Principled Programming
Introduction to Coding in Any Imperative Language

Tim Teitelbaum
Emeritus Professor
Department of Computer Science
Cornell University

Median
The median of an ordered array of \( n \) values is the middle value. If \( n \) is odd, this is \( A[n/2] \); if \( n \) is even, we also opt for \( A[n/2] \) rather than averaging the middle two values.
The median of an ordered array of \( n \) values is the middle value. If \( n \) is odd, this is \( A[n/2] \); if \( n \) is even, we also opt for \( A[n/2] \) rather than averaging the middle two values.

But what if the array is not ordered. How would you find the median then?

You could sort the array and select \( A[n/2] \). But sorting requires \( n \log n \) operations.

Is it possible to do better? Try it. You will find that everyday experience is no help.
The median of an ordered array of $n$ values is the middle value. If $n$ is odd, this is $A[n/2]$; if $n$ is even, we also opt for $A[n/2]$ rather than averaging the middle two values.

But what if the array is not ordered. How would you find the median then?

You could sort the array and select $A[n/2]$. But sorting requires $n \log n$ operations.

Is it possible to do better? Try it. You will find that everyday experience is no help.

We need principles to follow in such cases.
Three principles that can help are:

- Consider generalizing a problem when designing an algorithm.
- Consider Divide and Conquer when designing an algorithm.
- Consider recursion when designing an algorithm.

We will use them to derive:

- An Average-Case Linear-Time Median Algorithm
- A Worst-Case Linear-Time Median Algorithm

It is astounding that it is possible to find the median of an unordered array of length $n$ in linear time, i.e., time proportional to $n$. 
The median of an ordered array of $n$ values is the middle value. If $n$ is odd, this is $A[n/2]$; if $n$ is even, we opt for $A[n/2]$ rather than averaging the middle two values.

Consider generalizing a problem when designing an algorithm.

**Selection:** Given a set of $n$ rank-ordered values, select the $j^{th}$ smallest value of the set.
Selection: Given a set of \( n \) rank-ordered values, select the \( j^{th} \) smallest value of the set.

Consider Divide and Conquer when designing an algorithm.

Recall: Partitioning, based on the Dutch National Flag problem, for some pivot \( p \):

\[
\begin{array}{c|c|c|c}
0 & <p & ==p & >p \\
\hline
\end{array}
\]

\( 0 \leq j < w \). The \( j^{th} \) smallest value is the \( j^{th} \) smallest value of \( A[0..w-1] \)
\( w \leq j < b \). The \( j^{th} \) smallest value is the pivot, \( p \)
\( b \leq j < n \). The \( j^{th} \) smallest value is the \( (j-b)^{th} \) smallest value in \( A[b..n-1] \)

Choose one of the three regions based on a Partition (Divide) and repeat (Conquer).
Start with the code for Partition, and morph it into QuickSelect:

```c
/* Rearrange A[L..R-1] into all <p, then all ==p, then all >p. */
static void Partition( int A[], int L, int R, int p ) {
  (body of Partition)
} /* Partition */
```

Don’t type if you can avoid it; clone. Cut and paste, then adapt.
Start with the code for Partition, and morph it into QuickSelect:

/* Given int 0≤j<n, return j-th smallest in A[0..n-1]. */
static void QuickSelect( int A[], int L, int R, int p ) {
  ⟨body of Partition⟩
} /* Partition */
Start with the code for Partition, and morph it into Select:

```c
/* Given int 0≤j<n, return j-th smallest in A[0..n-1]. */
static void QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    int p = /* value of pivot */ ;
    (body of Partition)
} /* QuickSelect */
```

☞ Don’t type if you can avoid it; clone. Cut and paste, then adapt.

Move parameters L, R, and p into the body of QuickSelect, and introduce parameters n and j.
Start with the code for Partition, and morph it into Select:

/* Given int 0\leq j < n, return j-th smallest in A[0..n-1]. */
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    int p = /* value of pivot */ ;
    (body of Partition)
    return ____;
} /* QuickSelect */

☞ Don’t type if you can avoid it; clone. Cut and paste, then adapt.

Change return type to int, and introduce a return statement for the result.
Could consider recursion, but it is not needed because we can just ...

```c
/* Given int 0 ≤ j < n, return j-th smallest in A[0..n-1]. */
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    int p = /* value of pivot */ ;
    (body of Partition)
    return ____;
} /* QuickSelect */
```
Update L, R, and p iteratively using the IN Variant shown.

/* Given int 0≤j<n, return j-th smallest in A[0..n-1]. */
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    int p = /* value of pivot */ ;
    ⟨body of Partition⟩
    return ____;
} /* QuickSelect */

/* Initialize. */
while ( /* not finished */ ) {
    /* Compute. */
    /* Go on to next. */
}
Update L, R, and p iteratively using the INARIANT shown.

/* Given int 0≤j<n, return j-th smallest in A[0..n-1]. */
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    while ( /* not finished */ ) {
        int p = /* value of pivot */ ;
        ⟨body of Partition⟩
        /* Go on to next. */
    }
    return ____;
} /* QuickSelect */
Update \( L \), \( R \), and \( p \) iteratively using the **INARIANT** shown.

/* Given \( 0 \leq j < n \), return \( j \)-th smallest in \( A[0..n-1] \). */
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    while ( /* not finished */ ) {
        int p = /* value of pivot */ ;
        ⟨body of Partition⟩
        /* Go on to “\(<p\)” or “\(>p\)” region if \( j \)-th smallest there; else return \( p \). */
    }
    return ____;
} /* QuickSelect */
Update L, R, and p iteratively using the INARIANT shown.

/* Given int 0≤j<n, return j-th smallest in A[0..n-1]. */
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    while ( /* not finished */ ) {
        int p = /* value of pivot */ ;
        ⟨body of Partition⟩
        /* Go on to "<p" or ">p" region if j-th smallest there; else return p. */
        if ( j<w ) R = w;
        else if ( j<b ) return p;
        else L = b;
    }
    return ___;
} /* QuickSelect */
Update L, R, and p iteratively using the **INARIANT** shown.

/* Given int 0≤j<n, return j-th smallest in A[0..n-1]. */
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    while ( R-L > 1 ) {
        int p = /* value of pivot */ ;
        ⟨body of Partition⟩
        /* Go on to “<p” or “>p” region if j-th smallest there; else return p. */
        if ( j<w ) R = w;
        else if ( j<b ) return p;
        else L = b;
    }
    return A[j];
} /* QuickSelect */
Update $L$, $R$, and $p$ iteratively using the IN Variant shown.

```c
/* Given int $0 \leq j < n$, return $j$-th smallest in $A[0..n-1]$. */
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    while ( R-L > 1 ) {
        int p = /* value of pivot */;
        ⟨body of Partition⟩
        /* Go on to "<p" or ">p" region if $j$-th smallest there; else return $p$. */
        if ( j<w ) R = w;
        else if ( j<b ) return p;
        else L = b;
    }
    return A[j];
} /* QuickSelect */
```

Q. Where was $j$ ever updated?
A. Nowhere. Partitioning moved values so the $j^{th}$ smallest ended up in $A[j]$. 
Update L, R, and p iteratively using the **INVARIANT** shown.

/* Given int 0≤j<n, return j-th smallest in A[0..n-1]. */
static int QuickSelect( int A[], int n, int j ) {
    int L = 0; int R = n;
    while ( R-L > 1 ) {
        ⟨body of Partition⟩
        /* Go on to “<p” or “>p” region if j-th smallest there; else return p. */
        if ( j<w ) R = w;
        else if ( j<b ) return p;
        else L = b;
    }
    return A[j];
} /* QuickSelect */

- **Best case.** On each iteration, pivot is (serendipitously) the **median** of \(A[L..R-1]\), so region sizes reduced by \(\frac{1}{2}\). Partitioning time is linear in size.
  
  Total effort. \(1 \cdot n + \frac{1}{2} \cdot n + \frac{1}{4} \cdot n + \ldots = 2 \cdot n\), i.e., **linear in \(n\)**

- **Worst case.** On each iteration, pivot is (serendipitously) the **min or max** of \(A[L..R-1]\), so region sizes reduced by 1. Partitioning time is linear in size.
  
  Total effort. \(n + (n-1) + (n-2) + \ldots + 1 = n \cdot (n-1)/2\), i.e., **quadratic in \(n\)**.

- **Average case,** i.e., summed over all permutations of values in \(A[0..n-1]\).
  
  Total effort. **Linear in \(n\)**

*(offered without proof)*
Bad News: QuickSelect can have quadratic-time performance on some arrays.

Imagine telling the widow:

But Mrs. Jones, on average the code would have been fast enough to have saved your husband’s life.

Goal. Linear-time performance on every array.
Performance Goal: Pivots computed as ________ in the hope that

- Every case.
  (1) On each iteration, region sizes reduced by constant ratio $r$.
      Partitioning time is linear in region size.
      Total effort for partitioning. $1\cdot n + r\cdot n + r^2\cdot n + r^3\cdot n + \ldots = n/(1-r)$
      i.e., linear in $n$, not counting time to compute the pivot.
  (2) On each iteration, the cost to compute the pivot is also linear in region size.

Thus, total effort, would be linear in $n$. In particular, even in the worst-case.

- Every case.
  
  (1) On each iteration, region sizes reduced by constant ratio $r$.
      Partitioning time is linear in region size.
      Total effort for partitioning. $1 \cdot n + r \cdot n + r^2 \cdot n + r^3 \cdot n + ... = n/(1-r)$
      I.e., linear in $n$, not counting time to compute the pivot.
  
  (2) On each iteration, the cost to compute the pivot is also linear in region size.

Thus, total effort, would be linear in $n$. In particular, even in the worst-case.
Idea: Pivots computed as approximations to median of A[L..R-1].

Imagine that this array, with median 61:

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\
51 & 60 & 73 & 92 & 57 & 54 & 75 & 59 & 91 & 58 & 71 & 62 & 67 & 66 & 59 & 52 & 61 & 72 & 55 & 60 & 79 \\
\end{array}
\]

were laid out in a 3-high 2-D array in row major order:

\[
\begin{array}{cccccccccccc}
51 & 60 & 73 & 92 & 57 & 54 & 75 \\
59 & 91 & 58 & 71 & 62 & 67 & 66 \\
59 & 52 & 61 & 72 & 55 & 60 & 79 \\
\end{array}
\]

The median of each column is shown in red.
**Idea:** Pivots computed as approximations to median of $A[\text{L}..\text{R}-1]$.

Imagine that this array, with median 61:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>51</td>
<td>60</td>
<td>73</td>
<td>92</td>
<td>57</td>
<td>54</td>
<td>75</td>
<td>59</td>
<td>91</td>
<td>58</td>
<td>71</td>
<td>62</td>
<td>67</td>
<td>66</td>
<td>59</td>
<td>52</td>
<td>61</td>
<td>72</td>
<td>55</td>
<td>60</td>
<td>79</td>
</tr>
</tbody>
</table>

were laid out in a 3-high 2-D array in row major order:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 51| 52| 58| 71 |55 |54 |66 |
| 59| 60| 61 |72 |57 |60 |75 |
| 59| 91| 73 |92 |62 |67 |79 |

Now, imagine that each column were sorted, so its median comes to middle row.

The median of each column is shown in red.
**Idea:** Pivots computed as approximations to median of $A[L..R-1]$.

Imagine that this array, with median 61:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>51</td>
<td>60</td>
<td>73</td>
<td>92</td>
<td>57</td>
<td>54</td>
<td>75</td>
<td>59</td>
<td>91</td>
<td>58</td>
<td>71</td>
<td>62</td>
<td>67</td>
<td>66</td>
<td>59</td>
<td>52</td>
<td>61</td>
<td>72</td>
<td>55</td>
<td>60</td>
<td>79</td>
</tr>
</tbody>
</table>

were laid out in a 3-high 2-D array in row major order:

<table>
<thead>
<tr>
<th></th>
<th>55</th>
<th>51</th>
<th>52</th>
<th>54</th>
<th>58</th>
<th>66</th>
<th>71</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>57</td>
<td>59</td>
<td>60</td>
<td>60</td>
<td>61</td>
<td>75</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>62</td>
<td>59</td>
<td>91</td>
<td>67</td>
<td>73</td>
<td>79</td>
<td>92</td>
</tr>
</tbody>
</table>

Next, imagine that the columns were sorted by their medians. The median of the medians is shown with a green background.

The median of each column is shown in red.
**Idea:** Pivots computed as approximations to median of $A[L \ldots R-1]$.

Imagine that this array, with median 61:

<table>
<thead>
<tr>
<th>A</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>60</td>
<td>73</td>
<td>92</td>
<td>57</td>
<td>54</td>
<td>75</td>
<td>59</td>
<td>91</td>
<td>58</td>
<td>71</td>
<td>62</td>
<td>67</td>
<td>66</td>
<td>59</td>
<td>52</td>
<td>61</td>
<td>72</td>
<td>55</td>
<td>60</td>
<td>79</td>
<td></td>
</tr>
</tbody>
</table>

were laid out in a 3-high 2-D array in row major order:

|     | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 55  | 51 | 52 | 54 | 58 | 66 | 71 | 57 | 59 | 60 | 60 | 61 | 75 | 72 | 62 | 59 | 91 | 67 | 73 | 79 | 92 |

Finally, color code the values:
- pink, if $\leq$ median of medians
- blue, if $\geq$ median of medians

The median of each column is shown in red.

Imagine that this array, with median 61:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11| 12| 13| 14| 15| 16| 17| 18| 19| 20|
| 51| 60| 73| 92| 57| 54| 75| 59| 91| 58| 71| 62| 67| 66| 59| 52| 61| 72| 55| 60| 79|

were laid out in a 3-high 2-D array in row major order:

<table>
<thead>
<tr>
<th>55</th>
<th>51</th>
<th>52</th>
<th>54</th>
<th>58</th>
<th>66</th>
<th>71</th>
</tr>
</thead>
<tbody>
<tr>
<td>57</td>
<td>59</td>
<td>60</td>
<td>60</td>
<td>61</td>
<td>75</td>
<td>72</td>
</tr>
<tr>
<td>62</td>
<td>59</td>
<td>91</td>
<td>67</td>
<td>73</td>
<td>79</td>
<td>92</td>
</tr>
</tbody>
</table>

Finally, color code the values:
- pink, if ≤ median of medians
- blue, if ≥ median of medians

Choose the median of medians (60) as the pivot $p$, and partition $A$. 

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10| 11| 12| 13| 14| 15| 16| 17| 18| 19| 20|
| 51| 55| 57| 54| 52| 59| 59| 58| 60| 60| 62| 67| 66| 71| 91| 61| 72| 75| 92| 79| 73|

Imagine that this array, with median 61:

$$
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\
A & 51 & 60 & 73 & 92 & 57 & 54 & 75 & 59 & 91 & 58 & 71 & 62 & 67 & 66 & 59 & 52 & 61 & 72 & 55 & 60 & 79 \\
\end{array}
$$

were laid out in a 3-high 2-D array in row major order:

$$
\begin{array}{cccccccc}
55 & 51 & 52 & 54 & 58 & 66 & 71 \\
57 & 59 & 60 & 60 & 61 & 75 & 72 \\
62 & 59 & 91 & 67 & 73 & 79 & 92 \\
\end{array}
$$

We seek the median, i.e., the $n/2^{th}$ smallest $(n/2 = 21/2 = 10)$, which falls into $>p$ region.

Choose the median of medians (60) as the pivot $p$, and partition $A$. 

$$
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\
A & 51 & 55 & 57 & 54 & 52 & 59 & 60 & 60 & 62 & 67 & 66 & 71 & 91 & 61 & 72 & 75 & 92 & 79 & 73 \\
\end{array}
$$

Imagine that this array, with median 61:

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\
51 & 60 & 73 & 92 & 57 & 54 & 75 & 59 & 91 & 58 & 71 & 62 & 67 & 66 & 59 & 52 & 61 & 72 & 55 & 60 & 79 \\
\end{array}
\]

were laid out in a 3-high 2-D array in row major order:

\[
\begin{array}{cccccccc}
55 & 51 & 52 & 54 & 58 & 66 & 71 \\
57 & 59 & 60 & 60 & 61 & 75 & 72 \\
62 & 59 & 91 & 67 & 73 & 79 & 92 \\
\end{array}
\]

We seek the median, i.e., the $n/2^{th}$ smallest $(n/2 = 21/2 = 10)$, which falls into $>p$ region, eliminating at least $2/3 \cdot \frac{1}{2} = 1/3$ the values.

Thus, the $>p$ region is no larger than $r = 1-1/3 = 2/3$ the size of the whole.

- Every case.
  1. On each iteration, region sizes reduced by constant ratio $r$.
     - Partitioning time is linear in region size.
     - Total effort for partitioning. $n + r\cdot n + r^2\cdot n + r^3\cdot n + ... = n/(1-r)$
     - I.e., linear in $n$, not counting time to compute the pivot.
  2. On each iteration, the cost to compute the pivot is also linear in region size.

Thus, total effort, would be linear in $n$. In particular, even in the worst-case.

- Every case.
  1. On each iteration, region sizes reduced by constant ratio $r$.
     Partitioning time is linear in region size.
     Total effort for partitioning. \( n + \frac{2}{3} \cdot n + \left( \frac{2}{3} \right)^2 \cdot n + \left( \frac{2}{3} \right)^3 \cdot n + \ldots = \frac{n}{1-\frac{2}{3}} = 3 \cdot n \)
     ✔️ i.e., linear in $n$, not counting time to compute the pivot.
  2. On each iteration, the cost to compute the pivot is also linear in region size.

Thus, total effort, would be linear in $n$. In particular, even in the worst-case.
**Performance Goal:** Pivots computed as median of medians of A[ L . . R-1 ].

- **Every case.**
  1. On each iteration, region sizes reduced by constant ratio $r$.
     Partitioning time is linear in region size.
     Total effort for partitioning. $n + (2/3)n + (2/3)^2 n + (2/3)^3 n + ... = n/(1-2/3) = 3\cdot n$
     I.e., linear in $n$, *not counting time to compute the pivot.*
  2. On each iteration, the cost to compute the pivot is also linear in region size.
     But how will we compute the median of medians of A[ L . . R-1 ]?

Thus, total effort, would be linear in $n$. In particular, even in the **worst-case**.
Performance Goal: Pivots computed as median of medians of $A[L\ldots R-1]$ using recursion, i.e., apply the worst-case median algorithm to the $n/3$ medians of groups of 3 elements.

☞ Consider recursion when designing an algorithm.
**Performance Goal:** Pivots computed as median of medians of \( A[L..R-1] \) using recursion, i.e., apply the worst-case median algorithm to the \( n/3 \) medians of groups of 3 elements.

This works, but alas, there are too many groups of 3, so the total cost is super-linear.

☞ Consider recursion when designing an algorithm.
Performance Goal: Pivots computed as median of medians of $A[L..R-1]$ using recursion, i.e., apply the worst-case median algorithm to the $n/3$ medians of groups of 3 elements.

This works, but alas, there are too many groups of 3, so the total cost is super-linear.

But don’t lose heart. All is not lost, because ...
Performance Goal: Pivots computed as median of medians of $A[L..R-1]$ using recursion, i.e., apply the worst-case median algorithm to the $n/5$ medians of groups of 5 elements.

This works, and is linear.

Selection of a partition region eliminates at least $\frac{3}{5} \cdot \frac{1}{2} = \frac{3}{10}$ the values.

Thus, the selected region is no larger than $r = 1 - \frac{3}{10} = \frac{7}{10}$ the size of the whole.

Total effort for partitioning. $n + (\frac{7}{10}) \cdot n + (\frac{7}{10})^2 \cdot n + (\frac{7}{10})^3 \cdot n + ... = n/(1-\frac{7}{10}) = 3.33 \cdot n$

In effect, the reduction ratio $r$ shrinks slightly (from $2/3$ to $3/10$), but the number of groups shrinks more than enough (from $n/3$ to $n/5$) to render the total linear.