Selection is similar to sorting, except instead of an ordered list we are more interested in finding a k^{th} order statistic in a group of elements.

k^{th} ORDER STATISTIC: Given a group of numbers the k^{th} order statistic is the k^{th} smallest element in the group.

EXAMPLE: Given an arbitrary array find the 4^{th} order statistic.

ARRAY $= [7, 10, 2, 4, \emptyset] \xrightarrow{\text{SORTED}} [\emptyset, 2, 4, 7, 10]$

4^{th} SMALLEST ELEMENT (7)

4^{th} ORDER STATISTIC IN THE ARRAY IS 7.

$O(n \lg n)$ RUNTIME

Given previous work with mergesort and quicksort it is trivial to solve a selection problem in $O(n \lg n)$ time

1) Apply desired sorting algorithm
2) Return the k^{th} order statistic with the element at index position k ($k-1$ for zero-based arrays).

We can do better! Selection is easier than sorting.

$O(n)$ RUNTIME WITH RANDOMIZATION

We can accomplish $O(n)$ time on average utilizing the principles of quicksort.

Recall with quicksort you want to partition the array around a pivot element.

1) Pick an element of the array to be the pivot

3	8	2	5	1

← PIVOT

2) Rearrange the array so:
   a) elements to the left of the pivot are less than the pivot.
   b) elements to the right of the pivot are greater than the pivot.

1	2	3	5	8

$\underbrace{1\ 2}_{<\text{PIVOT}}\ 3\ \underbrace{5\ 8}_{>\text{PIVOT}}$

(con't on next sheet)

O(n) RUNTIME WITH RANDOMIZATION (cont)

The key to getting O(n) time on average with the selection algorithm lies in eliminating the section of the array the k^{th} order statistic cannot be in, reducing the problem size.

EXAMPLE: Suppose we are looking for the 5^{th} order statistic in an input array of length 10. We partition the array, and the pivot ends up in the third position of the partitioned array.

9	Ø	2	4	6	3	7	8	5	1

↑ PIVOT

APPLY QUICKSORT →

1	Ø	2	4	6	3	7	8	5	9

$\underbrace{1\ Ø}_{< \text{PIVOT}}$ 2 $\underbrace{4\ 6\ 3\ 7\ 8\ 5\ 9}_{> \text{PIVOT}}$

The 5^{th} order statistic ends up in the section of the partitioned array where elements are greater than the pivot. Therefore, we only need be concerned with that section of the array.

The above works for all input arrays and partitions; Compare the pivot's final index position with the k value and search the appropriate partition.

Now since we are no longer concerned with the pivot and elements less than the pivot when we perform a recursion on the elements greater than the pivot the search for the k^{th} order statistic must be adjusted.

new k^{th} order statistic = old k - pivot index

This holds for elements greater than the pivot. If the k^{th} order statistic is less than the pivot it is still the k^{th} order statistic. So since the 5^{th} order statistic was in the partition greater than the pivot the new value for the k^{th} order statistic is

$5 - 3 = 2, \quad k = 2$

Repeat until the selection target is found.

PSEUDOCODE:

```
Select( array A, length n, order statistic k)
  if n=1, return A
  choose pivot p from A uniformly at random
  partition A around p, let j = new index of p
  if j = k, return p
  if j > k, return Select( <p, j-1, k)
  if j < k, return Select( >p, n-k, k-j)
```

$O(n)$ RUNTIME WITH RANDOMIZATION (cont)

ANALYSIS

The runtime analysis runs very similarly to that of quicksort. Much like quicksort if you have an array sorted in descending order and you want the 1st order statistic with the pivot selection always being the first element in each successive search you will have a runtime of $O(n^2)$. This is why we use randomization to ensure an extremely low probability of having a quadratic runtime.

THEORETICAL BEST TIME

Forcing the algorithm to choose the median every time will evenly split the array, but with a uniformly random selection of the pivot index this scenario is about as likely as your computer getting blasted by a meteor. However, analyzing the scenario is helpful to let us know what our best case running time is.

Use the Master Method, dude! Median = evenly split partitions

RECURRENCE: $T(n) \le aT(n/b) + O(n^d)$

$a = 1$ since we throw away half of the array once we compare the pivot's resting index with the k^{th} order statistic.

$b = 2$ we're dividing the array equally in half every time by selecting the median as the pivot value.

$d = 1$ we run through all entries of the array or section of the array to form the partitions.

$b^d = 2^1 \rightarrow a < b^d$, CASE 2 $O(n^d) \rightarrow O(n)$

USING RANDOMIZED SELECTION

A 25% - 75% split in the input data gets us close enough to $O(n)$ runtime.

Since an average is about 50% of the data, a uniform distribution used to select a pivot element 50% of the input array elements are likely to be chosen as the potential pivot elements reside within $0.25n$ and $0.75n$.

Low-level proof not necessary, practical use is $O(n)$

PYTHON IMPLEMENTATION OF SELECTION ALGORITHM

```
# selection_algorithm_example.py
"""
An example of a linear runtime selection algorithm leveraging
quicksort principles.
"""

from random import randint

def main():
  target_list = [9, 8, 2, 1, 5, 7, 0, 4, 6, 3]

  k_order_statistic = randint(1, len(target_list))

  _, target_value = quick_selection(target_list, k_order_statistic)

  last_digit = k_order_statistic % 10

  if last_digit == 1:
    print("The {}st order statistic is {}.".format(k_order_statistic,
    target_value))
  elif last_digit == 2:
    print("The {}nd order statistic is {}.".format(k_order_statistic,
    target_value))
  elif last_digit == 3:
    print("The {}rd order statistic is {}.".format(k_order_statistic,
    target_value))
  else:
    print("The {}th order statistic is {}.".format(k_order_statistic,
    target_value))

def quick_selection(target_list, k):
  # Base case: if the list length is 1 or less the list is sorted.
  if len(target_list) < 2:
    return target_list, target_list[0]

  pivot_index = randint(0, len(target_list) - 1)
  pivot_value = target_list[pivot_index]

  # Swap the pivot index with the leftmost element.
  target_list = swap(target_list, pivot_index, 0)

```

(con't on next sheet)

PYTHON IMPLEMENTATION OF SELECTION ALGORITHM (con't)

```
    # Set pointers for partitions
    # i is the pointer for the index where all elements in
    # positions less than index i are less than or equal to
    # the pivot.
    # j is the pointer to the index where all elements in
    # positions greater than the index j have not yet been
    # compared to the pivot.
    i = j = 1

    while j < len(target_list):
        if target_list[j] < pivot_value:
            target_list = swap(target_list, i, j)
            i += 1
        j += 1

    # Set the index of where the pivot should reside.
    pivot_index = i-1

    # Place the pivot in its rightful place.
    target_list = swap(target_list, 0, pivot_index)

    # Determine how to continue solving the problem.
    # Remember, k is on a 1-based index and pivot_index is 0-based.
    if k-1 == pivot_index:
        return target_list, target_list[pivot_index]
    elif k-1 < pivot_index:
        return quick_selection(target_list[:pivot_index], k)
    else:
        return quick_selection(target_list[i:], k-i)

def swap(L, i, j):
    L[i], L[j] = L[j], L[i]
    return L

if __name__ == "__main__":
    main()
```

```go
package main

// An example of a linear runtime selection algorithm leveraging quicksort
// principles.

import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    targetSlice := []int{7, 0, 1, 2, 5, 8, 6, 3, 9, 4}
    r := rand.New(rand.NewSource(time.Now().UnixNano()))
    kOrderStat := 1 + r.Intn(len(targetSlice))

    fmt.Println("The list in question is:\t", targetSlice)

    _, targetValue := quickSelect(targetSlice, kOrderStat)

    lastDigit := kOrderStat % 10

    if lastDigit == 1 {
        fmt.Printf("The %dst order statistic is %d.\n", kOrderStat, targetValue)
    } else if lastDigit == 2 {
        fmt.Printf("The %dnd order statistic is %d.\n", kOrderStat, targetValue)
    } else if lastDigit == 3 {
        fmt.Printf("The %drd order statistic is %d.\n", kOrderStat, targetValue)
    } else {
        fmt.Printf("The %dth order statistic is %d.\n", kOrderStat, targetValue)
    }
}

func quickSelect(targetSlice []int, k int) ([]int, int) {
    // Base case: A slice of length 1 or 0 is sorted by default.
    if len(targetSlice) < 2 {
        return targetSlice, targetSlice[0]
    }

    r := rand.New(rand.NewSource(time.Now().UnixNano()))
    pivotIndex := r.Intn(len(targetSlice) - 1)
    pivotValue := targetSlice[pivotIndex]

    // Swap the pivot index with the leftmost element.
    targetSlice = swap(targetSlice, pivotIndex, 0)
```

```go

    // Set pointers for partitions.
    // i is the pointer for the index where all elements in positions
    // less than index i are less than or equal to the pivot.
    // j is the pointer to the index where all elements in positions
    // greater than the index j have not yet been compared to the pivot.
    i := 1
    j := 1

    for j < len(targetSlice) {
        if targetSlice[j] < pivotValue {
            targetSlice = swap(targetSlice, i, j)
            i++
        }
        j++
    }

    // Set the index of where the pivot should reside.
    pivotIndex = i - 1

    // Place the pivot in its rightful place.
    targetSlice = swap(targetSlice, 0, pivotIndex)

    // Determine how to continue solving the problem.
    // Remember, k is on a 1-based index and pivotIndex is 0-based.
    if k-1 == pivotIndex {
        return targetSlice, targetSlice[pivotIndex]
    } else if k-1 < pivotIndex {
        return quickSelect(targetSlice[:pivotIndex], k)
    } else {
        return quickSelect(targetSlice[i:], k-i)
    }
}

func swap(xint []int, i, j int) []int {
    xint[i], xint[j] = xint[j], xint[i]

    return xint
}
```

O(n) RUNTIME DETERMINISTICALLY

Gives guaranteed O(n) runtime in the worst case as opposed to randomized selections quadratic worst case performance.

* However, in practice the runtime of deterministic selection is slower than random selection as deterministic selection:

1) has larger constant factors than random selection in its non-bounded form (not using big-O notation as an estimated bound on performance)
2) does not operate in place as it requires an additional array to hold median values.

Deterministic selection runs off "median of medians" method.

MEDIAN OF MEDIANS

1) Divide the input array into as even chunks as possible
2) Sort each group (using merge sort, selection sort, etc.)
3) Extract the median of each group into a new array
4) Recursively compute median of the "median" array created in step 3.

DETERMINISTIC SELECTION PSEUDOCODE

SELECT MEDIAN OF MEDIANS (lines 3–6); SELECT (lines 7–10)

```
Select(array A, length n, order statistic k)
  if n = 1, return A
  break A into groups of 5  # (n/5 groups)
  sort each group
  copy the middle elements ("medians") of each group into array C
  pivot = Select(C, n/5, n/10)  # n/10 = median of n/5 elements
  partition A around pivot, let j = new index of pivot
  if j = k, return p
  if j < k, return Select(<pivot, j-1, k)
  else return Select(>pivot, n-k, k-j)
```

(cont on next sheet)

O(n) RUNTIME DETERMINISTICALLY (con't)

ANALYSIS

From the pseudocode on the previous page, runtime analysis by line:

```
O(1)   break A into groups of 5   # n/5 groups
O(n)   sort each group
O(n)   copy the group medians into a new array, C.
T(n/5) pivot = Select(C, n/5, n/10)   # n/10 = median of n/5 elements
O(n)   partition A around pivot, let j = new index of pivot
     { if j = k, return pivot
T(?) { if j < k, return Select(< pivot, j-1, k)
     { else return Select(> pivot, n-k, k-j)
```

sort each group O(n) proof

Recall from merge sort our tight analysis of the runtime was there are $6n(\lg n + 1)$ operations per call to merge sort.

Since we have selected to break the input array, A, of length n into even groups of 5 for n/5 (LET m = 5) groups we can determine how many operations will occur when calling merge sort per group since the divisor, 5, is constant and only n can vary.

SUB m, $6m(\lg m + 1) \rightarrow 6(5)(\lg 5 + 1) \approx \leq 120$ OPS ($\lg 5 \leq 3$)

$\therefore$ with $\frac{n}{5}$ GROUPS $\left(\leq 120 \frac{\text{OPS}}{\text{GROUP}}\right) \rightarrow \leq 24n$ OPS OR $\underline{\underline{O(n)}}$ for all groups

DETERMINING ? IN T(?)

Let $T(n)$ = maximum runtime of the deterministic selection algorithm on an input array of length n.

There is a constant, c, such that

1) $T(1) = 1$ (Base case return, if n = 1, return A)

2) $T(n) \leq \underbrace{cn}_{\substack{O(n)\\ \text{lines}}} + \underbrace{T(n/5)}_{\substack{\text{Determine}\\ \text{pivot}\\ \text{recursion}}} + \underbrace{T(?)}_{\substack{\text{Select}\\ k\\ \text{recursions}}}$

(con't on next sheet)

O(n) RUNTIME DETERMINISTICALLY (con't)

ANALYSIS (con't)

DETERMINING ? IN T(?) (con't)

SUPPORTING PROOF TO DETERMINE ?

The 2nd recursive call on Select is guaranteed to be on an array of roughly size $\leq \frac{7}{10} n$

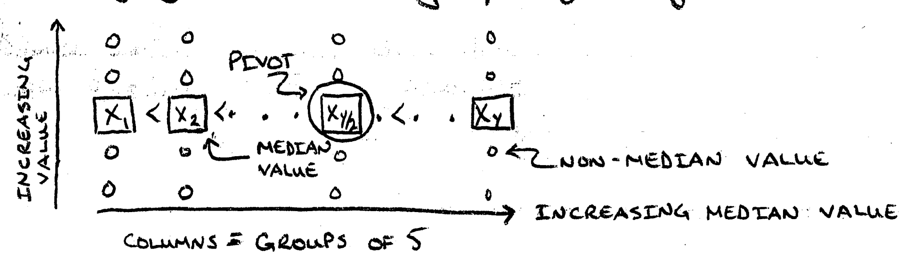
This will allow us to replace ? with $(7/10)n$

Let $y = n/5$, the number of groups

Let $x_i = i^{th}$ smallest statistic of the median elements drawn from each group in y.

$\therefore$ pivot $= x_{y/2}$

Laying out the y groups of array A in a 2-D grid:



EXAMPLE

A, n=20

7	2	17	12	13	8	20	4	6	3	19	1	9	5	16	10	15	18	14	11

SORT THE GROUPS

2	7	12	13	17	3	4	6	8	20	1	5	9	16	19	10	11	14	15	18

GRID REPRESENTATION

20	19	17	18
8	16	13	15
6	9	12	14
4	5	7	11
3	1	2	10

Likewise $x_{y/2}$ is guaranteed to be less than 3 out of 5 elements in about 50% of the groups.

$\therefore x_{y/2} <$ 30% of elements in A

when arranged on the grid it is easy to see $x_{y/2}$ is bigger than 3 out of 5 (60%) of the elements in about 50% of the groups

$\therefore x_{y/2} >$ 30% of the elements in A guaranteed

* 30% of input array eliminated leaving $\leq \frac{7}{10} n$

(con't on next sheet)

O(n) Runtime Deterministically (con't)

Analysis (con't)

Substituting ? into $T(n) \rightarrow T(n) \le cn + T(n/5) + T(7n/10)$

Since the recurrences do not break into evenly divided subproblems we cannot use the Master Method of analysis. That leaves... guess and check!

Guess: There is some constant, a (independant of n), such that $T(n) \le an$ whenever $n \ge 1$. (Proves linear runtime if true)

Let $a = 10c$ (constant multiple of constant work done)
← ARBITRARY, BUT CONVENIENT GUESS TO ADJUST IF WE ARE WRONG

Then $T(n) \le an$ for all $n \ge 1$

By induction:

Base case, $n = 1$: $T(1) = 1 \le a(1)$ since $c \ge 1$, $a = 10(1)$

Inductive step: $n > 1$
Hypothesis: $T(z) \le az$ for all $z < n$

Given: $T(n) \le cn + T(n/5) + T(7n/10)$
← less than n ← less than n

Since $T(n) \le an$ for all $n \ge 1$ we can substitute a in place of $T(\ldots)$ with the hypothesis

$\rightarrow T(n) \le cn + a(n/5) + a(7n/10)$

combining terms $T(n) \le n(c + 9a/10)$, $a = 10c \therefore c = a/10$

substitute c $T(n) \le n(a/10 + 9a/10) \rightarrow T(n) \le an$

$\therefore T(n)$ runs in $O(n)$ time!

```python
# deterministic_selection_algorithm_example.py
"""
An example of using the deterministic "median of medians" method to
conduct a selection in linear, O(n), runtime.
"""

from random import randint

def main():
    target_list = [7, 2, 17, 12, 13, 8, 20, 4, 6, 3, 19, 1, 9, 5, 16, 10, 15,
    18, 14, 11]

    k_order_statistic = randint(1, len(target_list))

    _, target_value = quick_select(target_list, k_order_statistic)

    last_digit = find_last_digit(k_order_statistic)

    print("From the list")
    print(sorted(target_list))
    result = output(k_order_statistic, target_value, last_digit)
    print(result)

def quick_select(target_list, k):
    # Base case: A list of length 1 or 0 is by default sorted.
    if len(target_list) < 2:
        return target_list, target_list[0]

    pivot_index = find_median_of_medians(target_list)
    pivot_value = target_list[pivot_index]

    # Swap the pivot value to the leftmost index position
    target_list = swap(target_list, 0, pivot_index)

    # Set up the pointers
    # i is the index delineating the partition of all values less
    # than or equal to the pivot.
    # j is the index value of which all indices greater than j have not
    # yet been compared against the pivot value.
    i = j = 1

    # Perform the sort
    while j < len(target_list):
        if target_list[j] <= pivot_value:
            target_list = swap(target_list, i, j)
```

```python
            i += 1
        j += 1

    # Swap the pivot value into its rightful position
    pivot_index = i - 1
    target_list = swap(target_list, 0, pivot_index)

    # Determine how to continue solving the problem.
    # Remember, k is on a 1-based index and pivot_index is 0-based.
    if k-1 == pivot_index:
        return target_list, target_list[pivot_index]
    elif k-1 < pivot_index:
        return quick_select(target_list[:pivot_index], k)
    else:
        return quick_select(target_list[i:], k-i)


def find_median_of_medians(target_list):
    """Method to select the median of medians from a list."""

    group_size = 5

    # Base case: A list less than the group size is close enough.
    if len(target_list) < group_size:
        return len(target_list)//2

    num_full_groups = len(target_list)//group_size
    medians = []
    median_indices = []

    for i in range(0, num_full_groups*group_size, group_size):
        target_list = selection_sort(target_list, i, i+5)
        medians.append(target_list[i+2])
        median_indices.append(i+2)

    _, median_of_medians = quick_select(medians,
    len(target_list)//(group_size*2))

    for idx, potential_median in enumerate(medians):
        if potential_median == median_of_medians:
            median_of_medians_index = median_indices[idx]

    return median_of_medians_index

def selection_sort(given_list, left_index, right_index):
```

```
    """Will always sort 5 elements. Used to determine median values."""

    for idx in range(left_index, right_index):
        min_value = given_list[idx]
        min_index = idx
        j = idx + 1
        while j < right_index:
            if given_list[j] < min_value:
                min_value = given_list[j]
                min_index = j
            j += 1
        given_list = swap(given_list, idx, min_index)

    return given_list

def swap(L, i, j):
    """Swaps values at indices i and j in list L."""

    L[i], L[j] = L[j], L[i]
    return L

def find_last_digit(k):
    """Determines the last digit in a base 10 integer."""

    return k%10

def output(k_order_statistic, target_value, last_digit):
    if k_order_statistic != 11 and last_digit == 1:
        result = "The {}st order statistic is {}.".format(k_order_statistic,
        target_value)
    elif k_order_statistic != 12 and last_digit == 2:
        result = "The {}nd order statistic is {}.".format(k_order_statistic,
        target_value)
    elif k_order_statistic != 13 and last_digit == 3:
        result = "The {}rd order statistic is {}.".format(k_order_statistic,
        target_value)
    else:
        result = "The {}th order statistic is {}.".format(k_order_statistic,
        target_value)
    return result

if __name__ == "__main__":
    main()
```

```go
package main

// An example of using the deterministic "median of medians" method to
// conduct a selection in linear, O(n), runtime.

import (
    "fmt"
    "math/rand"
    "sort"
    "time"
)

func main() {
    targetSlice := []int{7, 2, 17, 12, 13, 8, 20, 4, 6, 3, 19, 1, 9, 5, 16,
        10, 15, 18, 14, 11}

    kOrderStat := genKOrderStat(len(targetSlice))

    _, targetValue := quickSelect(targetSlice, kOrderStat)

    lastDigit := findLastDigit(kOrderStat)

    fmt.Println("From the list")
    sort.Ints(targetSlice)
    fmt.Println(targetSlice)
    result := genOutput(kOrderStat, targetValue, lastDigit)
    fmt.Println(result)
}

func genKOrderStat(sliceLength int) int {

    // Possible strategy to implement a uniform distribution of integers
    // Use Source (source of uniformly-distributed pseudo-random int64 values
    // in the range [0, 1<<63))
    // Use a modulo operator to shoehorn the values into the range I want.
    // For example, 1<<63 = 9223372036854775807
    // If I want values distributed between 1 and 20 inclusive
    // kOrderStat := 1 + uniformRandomValue%(upperLimit+1)

    r := rand.New(rand.NewSource(time.Now().UnixNano()))
    kOrderStat := r.Intn(sliceLength) + 1

    return kOrderStat
}

```

```
func quickSelect(xint []int, k int) ([]int, int) {
    // Base case: A list of length 1 or 0 is sorted by default.
    if len(xint) < 2 {
        return xint, xint[0]
    }

    pivotIndex := findMedianOfMedians(xint)
    pivotValue := xint[pivotIndex]

    // Swap the pivot value to the leftmost index position
    xint = swap(xint, 0, pivotIndex)

    // Set up the pointers
    // i is the index delineating the partition between all values less
    // than or equal to the pivot.
    // j is the index value of which all indices greater than j have
    // not yet been compared against the pivot value.
    i := 1
    j := 1

    // Perform the sort
    for j < len(xint) {
        if xint[j] <= pivotValue {
            xint = swap(xint, i, j)
            i++
        }
        j++
    }

    // Swap the pivot value into its rightful position.
    pivotIndex = i - 1
    xint = swap(xint, 0, pivotIndex)

    // Determine how to continue solving the problem.
    // Remember, k is on a 1-based index and pivotIndex is 0-based
    if k-1 == pivotIndex {
        return xint, xint[pivotIndex]
    } else if k-1 < pivotIndex {
        return quickSelect(xint[:pivotIndex], k)
    } else {
        return quickSelect(xint[i:], k-i)
    }
}

func findMedianOfMedians(xint []int) int {
```

```go
    // Function to select the median of medians from the given slice.

    groupSize := 5
    xintLen := len(xint)

    // Base case: A list less than the group size is close enough.
    if xintLen < groupSize {
        return xintLen / 2
    }

    numFullGroups := xintLen / groupSize
    medians := make([]int, numFullGroups)
    medianIndices := make([]int, numFullGroups)
    var medianOfMediansIndex int

    for i := 0; i < numFullGroups; i++ {
        startIndex := i * 5
        xint = selectionSort(xint, startIndex, startIndex+5)
        medians[i] = xint[startIndex+2]
        medianIndices[i] = startIndex + 2
    }

    _, medianOfMedians := quickSelect(medians, xintLen/(groupSize*2))

    for idx, potentialMedian := range medians {
        if potentialMedian == medianOfMedians {
            medianOfMediansIndex = medianIndices[idx]
        }
    }

    return medianOfMediansIndex
}

func selectionSort(xint []int, leftIndex, rightIndex int) []int {
    // Will always sort 5 elements. Used to determine median values.

    for i := leftIndex; i < rightIndex; i++ {
        minValue := xint[i]
        minIndex := i
        j := i + 1
        for j < rightIndex {
            if xint[j] < minValue {
                minValue = xint[j]
                minIndex = j
            }
```

```go
            j++
        }
        xint = swap(xint, i, minIndex)
    }

    return xint
}

func swap(xint []int, i, j int) []int {
    // Swaps values at indices i and j in slice xint.

    xint[i], xint[j] = xint[j], xint[i]
    return xint
}

func findLastDigit(n int) int {
    // Determines the last digit in a base 10 integer.

    return n % 10
}

func genOutput(kOrderStat, targetValue, lastDigit int) string {
    var result string

    if kOrderStat != 11 && lastDigit == 1 {
        result = fmt.Sprintf("The %dst order statistic is %d.", kOrderStat,
            targetValue)
    } else if kOrderStat != 12 && lastDigit == 2 {
        result = fmt.Sprintf("The %dnd order statistic is %d.", kOrderStat,
            targetValue)
    } else if kOrderStat != 13 && lastDigit == 3 {
        result = fmt.Sprintf("The %drd order statistic is %d.", kOrderStat,
            targetValue)
    } else {
        result = fmt.Sprintf("The %dth order statistic is %d.", kOrderStat,
            targetValue)
    }
    return result
}
```