**CS 202: Design and Analysis of Algorithms** March 17, 2021

# DAA Theory Project

IMT2019003 - Aditya Vardhan
IMT2019012 - Archit Sangal
IMT2019087 - Supreeth Varadarajan

## 1 Question

We have n threads, arranged one above the other, with each one having a bead. The initial positions of each bead is given. We want to line up all the beads vertically by moving them along the threads. The objective is to minimize the total distance moved. Given n numbers denoting the positions of beads, design a linear time algorithm to find the minimum total distance the beads have to be moved in order to line them up.

## 2 Overview

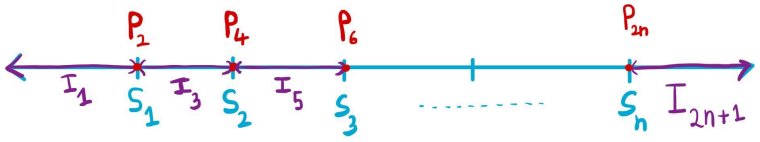
Suppose we represent position of beads with positive real numbers. In this case let the position of beads be given by - $s_1, s_2, \ldots, s_n$.

$$\text{So we need to } \textbf{minimize} \rightarrow \sum_{i=0}^{n} |s_i - x| \qquad \ldots (i)$$

So now we need to return $x$ such that the above condition is satisfies.

## 3 Mathematical Analysis

$$\text{Assuming } s_1 \leq s_2 \leq \ldots \leq s_n \text{ where } s_1, s_2, \ldots, s_n \in S$$
$$\text{Let } y = \sum_{i=1}^{n} |s_i - x|$$




Consider the following intervals $I_{2i+1}$ and points $P_{2i}$, defined as follows -

$I_1 \Rightarrow x \in (-\infty, s_1)$

$P_2 \Rightarrow x = s_1$

$I_3 \Rightarrow x \in (s_1, s_2)$

$P_4 \Rightarrow x = s_2$

$I_5 \Rightarrow x \in (s_2, s_3)$

$\vdots$

$P_{2n} \Rightarrow x = s_n$

$I_{2n+1} \Rightarrow x \in (s_n, \infty)$

Also let $\exists k$ *s.t.* $k \geq 0$
Let $x \in I_1$, say $x = x_1$ *s.t.* $x_1 + k$ also belong to $I_1$. Also, let $y(x_1) = Y_1$.

Hence, $y(x_1 + k) = Y_1 - kn$ as $|s_i - x|$ are nothing but distances between point $s_i$ and $x$. As $k$ is positive, consider it as we are moving towards right on the real axis in the figure shown above. So, when we move a distance $k$ in interval $I_1$ in positive $x$ direction, all the distances reduce by $k$.

$\Rightarrow y$ decreases as $x$ increases for $x \leq s_1$ and $n \geq 1$

Similarly, let $x \in I_3$, say $x = x_3$ *s.t.* $x_3 + k$ also belong to $I_3$. Also, let $y(x_3) = Y_3$.

Hence,

$$\begin{aligned} y(x_3 + k) &= Y_3 - k(n-1) + k \\ y(x_3 + k) &= Y_3 - kn + 2k \\ y(x_3 + k) &= Y_3 - k(n-2) \end{aligned}$$

as when we move a distance $k$ in positive $x$ direction in interval $I_3$, distance between $n-1$ points decrease by $k$ while distance from $s_1$ increase by $k$.

$\Rightarrow y$ decreases as $x$ increases for $s_1 \leq x \leq s_2$ and $n > 2$

For $x \in I_{2i+1}$, let $x = x_{2i+1}$ *s.t.* $x_{2i+1} + k$ also belong to $I_{2i+1}$ -
Let $y(x_{2i+1}) = Y_{2i+1}$.

$$\begin{aligned} y(s_{2i+1} + k) &= Y_{2i+1} - k(n-i) + ki \\ &= Y_{2i+1} - kn + k(2i) \\ &= Y_{2i+1} - k(n-2i) \end{aligned}$$

If $n - 2i > 0$, we would want to maximise $k$ so that $y$ becomes minimum. Therefore, we move in positive $x$ direction.

If $n - 2i < 0$, we would want to minimise $k$ and $x_{2i+1}$ so that $y$ becomes minimum. Therefore, we move in negative $x$ direction.

If $n - 2i = 0$, it will not matter in which ever direction we move in that interval. So if this condition is true any point between $s_i$ and $s_{i+1}$ will give us the same $y$; where $i = \frac{n}{2}$(follows from the very condition).

The last condition is true only when $n$ is even. So we can conclude the following if last condition is true -

$$x_1 \geq x \leq x_2; \text{ where } x_1 \in y_{I_{2i-1}}, x \in y_{I_{2i+1}} \text{ and } x_2 \in y_{I_{2i+3}}$$

But as mentioned earlier there may not exist a case where $n - 2i = 0$ i.e. when $n$ is odd (as $i \in I$). For such cases consider points $P_2, P_4, \ldots, P_{2n}$.

Let $\exists k$ *s.t.* $k > 0$. And $x \in P_2$, i.e. $x = x_2$. As we calculated distance previously, we will calculate the distance in vicinity of point $P_{2i}$. Here, let $y(x_2) = Y_2$

$$\begin{aligned} y(x_2 - k) &= Y_2 + kn \quad \text{(and)} \\ y(x_2 + k) &= Y_2 - k(n-1) + k \\ y(x_2 + k) &= Y_2 - k(n-2) \end{aligned}$$

$$\Rightarrow y(x_2 + k) < y(x_2 - k) \text{ for } n > 2$$

Similarly, let $\exists k$ *s.t.* $k > 0$. And $x \in P_4$, i.e. $x = x_4$. Here, let $y(x_4) = Y_4$. Hence,

$$
\begin{aligned}
y(x_4 - k) &= Y_4 + k(n-1) - k \\
&= Y_4 + k(n-2) \quad \text{(and)} \\
y(x_4 + k) &= Y_4 - k(n-2) + 2k \\
y(x_4 + k) &= Y_4 - k(n-4)
\end{aligned}
$$

$$\Rightarrow y(x_4 + k) < y(x_4 - k) \text{ for } n > 4$$

Similarly, let $\exists k \; s.t. \; k > 0$. And $x \in P_6$, i.e. $x = x_6$. Here, let $y(x_6) = Y_6$. Hence,

$$
\begin{aligned}
y(x_6 - k) &= Y_6 + k(n-2) - 2k \\
&= Y_6 + k(n-4) \quad \text{(and)} \\
y(x_6 + k) &= Y_6 - k(n-3) + 3k \\
y(x_6 + k) &= Y_6 - k(n-6)
\end{aligned}
$$

$$\Rightarrow y(x_6 + k) < y(x_6 - k) \text{ for } n > 6$$

For $y(x_{2i}) = Y_{2i}$, we can conclude -

$$
\begin{aligned}
y(x_{2i} - k) &= Y_{2i} + k(n - 2(i-1)) \\
y(x_{2i} + k) &= Y_{2i} - k(n - 2i)
\end{aligned}
$$

For $y$ to be minimum, it must satisfy the following conditions-

$$
\begin{aligned}
& y(x_{2i}) \leq y(x_{2i} - k) \\
\Rightarrow\ & y(x_{2i}) \leq Y_{2i} + k(n - 2(i-1)) \\
\Rightarrow\ & Y_{2i} \leq Y_{2i} + k(n - 2(i-1)) \\
\Rightarrow\ & 0 \leq n - 2(i-1) \\
\Rightarrow\ & i \leq \frac{n}{2} + 1 \quad \text{(and)} \\
& y(x_{2i}) \leq y(x_{2i} + k) \\
\Rightarrow\ & y(x_{2i}) \leq Y_{2i} - k(n - 2i) \\
\Rightarrow\ & Y_{2i} \leq Y_{2i} - k(n - 2i) \\
\Rightarrow\ & 0 \leq -k(n - 2i) \\
\Rightarrow\ & 0 \geq n - 2i \\
\Rightarrow\ & i \geq \frac{n}{2}
\end{aligned}
$$

Hence, $\frac{n}{2} \leq i \leq \frac{n}{2} + 1$. We know that $n$ is odd and $i \in I$.
$\Rightarrow i = \frac{n+1}{2}$; when $n$ is odd

Hence, is proved that $y = \sum_{i=1}^{n} |s_i - x|$ is minimum when $x$ is median of set $S$, ($s_1 \leq s_2 \leq \ldots \leq s_n$ where $s_1, s_2, \ldots, s_n \in S$).

Without loss of generality above proof is true and can be used for $s_i$'s if there are any repetitions.

# 4 Algorithm

Given an array $A[\,]$ of size $n$, we need to find:

$$\frac{n+1}{2}\text{th smallest element, if } n \text{ is odd}$$

$$\frac{n}{2}\text{th smallest element, if } n \text{ is even}$$

So, we'll give a general algorithm which finds the $ith$ smallest number in $A[\,]$, where $i$ can be any natural number from 1 to $n$.

**NOTE:**

- In the algorithm below, whenever we mention "median" for an array of size say $m$, we're trying to imply the $\lfloor \frac{m+1}{2} \rfloor$th smallest element in the array.
- For simplicity and clarity, we'll assume the numbers to be distinct for now. We'll see the modifications needed for the case when numbers are repeated, later on.

**Following is the detailed algorithm**

1. Divide array $A[\,]$ (of size $n$) into groups of 5 each.

   If $n \,\%\, 5 = 0$

   $$\text{There will be exactly } \frac{n}{5} \text{ groups of size 5 each}$$

   Else

   $$\text{There will be an additional group of size } n \,\%\, 5$$

   So, it is safe to say, in general, that we'll have $\lceil \frac{n}{5} \rceil$ groups.

2. Find the median of each of these $\lceil \frac{n}{5} \rceil$ groups by sorting (merge-sort, bubble-sort etc.)

3. Recursively perform the above steps until you find the median of the already found medians.

   Let this "median of medians" be $x$

4. Partition $A[\,]$ using $x$ as the pivot element. Let us say, after partitioning, we've $(k-1)$ elements lying on the left side of the partition i.e $(k-1)$ elements in the array are smaller than $x$

5. We've 3 cases

   (a) if $i = k$ We've $x$ as our answer

   (b) if $i < k$ Recursively perform the above steps to find the $i$th smallest element in the left side of the partition

   (c) if $i > k$ Recursively perform the above steps to find the $(i-k)$th smallest element in the right side of the partition

# 5 Order Calculation

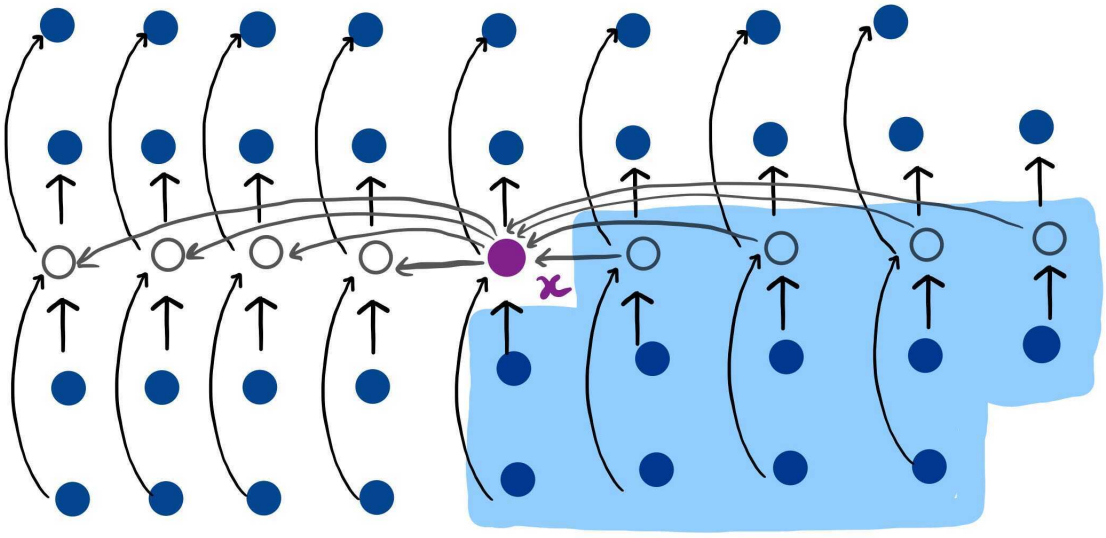
Let our algorithm take a total time of $T(n)$. The recurrence relation that we're going to build up will have 3 components to it:

1. Steps 1, 2 and 4 of our algorithm take $O(n)$ time. In step 2, we make $O(n)$ calls to sorting, for constant sized arrays(of size 5 or less) i.e. $O(1)$ sized arrays.

   A bit more careful analysis easily tells us that they are actually $\Omega(n)$ as well.

2. Step 3 takes $T(\lceil \frac{n}{5} \rceil)$ time

3. Now, to find the time for step 5, a bit deeper analysis is needed.
   Consider the given figure.




- The $n$ elements are represented by circles.
- Each of the $\lceil \frac{n}{5} \rceil$ groups occupies a column
- Medians of those $\lceil \frac{n}{5} \rceil$ groups are in white.
- Arrows go from larger to smaller elements. The arrows, if seen carefully denote a transitive relation in the sense that if there is an arrow from node $a$ to $b$ (implying $a > b$) and an arrow from node $b$ to $c$ (implying $b > c$), then there lies an invisible arrow from $a$ to $c$ (implying $a > c$)
- As can be seen from the figure, 3 out of every full group of 5 elements to the right of $x$ are greater than x, and 3 out of every group of 5 elements to the left are smaller than $x$. The element that are known to be greater than $x$ appear on the shaded background.

Coming back to analyzing step 5.
At least half of the medians computed in step 2 are greater than or equal to the median-of-medians $x$

$\rightarrow$ At least half of the $\lceil \frac{n}{5} \rceil$ groups contribute atleast 3 elements that are greater than $x$, except for the one group that has fewer than 5 elements (if $n \,\%\, 5 \neq 0$) and the group containing $x$ itself.

Ignoring these two groups, we've number of elements greater than $x$ to be at least equal to

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

Similarly, at least $(\frac{3n}{10} - 6)$ elements are less than $x$.

$$\rightarrow \text{in worst case, step 5 calls its previous steps recursively upon at most } \left(\frac{7n}{10} + 6\right) \text{ elements}$$

$$\rightarrow \text{step 5 takes at most } T\left(\frac{7n}{10} + 6\right) \text{ time}$$

Hence, we come to our final recurrence relation given as

$$T(n) \leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + O(n)$$

We'll compute the running time using the substitution method.

We assume that $T(p) \leq cp$ for some suitable $c, \forall p < n$ We also pick $a$ such that the $O(n)$ term is bounded above by $an, \forall n > 0$

$$
\begin{aligned}
T(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + an \\
&\leq cn/5 + c + 7cn/10 + 6c + an \\
&= 9cn/10 + 7c + an \\
&= n(9c/10 + a) + 7c \\
\rightarrow T(n) &= O(n)
\end{aligned} \tag{1}
$$

Considering the lower bound for $T(n)$, we already know:

$$
T(n) \geq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{3n}{10} - 6\right) + \Omega(n)
$$

We assume that $T(p) \geq cp$ for some suitable $c, \forall p < n$ We also pick $a$ such that the $\Omega(n)$ term is bounded below by $an, \forall n > 0$

$$
\begin{aligned}
T(n) &\geq c\lceil n/5 \rceil + c(3n/10 - 6) + an \\
&\geq cn/5 + 3cn/10 - 6c + an \\
&= 5cn/10 - 6c + an \\
&= n(5c/10 + a) - 6c \\
\rightarrow T(n) &= \Omega(n)
\end{aligned} \tag{2}
$$

From (1) and (2), we can say that $T(n) = \Theta(n)$

# 6 Modifications for repeated elements case

1. In the step 4 of our above stated algorithm, instead of using a simple partition, we'll use a 3-way partition. After partitioning, this 3-way partition will return 2 indices $p$ and $q$ $(1 \leq p, q \leq n)$ such that $\forall a_j \in A[\,]$:

$$
\begin{aligned}
a_j &< x, \quad \forall j \in [1, p) \\
a_j &= x, \quad \forall j \in [p, q] \\
a_j &> x, \quad \forall j \in (q, n]
\end{aligned}
$$

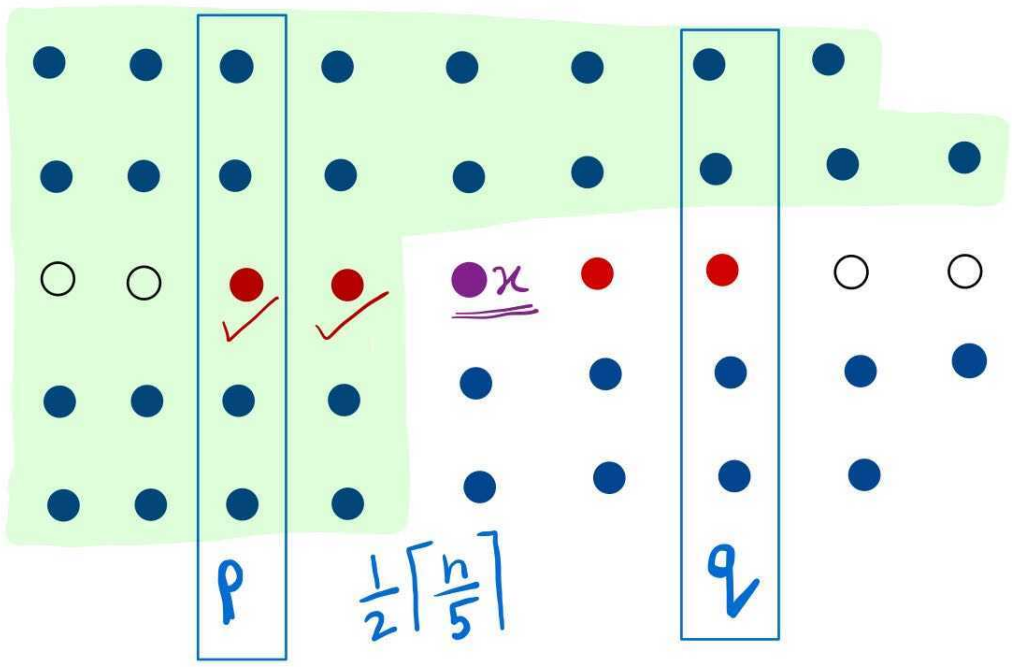
   Note that if $x$ is not repeated then the 3-way partition returns $p = q$ i.e. it behaves like a simple partition.

2. The step 5 modifies to the following:
   (a) if $p \leq i \leq q$ : $x$ is our answer
   (b) if $i < p$ : recurse for the $i$th smallest in $A[1 \cdots p-1]$
   (c) if $i > q$ : recurse for the $(i-q)$th smallest in $A[q+1 \cdots n]$

### Time complexity Analysis if elements are repeated

The run time analysis of steps 1,2,3 and 4 remain unchanged. The only doubt that can arise, is for the run time of step 5.

Now, the upper bound for step 5 is still $T(\frac{7n}{10})$, as can be seen from the figure below, there can be only some constant number of points removed from the initial maximum set of points that were needed to recused upon (in distinct elements case). Hence $T(n)$ is still $O(n)$.




The lower bound for step 5 may be changed however, if there are too many repeated elements and the partition that we need to recurse upon is $O(1)$ size. But this doesn't change the lower bound for runtime of the overall algorithm as runtime for other steps remain unchanged. Thus, T(n) is still $\Omega(n)$

$$\rightarrow T(n) \text{ is still } \Theta(n)$$

# 7 Pseudocode

---

```
# returns kth smallest element in arr[] between l and r.
def kthSmallest(arr, l, r, k):

    median = []
    i = 0
    while (i < n // 5):
        median.append(findMedian(arr, l + i * 5, 5))
    # for elements not multiple of 5
    if (i * 5 < n):
        median.append(findMedian(arr, l + i * 5, n % 5))

    # get the median of median[]
    if i == 1:
        medOfMed = median[i - 1]
    else:
        medOfMed = kthSmallest(median, 0, i - 1, i // 2)

    # 3-way partition array around medOfMed
    pos = partition(arr, l, r, medOfMed)

    # Cases for pos and k
    if (pos[0] - l <= k - 1 and k - 1 <= pos[1] - l):
        return arr[pos[0]]
    if (pos[0] - l > k - 1):
        return kthSmallest(arr, l, pos[0] - 1, k)
    return kthSmallest(arr, pos[1] + 1, r, k - pos[1] + l - 1)
```

```
# finds median in O(1)
def findMedian(arr, l, n):

    for i in range(l, l + n):
        lis.append(arr[i])
    lis.sort()
    return lis[n // 2]
```

# 8 Proof Of Correctness

As we have seen in the mathematical analysis, the point which is at the shortest distance from all the beads is the median of the positions of the beads. Here, the algorithm used to find the median is the algorithm for finding the k-th largest element in the array. Therefore if we can prove the correctness of this algorithm, then will imply that the algorithm used is correct.

Now, for proving this algorithm, we can use strong induction.

1. **Base case:** n = 1. Here, k can only take the value of 0 and the algorithm will give a correct answer for $k = 0$.
2. **Inductive Hypothesis:** For all valid values of k, let this algorithm give a correct answer for all values of n less than or equal to some m.
3. **Induction step:** Now, for proving the algorithm for $n = m + 1$, let us consider some arbitrary value for k between 0 and m+1. Let the array's upper and lower bounds be $l$ and $r$ respectively. At the starting of the algorithm, we need to select any element in the array(say $x$) and pass it in the partition function. But for reducing the time complexity of the algorithm as explained in the Order Calculation part, we will group the array into parts of 5, find the medians of those arrays and find the median of those medians. This median of medians is selected to be $x$.

   Now this value x is passed in the partition function. The partition function gives the minimum and maximum indices of the element x in the sorted array. Also, it moves all the elements less than x to the left of x and vice versa.

   After the above step, we get the minimum and maximum indices of the element x in the sorted array. Let us call these $n_1$ and $n_2$ ($n_1 \leq n_2$). Now, the algorithm takes 3 cases for the indices $n_1$ and $n_2$.

   - **Case 1:** $n_1 \leq k \leq n_2$. This means that the k-th element occurs between the indices $n_1$ and $n_2$. But all the elements between $n_1$ and $n_2$ takes the value array[$n_1$]. Therefore, array[$n_1$] is the required answer so the algorithm return it.
   - **Case 2:** $k < n_1$. This means that the k-th element is present before $n_1$. So, the k-th element is smaller than x. But all the elements smaller than x are present before the index $n_1$ as a result of the partition function. So, we need to search only in the part of the array before $n_1$. Therefore, we can just apply this algorithm for the k-th element in the array between the indices $l$ to $n_1 - 1$. But this is an array of size less than or equal to m. Therefore, the algorithm gives a correct solution for this array.
   - **Case 3:** $k > n_2$. This means that the k-th element is present after $n_2$. So, the k-th element is larger than x. But all the elements larger than x are present after the index $n_2$ as a result of the partition function. So, we need to search only in the part of the array after $n_2$. Therefore, we can just apply this algorithm for the $k - (n_2 - l + 1)$-th element in the array between the indices $n_2 + 1$ to $r$. But this is an array of size less than or equal to m. Therefore, the algorithm gives a correct solution for this array.

   This implies that for some random valid k, this algorithm takes gives a correct solution for the array of size m+1. But we know that k is arbitrary. So, we can say for all valid k that this algorithm gives the correct solution for an array of size $m + 1$. Therefore, by the principle of strong induction, we can imply that this algorithm gives a correct solution for all n.

Now that we know the algorithm for finding the k-th element in an array is correct, we apply this algorithm to find the median of the array and then find the total distance of all beads from the median. Therefore, we can imply that this algorithm is correct.

# 9 Code

The code for this problem is written in python. The input is given in the list PosOfBeads and the output is printed in the console.

---

```
# returns kth smallest element in arr[] between l and r.
def kthSmallest(arr, l, r, k):
    # checking for valid k
    if (k > 0 and k <= r - l + 1):
        n = r - l + 1

        # Divide arr[] in groups of size 5, calculate median of every group and store it in median[].
        median = []
        i = 0
        while (i < n // 5):
            median.append(findMedian(arr, l + i * 5, 5))
            i += 1
        if (i * 5 < n):
            median.append(findMedian(arr, l + i * 5, n % 5))
            i += 1

        # get the median of median[]
        if i == 1:
            medOfMed = median[i - 1]
        else:
            medOfMed = kthSmallest(median, 0, i - 1, i // 2)

        # partition array around medOfMed
        pos = partition(arr, l, r, medOfMed)

        # Cases for pos and k
        if (pos[0] - l <= k - 1 and k - 1 <= pos[1] - l):
            return arr[pos[0]]
        if (pos[0] - l > k - 1):
            return kthSmallest(arr, l, pos[0] - 1, k)
        return kthSmallest(arr, pos[1] + 1, r, k - pos[1] + l - 1)

    # If k is invalid
    return 999999999999

# swaps elements in positions a and b
def swap(arr, a, b):
    temp = arr[a]
    arr[a] = arr[b]
    arr[b] = temp

# partitions around x and returns its position range.
def partition(arr, l, r, x):
    for i in range(l, r+1):
        if arr[i] == x:
            swap(arr, l, i)
            break
    x = arr[l]
    i = l
    for j in range(l+1, r+1):
        if (arr[j] <= x):
            i += 1
```

```
            swap(arr, i, j)
    m1 = i
    m2 = i
    k = l
    for j in range(l+1, i+1):
        if(arr[j] < x):
            k += 1
            swap(arr, j, k)
        else:
            m1 -= 1
    swap(arr, l, k)
    return (m1, m2)

# finds median
def findMedian(arr, l, n):
    lis = []
    for i in range(l, l + n):
        lis.append(arr[i])
    lis.sort()
    return lis[n // 2]

# Driver Code
if __name__ == '__main__':
    PosOfBeads = [12, 3, 5, 7, 5, 5, 26, 9, 4, 7, 8, 3, 9, 6, 3, 13, 10, 2, 0] # positions of beads
    n = len(PosOfBeads)
    med = kthSmallest(PosOfBeads, 0, n - 1, n // 2 + 1) # get the median
    s = 0
    for p in PosOfBeads:
        s += abs(p - med) # add distances
    print(s)
```

---

# 10 References


Idea was derived from the book -

**Introduction to Algorithms by Thomas H. Cormen (CLRS)**