יתרונות וחסרונות של רשתות מיון ביחס לאלגוריתמי מיון אחרים:

יתרונות: מיון מקבילי - רשתות מיון מאפשרות מיון מקבילי ויעיל של נתונים, מה שהופך אותן למתאימות ליישומים הדורשים זמן ריצה מהיר. **יעילות-** רשתות מיון ידועות ביעילותן, הן מבחינת זמן הריצה והן מבחינת השימוש בזיכרון. **גמישו**ת - ניתן להתאים את רשתות המיון בקלות לגדלים שונים של קלט ולדרישות מיון ספציפיות .**מטסטביליות -** ניתן להפוך רשתות מיון ליציבות באמצעות טכניקות מיוחדות, מה שהופך אותן למתאימות ליישומים הדורשים עמידות בפני שגיאות.

חסרונות: מורכבות- רשתות מיון הן מורכבות יותר ליישום מאשר אלגוריתמי מיון אחרים, כגון Quicksort ו Merge Sort. **עומס-** רשתות מיון דורשות עומס חומרה משמעותי, מה שעלול להוות מגבלה ביישומים מסוימים. **גודל -** רשתות מיון יכולות להיות גדולות יחסית, הן מבחינת שטח האחסון והן מבחינת צריכת החשמל.

רשת ביטונית והשימוש בHalf Cleaner-

רשת ביטונית היא אחד הדוגמאות הפופולריות לרשת מיון, והיא מבוססת על אלגוריתם ביטון-מרג'. פעולת ה Half Cleaner-היא ליבת הרשת הזו, שבה כל אלמנט בצמתים זוגיים משווה את עצמו לבן זוגו הבא בסדר, והם מחליפים מקומות אם הצורך כך שהאלמנט הגדול יעבור למיקום הגבוה יותר. השימוש ב Half Cleaner-מאפשר לפצל כל קבוצת נתונים לשניים, אחד עם האלמנטים הגדולים והשני עם האלמנטים הקטנים יותר, תוך הבטחת יעילות גבוהה בתהליך המיון.
רשת ביטונית מאופיינת ביכולתה לסדר רצפים ביטוניים, כלומר רצפים המורכבים מסדרה

יורדת של אלמנטים המתוחם בסדרה עולה. ה "Half Cleaner" הוא שלב חשוב ברשת זו, שבו כל קלט מתוחם לשני רצפים ביטוניים - חצי עליון וחצי תחתון - כאשר כל אלמנט בחצי העליון קטן או שווה לאלמנטים בחצי התחתון.

מטסטביליות ברשתות מיון

מטסטביליות היא תכונה של מערכת המחייבת אותה להתנהג באופן עקבי ועקיב גם כאשר מתרחשות שגיאות או תקלות. בהקשר של רשתות מיון, מטסטביליות פירושה שהמיון יתבצע באופן נכון גם כאשר ישנם קלטים לא יציבים או שגיאות במהלך פעולת המיון. המטסטביליות חשובה במגוון יישומים, במיוחד אלו הדורשים אמינות גבוהה. לדוגמה, ברשתות מחשבים, חשוב שרשתות מיון יוכלו לבצע מיון נכון של נתונים גם במקרה של תקלות חומרה או תוכנה.

בפרויקט זה, נעסוק במטסטביליות של רשתות בחירה, שבה נדרשת עמידות בפני שגיאות במהלך המיון. כאשר מעוניינים במטסטביליות, משתמשים בקומפרטורים שונים שסיבוכיותם היא ((O(LOG(N מה שמבטיח עמידות גבוהה יותר בפני שגיאות אפשריות.
בהקשר של מטסטביליות, רשתות מיון צריכות להיות עמידות לשגיאות ולהבטיח שהמיון יתבצע באופן יציב גם כאשר ישנם קלטים לא יציבים. כפי שצוין לעיל, זה מחייב שימוש בקומפרטורים מיוחדים עם סיבוכיות של: ((O(LOG(N.

מציאת הסטטיסטי הסדר k

בפרויקט זה נחפש פיתרון למציאת הסטטיסטי הסדר k באמצעות רשתות בחירה. מציאת האלמנט ה k בתחום האלגוריתמי לרוב קוראת על ידי שימוש ממוקד של אלגוריתמים מתקדמים של מיון, כגון ,Quicksort שמתאפיין ביעילות גבוהה והתמקדות במציאת האיבר הK ולא על ידי מיון כלל המערך ואז החזרת האיבר הK ולכן האלגוריתמים האלה מתבצעים בתוחלת בO(n). בפרוייקט זה ננסה לממש בצורה דומה את מציאת האיבר הK על ידי שימוש ממקוד של רשת מיון הנלמדת בקורס.

Let $T(n)$ be a monotonically increasing function that satisfies:

- $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$
- $T(1) = c$

Where $a > 0, b > 1, d \geq 0$. Then:

$$T(n) = \begin{cases} \Theta(n^d), & if\ d > \log_b a \\ \Theta(n^d \log n), & if\ d = \log_b a \\ \Theta(n^{\log_b a}), & if\ d < \log_b a \end{cases}$$

Figure1 -master furmula

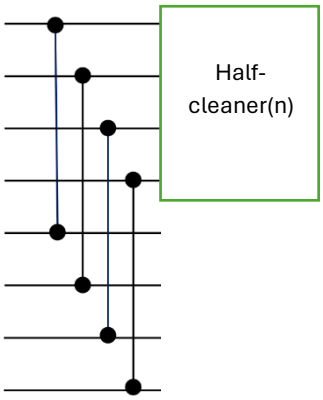
פסאודוקוד

רשת בחירה פשוטה

1. Selction_newtwork(arr,k)
2. Compare the first half with the second half of the array(first to first, second to second and etc.) using half cleaner
3. Replace if needed the number in the first half with the second half with greater to the second and lowest to the first half(it mean if a in second and b in first and and b>a so a and b switch)
4. Return on 2 and 3 until If length(arr)==1 ,with the half k in it, and each iteration defined k_new=k-$\left\lfloor\frac{lengt(arr)}{2}\right\rfloor$

ציור הרשת

בהנחה שהאיבר ה-K נמצא בחצי העליון, ללא הגבלת הכלליות על הגודל ובהנחה ויש 2n איברים.



נכונות

נכונות הרשת נובעת מהנכונות של "half cleaner" ומהבסיס של ממיין ביטוני כלומר אנו מניחים שהמערך הוא ביטוני במידה ואינו ביטוני אז נסדר אותו לכך, מכיוון שב" half cleaner" אנו נחלק את המערך לשניים ונעשה השוואה בין האיבר הראשון מהחלק הראשון עם האיבר הראשון מהחלק השני של מערך המספרים. ההשוואה מתבסס על איתור האיבר הגדול ואיתור האיבר הקטן והחלפה ביניהם, כך שכל הקטנים יהיו מצד אחד והגדולים מהצד האחר. כלומר כל איטרציה מתמקדת יותר על האיזור בו K אמור להיות עד שנגיע אליו ונדע במדויק שזה הK שצריך להחזיר. בגלל שסידרנו את המערך שיהיה מסודר בחלקים של K לפי גודל, אז נוכל להתמקד בחלק שבו אנחנו צריכים למצוא את ה- K. בסוף נעשית השוואה בין 2 המספרים לבין המקום ש ל K והמקום מעליו או מתחתיו. בסוף האלגוריתם, שני המספרים האלו, יהיו בהכרח במקום שלהם אילו זה היה מיון מלא.

Cost

$$Cost(2n) =_1 Cost(n) + \frac{n}{2} =_2 \theta(n)$$
$$Cost(n)_3 = Cost\left(\frac{n}{2}\right) + \frac{n}{4}$$
$$\mathrm{a} = 1, \mathrm{b} = 2, \mathrm{d} = 1 \rightarrow \mathrm{d} = 1 > \log_b a = 0 \rightarrow \theta(n^d) = \theta(n)$$

ההשוואה של 1 נובעת מכך שכל פעם אנו מתמקדים רק על חצי, וה*n/2* נובע מהמספר השוואות שקורות שהן חצי בדיוק מהמשתנים.

שיוויון 2 נובע מנוסחת המאסטר שציינו לעיל, והפיתוח שלה שמתאים כאן בשיוויון 3.

Delay

$$Delay(2n) = Delay(n) + 1 = \theta(\log(n))$$
$$Delay(n) = Delay\left(\frac{n}{2}\right) + \frac{1}{2} \rightarrow a = 1, b = 2, d = 0 \rightarrow d$$
$$= \log_b a = 0 \rightarrow T(n) = \theta(n^d logn) = \theta(logn)$$

העומק נובע מכך שכל שכבה אנו מתקדמים על חצי מהערכים, בנוסף התוספת של ה1 נובעת מכך שאת כל המשוונים בכל שכבה ניתן לעשות בפעם אחת במקביל. התוצאה היא הגיונית כי אנחנו מקבלים בסוף חצי עץ מאוזן והעומק של עץ מאוזן הוא logn, אנו נקבל חצי עץ מאוזן כי אנחנו מחלקים לשתיים ומתקדים בחצי כל פעם.

רשת בחירה עם קוד גריי ללא מטסטביליות

אלגוריתם זהה לאלגוריתם הקודם שמתווסף לו עוד שלב של המרה מקוד גריי למספרים רגילים.

Cost

עלות זהה $\theta(n)$

Delay

השהייה זהה $\theta(logn)$

רשת בחירה עם קוד גריי ומכילה מטסטביליות

האלגוריתם דומה לאלגוריתם הקודם נתאר בבירור עם התוספות:

1. Selction_newtwork_metastabily(arr,k)
2. Compare the first half with the second half of the array(first to first, second to second and etc.) using half cleaner
3. The compare go like this each variable there was a M inside it replace to be the res of it(for example 00M will be (001,000)) then take the max of the two numbers and compare to the other max of the other number
4. If the number are equal you need to compare bit by bit In the order of 1>0>M from the MSB to the LSB.
5. Replace if needed the number in the first half with the second half with greater to the second and lowest to the first half(it mean if a in second and b in first and and b>a so a and b switch)
6. Return on 2 and 3 until If length(arr)==1 ,with the half k in it, and each iteration defined k_new=k-$\left\lfloor\frac{lengt(arr)}{2}\right\rfloor$

נכונות

הנכונות נובעת מהנכונות של האלגוריתם הבסיסי שהוכחנו.

Cost

העלות של האלגוריתם דומה לאלגוריתם הקודם ההבדל הוא יש את תופסת השיוויון:

$$Cost(2n) =_{\mathbf{1}} \left(Cost(n) + \frac{n}{2}\right) cost(comparator) =_{\mathbf{2}} \theta(nlogn)$$

$$Cost(n)_{\mathbf{3}} = Cost\left(\frac{n}{2}\right) + \frac{n}{4}$$

$$\mathrm{a} = 1, \mathrm{b} = 2, \mathrm{d} = 1 \rightarrow \mathrm{d} = 1 > \log_b a = 0 \rightarrow \theta(n^d) = \theta(n)$$

ההשוואה של 1 נובעת מכך שכל פעם אנו מתמקדים רק על חצי, וה*n/2* נובע מהמספר השוואות שקורות שהן חצי בדיוק מהמשתנים.

שיוויון 2 נובע מנוסחת המאסטר שציינו לעיל והקומפרטור שלמדנו בהרצאה שהעלות שלו היא *logn* והפיתוח שלה שמתאים כאן בשיוויון 3.

Delay

$$Delay(2n) = Delay(n) + 1 = \theta(\log(n))$$

$$Delay(n) = Delay\left(\frac{n}{2}\right) + \frac{1}{2} \rightarrow a = 1, b = 2, d = 0 \rightarrow d$$
$$= \log_b a = 0 \rightarrow T(n) = \theta(n^d logn) = \theta(logn)$$

העומק נובע מכך שבכל שכבה אנו מתקדמים על חצי מהערכים, בנוסף התוספת של ה1 נובעת מכך שאת כל המשוונים בכל שכבה ניתן לעשות בפעם אחת במקביל. התוצאה היא הגיונית כי אנחנו מקבלים בסוף חצי עץ מאוזן והעומק של עץ מאוזן הוא logn, אנו נקבל חצי עץ מאוזן כי אנחנו מחלקים לשתיים ומתקדים בחצי כל פעם. ניתן לראות שההשפעה של הפעלת השיוויון השונה לא תשפיע על העומק מכיוון שעדיין ניתן למקבל את התהליכים. אנו מסתכלים על העומק בתור כמה שלבי שיוויון וניתן לבצע במקביל וללא תלות בשלב הקודם, ולא כמה זמן לוקח לכל שיוויון להתבצע.

סיכום

רשת הבחירה מומשה על בסיס הרעיון של ממיין ביטוני והשימוש בhalf cleaner. משום כך הגענו לעלות של אלגוריתם של O(n) מצב זה הינו אופטימלי מכיוון שאנו יודעים כי המינימום הנדרש לביצוע מיון שלם הוא שימוש ב-nlogn השוואות, במקרה זה עבור כל איבר K העלות היא O(n) ולכן העלות הכוללת של מיון שלם בדרך זו תהיה O(nlogn) כלומר בסופו של דבר זהו גודל אופטימלי, כמובן בהתייחס לכך שהעלות של כל קומפרטור היא 1, בשונה ממצב של מטסטביליות שהעלות של קומפרטור היא logn. בצורה דומה העומק הינו אופטימלי כי בניית האלגוריתם הביאה אותנו לעומק של logn. על מנת לבנות את הקומפרטור לשימוש במצב מטסטבילי השתמשנו בידע הנלמד מהקורס ומימשנו בהתאם לכך. לסיכום, בפרוייקט זה למדנו על רשתות מיון בצורה כללית והשוואה לאלגוריתמי מיון, התמקדנו ברשת בחירה והשוואתה למציאת סטטיסטי הסדר K מהקורס של מבנ"ת מאלגוריתמים.

קוד פייתון

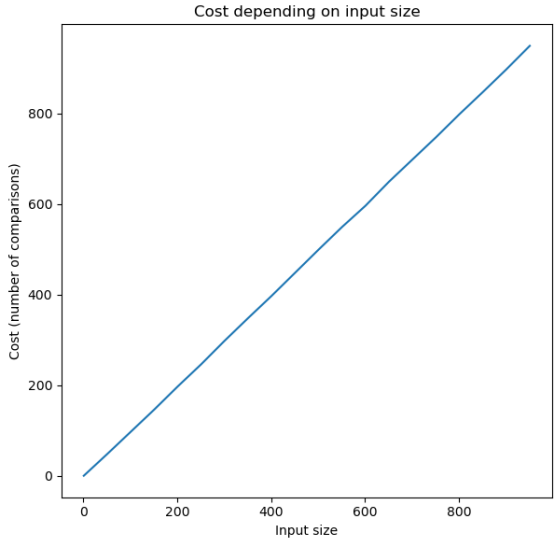
רשת בחירה פשוטה

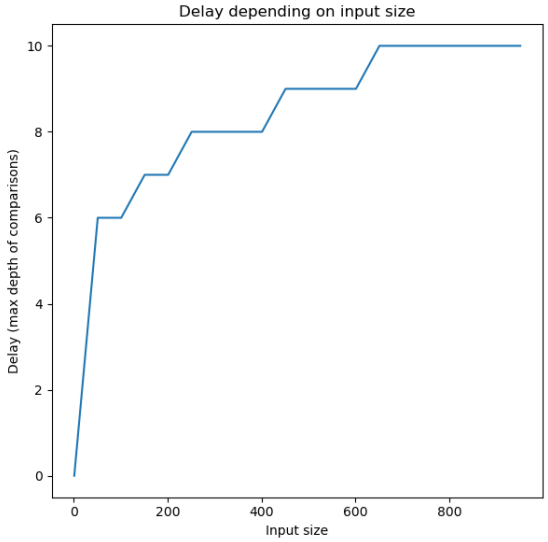
```
def selection_network(arr, k):
    n = len(arr)
    if n == 1:
        return arr[0], 0, 0  # No comparators used for a single-element
array
    mid = n // 2
    left = arr[:mid]
    right = arr[mid:]
    cost = 0
    for i in range(mid):
        if left[i] > right[i]:
            left[i], right[i] = right[i], left[i]
        cost += 1  # Increment cost for each comparison
    delay = 1  # At least one comparison was made
    if k <= mid:
        val, left_cost, left_delay = selection_network(left, k)
        cost += left_cost
        delay = max(delay, left_delay + 1)  # Increment delay if we go
deeper
    else:
        val, right_cost, right_delay = selection_network(right, k -
mid)
        cost += right_cost
        delay = max(delay, right_delay + 1)  # Increment delay if we go
deeper
    return val, cost, delay

# Test the function
arr = [3,5,6,4,2,1]
k = 3
val, cost, delay = selection_network(arr, k)
print(f"Value: {val}, Cost: {cost}, Delay: {delay}")
```

התוצאה שקיבלנו:

```
Value: 3, Cost: 5, Delay: 3
```

ויזואליזציה





ניתן לראות שאכן עלות הפונקציה היא לינארית והעומק הינו לוגריתמי.

רשת בחירה עם קוד גריי ללא מטסטביליות

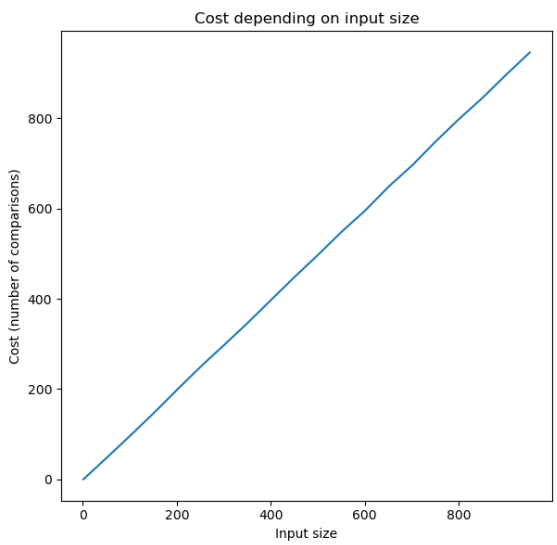
```
def gray_to_int(gray):
    binary = gray[0]
    for i in range(1, len(gray)):
        binary += str(int(binary[i - 1]) ^ int(gray[i]))
    #print(gray,int(binary, 2))
    return int(binary, 2)
def selection_network_gray(arr, k, is_gray=True):
    # Convert Gray codes to integers only in the first call
    if is_gray:
        arr = [gray_to_int(g) for g in arr]


    n = len(arr)
    if n == 1:
        return arr[0], 0, 0  # No comparators used for a single-element
array
    mid = n // 2
    left = arr[:mid]
    right = arr[mid:]
    cost = 0
    for i in range(mid):
        if left[i] > right[i]:
            left[i], right[i] = right[i], left[i]
        cost += 1  # Increment cost for each comparison
    delay = 1  # At least one comparison was made
    if k <= mid:
        val, left_cost, left_delay = selection_network_gray(left, k,
is_gray=False)
```

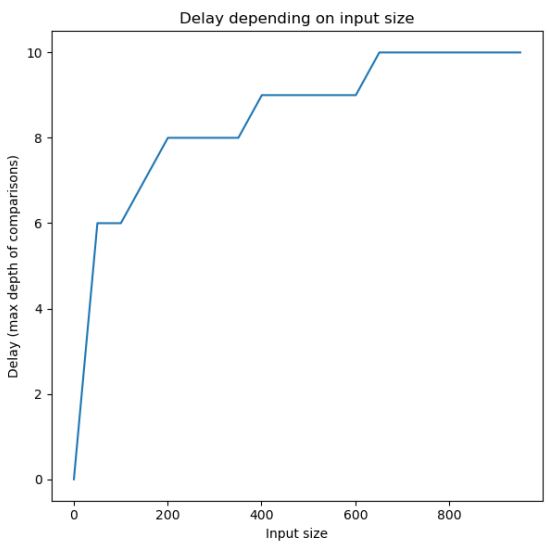
```
        cost += left_cost
        delay = max(delay, left_delay + 1)  # Increment delay if we go
deeper
    else:
        val, right_cost, right_delay = selection_network_gray(right, k
- mid, is_gray=False)
        cost += right_cost
        delay = max(delay, right_delay + 1)  # Increment delay if we go
deeper
    return val, cost, delay


# Test the function
arr = ['00', '01', '11', '10']  # Gray codes for 0, 1, 2, 3
k = 2
val, cost, delay = selection_network_gray(arr, k)
print(f"Value two bits: {val}, Cost: {cost}, Delay: {delay}")

arr = ['000', '001', '011', '010', '110', '111', '101', '100']  # Gray
codes for 0, 1, 2, 3, 4, 5, 6, 7
k = 2
val, cost, delay = selection_network_gray(arr, k)
print(f"Value three bits: {val}, Cost: {cost}, Delay: {delay}")
```

והתוצאה:

```
Value two bits: 1, Cost: 3, Delay: 2
Value three bits: 1, Cost: 7, Delay: 3
```

ויזואליזציה





אכן ראינו בצורה זהה לרשת הבסיסית, שגם כאן העלות הינה לינארית והעומק לוגריתמי.

רשת בחירה עם קוד גריי ומכילה מטסטביליות

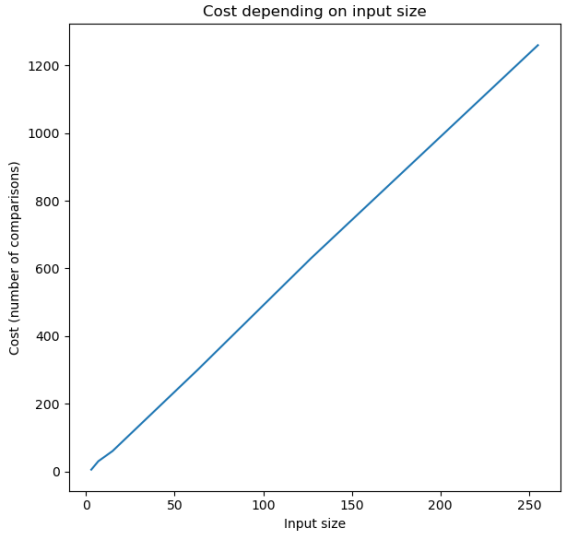
```
def gray_to_int(gray):
    binary = gray[0]
    for i in range(1, len(gray)):
        binary += str(int(binary[i - 1]) ^ int(gray[i]))
    #print(gray,int(binary, 2))
    return int(binary, 2)
def custom_compare(code1, code2):
    order = {'1': 3, 'M': 1, '0': 2} # Define the order of the bits
    # 1>0>M 0 bigger then M becuase if it will compare between
    # 000 to 00M so this not go her but if it compare
    # 01m to 010 so 010 bigger
    for bit1, bit2 in zip(code1, code2):
        if order[bit1] > order[bit2]:
            return 1
        elif order[bit1] < order[bit2]:
            return -1
    return 0

def compare_gray_codes(code1, code2):
    cost = 0
    if 'M' in code1 or 'M' in code2:
        codes1 = [code1.replace('M', '0'), code1.replace('M', '1')]
        codes2 = [code2.replace('M', '0'), code2.replace('M', '1')]
        cost += 2  # Cost for two replace operations
        max_code1 = max(codes1, key=gray_to_int)
        max_code2 = max(codes2, key=gray_to_int)
        cost += 2  # Cost for two max operations
    else:
        max_code1 = code1
        max_code2 = code2
    if gray_to_int(max_code1) > gray_to_int(max_code2):
        cost += 1  # Cost for comparison
        return 1, cost
    elif gray_to_int(max_code1) < gray_to_int(max_code2):
        cost += 1  # Cost for comparison
        return -1, cost
    else:
        cost += 1  # Cost for comparison
        custom_compare_result = custom_compare(code1, code2)
        cost += len(code1)  # Cost for custom comparison (worst case)
        return custom_compare_result, cost
def selection_network_gray(arr, k):
    n = len(arr)
```

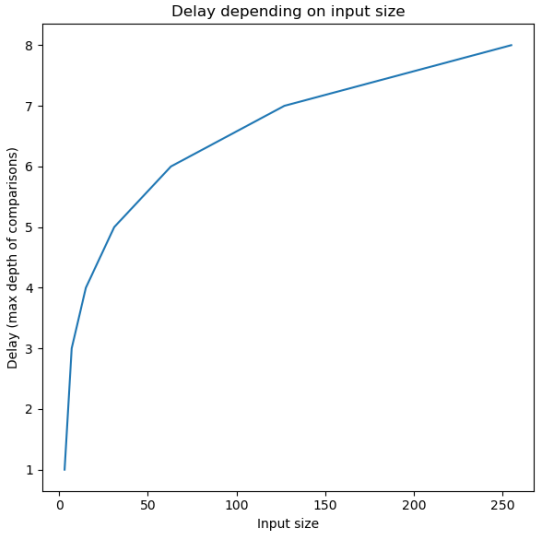
```
    if n == 1:
        return arr[0], 0, 0  # No comparators used for a single-element
array
    mid = n // 2
    left = arr[:mid]
    right = arr[mid:]
    cost = 0
    for i in range(mid):
        comparison, cost_temp = compare_gray_codes(left[i], right[i])
        cost += cost_temp  # Increment cost for each comparison
        if comparison > 0:
            left[i], right[i] = right[i], left[i]
    delay = 1  # At least one comparison was made
    if k <= mid:
        val, left_cost, left_delay = selection_network_gray(left, k)
        cost += left_cost
        delay = max(delay, left_delay + 1)  # Increment delay if we go
deeper
    else:
        val, right_cost, right_delay = selection_network_gray(right, k
- mid)
        cost += right_cost
        delay = max(delay, right_delay + 1)  # Increment delay if we go
deeper
    return val, cost, delay
arr = ['100','00M', '0M1','11M', '1M1', '10M', '01M', 'M10' ]  # Gray
codes for 0, 1, 2, 3, 4, 5, 6, 7
arr2 = ['0110', '0001','0011', '0111', '0100','1101', '1100', '0101']
k = 8
val, cost, delay = selection_network_gray(arr, k)
val2, cost2, delay2 = selection_network_gray(arr2, 1)
print(f"Value of {k} position: {val}, Cost: {cost}, Delay: {delay}")
print(f"Value of {1} position: {val2}, Cost: {cost2}, Delay: {delay2}")
arr = ['0M', 'M1', '1M']  # Gray codes for 0, 1, 2, 3
k = 2
val, cost, delay = selection_network_gray(arr, k)
print(f"Value: {val}, Cost: {cost}, Delay: {delay}")
```

והתוצאה:

```
Value of 8 position: 100, Cost: 38, Delay: 3
Value of 1 position: 0001, Cost: 7, Delay: 3
Value: M1, Cost: 10, Delay: 2
```

ויזואליזציה






```
Size: 3, Metastable Gray codes: ['0M', 'M1', '1M']
Cost: 5, Delay: 1
Size: 7, Metastable Gray codes: ['00M', '0M1', '01M', 'M10',
Cost: 25, Delay: 3
Size: 15, Metastable Gray codes: ['000M', '00M1', '001M', '0
Cost: 65, Delay: 4
Size: 31, Metastable Gray codes: ['0000M', '000M1', '0001M',
Cost: 150, Delay: 5
Size: 63, Metastable Gray codes: ['00000M', '0000M1', '00001
Cost: 300, Delay: 6
Size: 127, Metastable Gray codes: ['000000M', '00000M1', '00
Cost: 630, Delay: 7
Size: 255, Metastable Gray codes: ['0000000M', '000000M1', '
Cost: 1270, Delay: 8
```

על פי הגדלים אנו רואים שהגרף של העלות הוא בסדר גודל של nlogn ובצורה דומה העומק הוא גרף של logn.

בודק ביטוני ונספחים

```python
def custom_compare(code1, code2):
    order = {'1': 3, 'M': 1, '0': 2} # Define the order of the bits
    # 1>0>M 0 bigger then M becuase if it will compare between
    # 000 to 00M so this not go her but if it compare
    # 01m to 010 so 010 bigger
    for bit1, bit2 in zip(code1, code2):
        if order[bit1] > order[bit2]:
            return 1
```

```
        elif order[bit1] < order[bit2]:
            return -1
    return 0

def compare_gray_codes(code1, code2):
    # Generate two versions of each code if it contains 'M'
    codes1 = [code1.replace('M', '0'), code1.replace('M', '1')]
    codes2 = [code2.replace('M', '0'), code2.replace('M', '1')]
    # Compare the maximum of these two versions for each code
    max_code1 = max(codes1, key=gray_to_int)
    max_code2 = max(codes2, key=gray_to_int)
    # If one code is greater than the other, return 1 or -1 accordingly
    if gray_to_int(max_code1) > gray_to_int(max_code2):
        print(f"{max_code1} ({gray_to_int(max_code1)}) > {max_code2}
({gray_to_int(max_code2)})")
        return 1
    elif gray_to_int(max_code1) < gray_to_int(max_code2):
        print(f"{max_code1} ({gray_to_int(max_code1)}) < {max_code2}
({gray_to_int(max_code2)})")
        return -1
    # If both codes are equal, compare the original codes using the
custom comparison function
    else:
        return custom_compare(code1, code2)
def is_increasing(seq):
    return all(compare_gray_codes(x, y) <= 0 for x, y in zip(seq,
seq[1:]))

def is_decreasing(seq):
    return all(compare_gray_codes(x, y) >= 0 for x, y in zip(seq,
seq[1:]))

def is_bitonic(seq):
    if is_increasing(seq) or is_decreasing(seq):
        return True
    increasing = False
    changes = 0
    for i in range(len(seq) - 1):
        if compare_gray_codes(seq[i], seq[i+1]) < 0:
            if not increasing:
                increasing = True
                changes += 1
        elif compare_gray_codes(seq[i], seq[i+1]) > 0:
            if increasing:
                increasing = False
                changes += 1
        if changes > 2:
            return False
```

בס"ד

```
    return True
arr = [ '01M', '00M', '0M1', 'M10', '1M1', '100','10M', '11M']
# Test the function
is_bitonic(arr)
```

התוצאה:

```
True
```

ועבור

```
# SAME IDEA OF 1 4 2 3
arr = ['00M', '01M', '0M1', 'M10']
# Test the function
is_bitonic(arr)
```

נקבל:

```
False
```

בנוסף לכך בנות שונות שליצירת מערך ביטוני:

```
def make_bitonic(arr):
    n = len(arr)
    mid = n // 2

    # First half
    for i in range(mid - 1):
        if arr[i] > arr[i + 1]:
            arr[i], arr[i + 1] = arr[i + 1], arr[i]
        if arr[i + mid] < arr[i + mid + 1]:
            arr[i + mid], arr[i + mid + 1] = arr[i + mid + 1], arr[i +
mid]

    # Second half
    for i in range(mid):
        for j in range(i + 1, mid):
            if arr[i] > arr[j]:
                arr[i], arr[j] = arr[j], arr[i]
            if arr[i + mid] < arr[j + mid]:
                arr[i + mid], arr[j + mid] = arr[j + mid], arr[i + mid]

    return arr
arr = [30, 70, 40, 80, 60, 20, 10, 50]
sorted_arr = make_bitonic(arr)
print(sorted_arr)
```

```
[30, 40, 70, 80, 60, 50, 20, 10]
```

ובנייה מההרצאה למערך של 8 איברים

```
def bitonic_sequence(arr, compare):
    def swap_if_needed(a, b, asc=True):
        if (compare(a, b) > 0) == asc:
            return b, a
        return a, b

    # Pad the array if necessary
    original_length = len(arr)
    while len(arr) % 8 != 0:
        arr.append('0')

    # Split the array into two halves
    first_half = arr[:len(arr)//2]
    second_half = arr[len(arr)//2:]

    # For the first half, compare and swap a1 with a2 and a3 with a4
    first_half[0], first_half[1] = swap_if_needed(first_half[0],
first_half[1])
    first_half[2], first_half[3] = swap_if_needed(first_half[2],
first_half[3])

    # Then compare and swap a1 with a3 and a2 with a4
    first_half[0], first_half[2] = swap_if_needed(first_half[0],
first_half[2])
    first_half[1], first_half[3] = swap_if_needed(first_half[1],
first_half[3])

    # Finally, compare and swap a2 with a3
    first_half[1], first_half[2] = swap_if_needed(first_half[1],
first_half[2])

    # For the second half, perform the same comparisons and swaps but
in reverse order
    second_half[0], second_half[1] = swap_if_needed(second_half[0],
second_half[1], False)
    second_half[2], second_half[3] = swap_if_needed(second_half[2],
second_half[3], False)
    second_half[0], second_half[2] = swap_if_needed(second_half[0],
second_half[2], False)
    second_half[1], second_half[3] = swap_if_needed(second_half[1],
second_half[3], False)
```

```
    second_half[1], second_half[2] = swap_if_needed(second_half[1],
second_half[2], False)

    # Concatenate the two halves to form a bitonic sequence
    bitonic_seq = first_half + second_half

    # Remove the padding elements from the end of the array
    bitonic_seq = bitonic_seq[:original_length]

    return bitonic_seq

arr = ['100','00M', '0M1','11M', '1M1', '10M', '01M', 'M10' ]
print(bitonic_sequence(arr, compare_gray_codes))  # Output depends on
the implementation of gray_to_int and compare_gray_codes
```

והתוצאה:

```
['00M', '0M1', '11M', '100', '10M', '1M1', 'M10', '01M']
```