Sequential search

- **sequential search**: Locates a target value in an array/list by examining each element from start to finish.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value 42:

<table>
<thead>
<tr>
<th>index</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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

- Notice that the array is sorted. Could we take advantage of this?
### Sequential search

**Function:** `search(A[], thingToFind)`

1. Start at the beginning.
2. For each item in `A`:
   - If the item is what we're looking for:
     - Return its location.
3. If we got this far without returning already, what we're looking for isn't found.
   - Return failure.

<table>
<thead>
<tr>
<th>index</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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

```
  i
```
public static int search(int A[], int thingToFind) {
    for (int i=0; i<A.length; i++) {
        if (A[i]==thingToFind) {
            return i;
        }
    }
    /* we didn’t find it. return failure */
    return -1;
}
When Array Is Sorted

- This array is already sorted
- Do we really need to go through entire thing before quitting?
  - Suppose we’re searching for 38:
    - we know once we’ve reached $A[10]$ that we didn’t find it
When Array Is Sorted

```plaintext
/* version 2. ONLY WORKS if A[] IS SORTED */

search(A[], thingToFind)
    start at the beginning
    for each item in A <= thingToFind:
        if the item is what we're looking for:
            return its location
    if we got this far without returning already,
    what we're looking for isn't found.
    return failure
```

<table>
<thead>
<tr>
<th>index</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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>
```
/* version 2. ONLY WORKS if A[] IS SORTED */

public static int search(int A[], int thingToFind) {
    for (int i=0; i<A.length && A[i] <= thingToFind; i++) {
        if (A[i] == thingToFind) {
            return i;
        }
    }
    return -1;
}
Searching a Sorted List

- What if we’re ordering a pizza (and it’s 1998)?
- Looking for phone number of Sammy’s Pizza
- Start with the A’s, then the B’s, *etc.*?
**Binary search (13.1)**

- **binary search**: Locates a target value in a *sorted* array/list by successively eliminating half of the array from consideration.
  
  - How many elements will it need to examine?
  
  - Example: Searching the array below for the value 42:

<table>
<thead>
<tr>
<th>index</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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

**min**

**mid**

**max**
• Class Arrays in java.util has many useful array methods:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarySearch(array, value)</td>
<td>returns the index of the given value in a sorted array (or &lt; 0 if not found)</td>
</tr>
<tr>
<td>binarySearch(array, minIndex, maxIndex, value)</td>
<td>returns index of given value in a sorted array between indexes min / max - 1 (&lt; 0 if not found)</td>
</tr>
<tr>
<td>copyOf(array, length)</td>
<td>returns a new resized copy of an array</td>
</tr>
<tr>
<td>equals(array1, array2)</td>
<td>returns true if the two arrays contain same elements in the same order</td>
</tr>
<tr>
<td>fill(array, value)</td>
<td>sets every element to the given value</td>
</tr>
<tr>
<td>sort(array)</td>
<td>arranges the elements into sorted order</td>
</tr>
<tr>
<td>toString(array)</td>
<td>returns a string representing the array, such as &quot;[10, 30, -25, 17]&quot;</td>
</tr>
</tbody>
</table>

• Syntax: Arrays.methodName(parameters)
Arrays.binarySearch

// searches an entire sorted array for a given value
// returns its index if found; a negative number if not found
// Precondition: array is sorted
Arrays.binarySearch(array, value)

// searches given portion of a sorted array for a given value
// examines minIndex (inclusive) through maxIndex (exclusive)
// returns its index if found; a negative number if not found
// Precondition: array is sorted
Arrays.binarySearch(array, minIndex, maxIndex, value)

• The binarySearch method in the Arrays class searches an array very efficiently if the array is sorted.
  – You can search the entire array, or just a range of indexes (useful for "unfilled" arrays such as the one in ArrayIntList)
  – If the array is not sorted, you may need to sort it first
// index    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
int[] a = {-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92};

int index  = Arrays.binarySearch(a, 0, 16, 42);   // index1 is 10
int index2 = Arrays.binarySearch(a, 0, 16, 21);   // index2 is -7

• binarySearch returns the index where the value is found

• if the value is *not* found, binarySearch returns:
  
  -(insertionPoint + 1)

• where insertionPoint is the index where the element *would* have been, if it had been in the array in sorted order.
• To insert the value into the array, negate insertionPoint + 1

  int indexToInsert21 = -(index2 + 1);    // 6
// Returns the index of an occurrence of target in a, // or a negative number if the target is not found. // Precondition: elements of a are in sorted order
public static int binarySearch(int[] a, int target) {
    int min = 0;
    int max = a.length - 1;

    while (min <= max) {
        int mid = (min + max) / 2;
        if (a[mid] < target) {
            min = mid + 1;
        } else if (a[mid] > target) {
            max = mid - 1;
        } else {
            return mid; // target found
        }
    }

    return -(min + 1); // target not found
}
Recursive binary search (13.3)

- Write a recursive `binarySearch` method.
  - If the target value is not found, return its negative insertion point.

```
int index  = binarySearch(data, 42);  // 10
int index2 = binarySearch(data, 66);  // -14
```
// Returns the index of an occurrence of the given value in 
// the given array, or a negative number if not found. 
// Precondition: elements of a are in sorted order
public static int binarySearch(int[] a, int target) {
    return binarySearch(a, target, 0, a.length - 1);
}

// Recursive helper to implement search behavior.
private static int binarySearch(int[] a, int target,
                                int min, int max) {
    if (min > max) { 
        return -1;       // target not found
    } else {
        int mid = (min + max) / 2;
        if (a[mid] < target) {
            return binarySearch(a, target, mid + 1, max);
        } else if (a[mid] > target) {
            return binarySearch(a, target, min, mid - 1);
        } else {
            return mid;       // target found; a[mid] == target
        }
    }
}
• Can we `binarySearch` an array of Strings?
  – Operators like `<` and `>` do not work with `String` objects.
  – But we do think of strings as having an alphabetical ordering.

• **natural ordering**: Rules governing the relative placement of all values of a given type.

• **comparison function**: Code that, when given two values `A` and `B` of a given type, decides their relative ordering:
  – `A < B`, `A == B`, `A > B`
The `compareTo` method (10.2)

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
  - Example: in the `String` class, there is a method:
    ```java
    public int compareTo(String other)
    ```

- A call of `A.compareTo(B)` will return:
  - a value < 0 if `A` comes "before" `B` in the ordering,
  - a value > 0 if `A` comes "after" `B` in the ordering,
  - or 0 if `A` and `B` are considered "equal" in the ordering.
**Runtime Efficiency (13.2)**

- **efficiency**: A measure of the use of computing resources by code.
  - can be relative to speed (time), memory (space), etc.
  - most commonly refers to run time

- Assume the following:
  - Any single Java statement takes the same amount of time to run.
  - A method call's runtime is measured by the total of the statements inside the method's body.
  - A loop's runtime, if the loop repeats N times, is N times the runtime of the statements in its body.
Efficiency examples

\[
\begin{aligned}
\text{statement1; } & \quad 3 \\
\text{statement2; } & \quad \text{N} \\
\text{statement3; } & \quad 4N + 3 \\
\end{aligned}
\]

\[
\begin{aligned}
& \text{for (int } i = 1; \ i \leq N; \ i++ \text{) } \{ \\
& \quad \text{statement4; } \\
& \}\ \\
& \text{for (int } i = 1; \ i \leq N; \ i++ \text{) } \{ \\
& \quad \text{statement5; } \\
& \quad \text{statement6; } \\
& \quad \text{statement7; } \\
& \}\end{aligned}
\]
for (int i = 1; i <= N; i++) {
  for (int j = 1; j <= N; j++) {
    statement1;
  }
}

for (int i = 1; i <= N; i++) {
  statement2;
  statement3;
  statement4;
  statement5;
}

• How many statements will execute if N = 10? If N = 1000?
Algorithm growth rates (13.2)

• We measure runtime in proportion to the input data size, N.
  – growth rate: Change in runtime as N changes.

• Say an algorithm runs \(0.4N^3 + 25N^2 + 8N + 17\) statements.
  – Consider the runtime when N is extremely large.
  – We ignore constants like 25 because they are tiny next to N.
  – The highest-order term \((N^3)\) dominates the overall runtime.

  – We say that this algorithm runs "on the order of" \(N^3\).
  – or \(O(N^3)\) for short ("Big-Oh of N cubed")
Complexity classes

- **complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size $N$.

<table>
<thead>
<tr>
<th>Class</th>
<th>Big-Oh</th>
<th>If you double $N$, ...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$O(1)$</td>
<td>unchanged</td>
<td>10ms</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log_2 N)$</td>
<td>increases slightly</td>
<td>175ms</td>
</tr>
<tr>
<td>linear</td>
<td>$O(N)$</td>
<td>doubles</td>
<td>3.2 sec</td>
</tr>
<tr>
<td>log-linear</td>
<td>$O(N \log_2 N)$</td>
<td>slightly more than doubles</td>
<td>6 sec</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(N^2)$</td>
<td>quadruples</td>
<td>1 min 42 sec</td>
</tr>
<tr>
<td>cubic</td>
<td>$O(N^3)$</td>
<td>multiplies by 8</td>
<td>55 min</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^N)$</td>
<td>multiplies drastically</td>
<td>$5 \times 10^{61}$ years</td>
</tr>
</tbody>
</table>
**Binary search (13.1, 13.3)**

- **Binary search** successively eliminates half of the elements.
  
  - *Algorithm:* Examine the middle element of the array.
    - If it is too big, eliminate the right half of the array and repeat.
    - If it is too small, eliminate the left half of the array and repeat.
    - Else it is the value we're searching for, so stop.
  
  - Which indexes does the algorithm examine to find value 22?
  - What is the runtime complexity class of binary search?

<table>
<thead>
<tr>
<th>index</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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>-1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>14</td>
<td>22</td>
<td>29</td>
<td>31</td>
<td>37</td>
<td>56</td>
</tr>
</tbody>
</table>
For an array of size \( N \), it eliminates \( \frac{1}{2} \) until 1 element remains.

\[ N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \ldots, 4, 2, 1 \]

- How many divisions does it take?

Think of it from the other direction:
- How many times do I have to multiply by 2 to reach \( N \)?

\[ 1, 2, 4, 8, \ldots, \frac{N}{4}, \frac{N}{2}, N \]
- Call this number of multiplications "\( x \)."

\[ 2^x = N \]

\[ x = \log_2 N \]

Binary search is in the **logarithmic** complexity class.
• **sorting**: Rearranging the values in an array or collection into a specific order (usually into their "natural ordering").

  – one of the fundamental problems in computer science
  – can be solved in many ways:
    • there are many sorting algorithms
    • some are faster/slower than others
    • some use more/less memory than others
    • some work better with specific kinds of data
    • some can utilize multiple computers / processors, ...

  – *comparison-based sorting*: determining order by comparing pairs of elements:
    • `<`, `>`, `compareTo`, ...
• The Arrays class in java.util has a static method sort that sorts the elements of an array

```java
String[] words = {"foo", "bar", "baz", "ball"};
Arrays.sort(words);
System.out.println(Arrays.toString(words));
// [ball, bar, baz, foo]
```
• **selection sort**: Orders a list of values by repeatedly putting the smallest or largest unplaced value into its final position.

The algorithm:
– Look through the list to find the smallest value.
– Swap it so that it is at index 0.
– Look through the list to find the second-smallest value.
– Swap it so that it is at index 1.
...
– Repeat until all values are in their proper places.
**Selection sort example**

- **Initial array:**

| index | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  | 11  | 12  | 13  | 14  | 15  | 16  |
|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| value | 22  | 18  | 12  | -4  | 27  | 30  | 36  | 50  | 7   | 68  | 91  | 56  | 2   | 85  | 42  | 98  | 25  |

- **After 1st, 2nd, and 3rd passes:**

<table>
<thead>
<tr>
<th>index</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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>18</td>
<td>12</td>
<td>22</td>
<td>27</td>
<td>30</td>
<td>36</td>
<td>50</td>
<td>7</td>
<td>68</td>
<td>91</td>
<td>56</td>
<td>2</td>
<td>85</td>
<td>42</td>
<td>98</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>12</td>
<td>22</td>
<td>27</td>
<td>30</td>
<td>36</td>
<td>50</td>
<td>7</td>
<td>68</td>
<td>91</td>
<td>56</td>
<td>18</td>
<td>85</td>
<td>42</td>
<td>98</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>22</td>
<td>27</td>
<td>30</td>
<td>36</td>
<td>50</td>
<td>12</td>
<td>68</td>
<td>91</td>
<td>56</td>
<td>18</td>
<td>85</td>
<td>42</td>
<td>98</td>
<td>25</td>
</tr>
</tbody>
</table>
Selection sort code

// Rearranges the elements of a into sorted order using
// the selection sort algorithm.
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        // find index of smallest remaining value
        int min = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }

        // swap smallest value its proper place, a[i]
        swap(a, i, min);
    }
}
What is the complexity class (Big-Oh) of selection sort?

<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>16</td>
</tr>
<tr>
<td>4000</td>
<td>47</td>
</tr>
<tr>
<td>8000</td>
<td>234</td>
</tr>
<tr>
<td>16000</td>
<td>657</td>
</tr>
<tr>
<td>32000</td>
<td>2562</td>
</tr>
<tr>
<td>64000</td>
<td>10265</td>
</tr>
<tr>
<td>128000</td>
<td>41141</td>
</tr>
<tr>
<td>256000</td>
<td>164985</td>
</tr>
</tbody>
</table>
• **merge sort**: Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

The algorithm:
- Divide the list into two roughly equal halves.
- Sort the left half.
- Sort the right half.
- Merge the two sorted halves into one sorted list.

- Often implemented recursively.
- An example of a "divide and conquer" algorithm.
  - Invented by John von Neumann in 1945
Merge sort example

<table>
<thead>
<tr>
<th>index</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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>22</td>
<td>18</td>
<td>12</td>
<td>-4</td>
<td>58</td>
<td>7</td>
<td>31</td>
<td>42</td>
</tr>
</tbody>
</table>
## Merging sorted halves

<table>
<thead>
<tr>
<th>Subarrays</th>
<th>Next include</th>
<th>Merged array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>i1 23 41 58 85</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>i2 23 41 58 85</td>
<td>14 23 32 41 58 67 76 85</td>
</tr>
<tr>
<td>1 2 3</td>
<td>i1 23 41 58 85</td>
<td>i</td>
</tr>
<tr>
<td>1 2 3</td>
<td>i2 23 41 58 85</td>
<td>i</td>
</tr>
<tr>
<td>1 2 3</td>
<td>i1 23 41 58 85</td>
<td>14 23 32 41 58 67 76 85</td>
</tr>
<tr>
<td>1 2 3</td>
<td>i2 23 41 58 85</td>
<td>i</td>
</tr>
<tr>
<td>1 2 3</td>
<td>i1 23 41 58 85</td>
<td>14 23 32 41 58 67 76 85</td>
</tr>
<tr>
<td>1 2 3</td>
<td>i2 23 41 58 85</td>
<td>i</td>
</tr>
<tr>
<td>1 2 3</td>
<td>i1 23 41 58 85</td>
<td>14 23 32 41 58 67 76 85</td>
</tr>
<tr>
<td>1 2 3</td>
<td>i2 23 41 58 85</td>
<td>i</td>
</tr>
</tbody>
</table>
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted
public static void merge(int[] result, int[] left, int[] right) {
    int i1 = 0; // index into left array
    int i2 = 0; // index into right array

    for (int i = 0; i < result.length; i++) {
        if (i2 >= right.length ||
            (i1 < left.length && left[i1] <= right[i2])) {
            result[i] = left[i1]; // take from left
            i1++;
        } else {
            result[i] = right[i2]; // take from right
            i2++;
        }
    }
}
// Rearranges the elements of a into sorted order using
// the merge sort algorithm.
public static void mergeSort(int[] a) {
    // split array into two halves
    int[] left  = Arrays.copyOfRange(a, 0, a.length/2);
    int[] right = Arrays.copyOfRange(a, a.length/2, a.length);

    // sort the two halves
    ...

    // merge the sorted halves into a sorted whole
    merge(a, left, right);
}
// Rearranges the elements of a into sorted order using
// the merge sort algorithm (recursive).
public static void mergeSort(int[] a) {
    if (a.length >= 2) {
        // split array into two halves
        int[] left = Arrays.copyOfRange(a, 0, a.length/2);
        int[] right = Arrays.copyOfRange(a, a.length/2, a.length);

        // sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        merge(a, left, right);
    }
}
• What is the complexity class (Big-Oh) of merge sort?

<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
</tr>
<tr>
<td>8000</td>
<td>0</td>
</tr>
<tr>
<td>16000</td>
<td>0</td>
</tr>
<tr>
<td>32000</td>
<td>15</td>
</tr>
<tr>
<td>64000</td>
<td>16</td>
</tr>
<tr>
<td>128000</td>
<td>47</td>
</tr>
<tr>
<td>256000</td>
<td>125</td>
</tr>
<tr>
<td>512000</td>
<td>250</td>
</tr>
<tr>
<td>1e6</td>
<td>532</td>
</tr>
<tr>
<td>2e6</td>
<td>1078</td>
</tr>
<tr>
<td>4e6</td>
<td>2265</td>
</tr>
<tr>
<td>8e6</td>
<td>4781</td>
</tr>
<tr>
<td>1.6e7</td>
<td>9828</td>
</tr>
<tr>
<td>3.3e7</td>
<td>20422</td>
</tr>
<tr>
<td>6.5e7</td>
<td>42406</td>
</tr>
<tr>
<td>1.3e8</td>
<td>88344</td>
</tr>
</tbody>
</table>

Input size (N)