Selection Problem

- In computer science, a selection algorithm is an algorithm for finding the kth smallest number in a list or array (such a number is called the kth order statistic).
- This includes the special cases for finding the minimum, maximum, and median elements.
- Selection algorithms with linear expected running time exist.
- Selection is a sub-problem of more complex problems like the nearest neighbor and shortest path problems.

Selection Problem

- We can formally specify the selection problem as follows:
  - Input: A[n] – a set of n (distinct) numbers, and an integer 1 ≤ i ≤ n.
  - Output: x ∈ A, where x is the kth largest number in A.
- We can solve the selection problem in O(n log n) time by sorting and returning the i'th element. But we can improve on this running time!

Selection Problem

(Minimum or Maximum)

- The Minimum (or Maximum) selection problem is a special case of the general selection problem
  - Input: A[n] – a set of n (distinct) numbers and an integer 1 (or n).
  - Output: The element x ∈ A that is the smallest (or largest) number in A.

Selection Problem

(Minimum or Maximum)

- In some applications, we may need to find both the minimum and the maximum of a set of n elements.
  - E.g., a graphics program may need to scale (x, y) coordinate data to fit onto a rectangular display screen of a particular size. The program must determine both the maximum and minimum value of each coordinate.
- One simple way to determine both the maximum and minimum is to sequentially call two separate functions that determine the minimum and the maximum.
Selection Problem
(Minimum or Maximum)

Both Minimum and Maximum

```
Min_Max(A)
{
    Min = Minimum(A);
    Max = Maximum(A);
    Return (Min, Max);
}
```

Running time: \((n - 1) + (n - 1) = 2n - 2\) comparisons

Is there a better algorithm?

Min_Max(A) {
        Min = A[1];
        Max = A[0];
    } else {
        Min = A[0];
        Max = A[1];
    }
    for j = 1 to \(|A|/2 - 1\) {
            Mn = A[2j + 1];
            Mx = A[2j];
        } else {
            Mn = A[2j];
            Mx = A[2j + 1];
        }
        if Min > Mn Min = Mn;
        if Max < Mx Max = Mx;
    }
    return (Max, Min);
}

This algorithm works well without any modification when the size of A is an even number.

If the size of A is an odd number, 2 more comparisons are needed to get the correct min and max.

The \(\#\) of comparisons for this algorithm is \(3 \times \frac{n}{2}\) when \(n\) is even or \(3 \times \frac{n}{2} + 1\) when \(n\) is odd to find out maximum and minimum.

Selection Problem
(Find \(k^{th}\) Element)

The general selection problem is to find the \(k^{th}\) element in a list.

In computer science, the general selection problem is for finding the \(k^{th}\) smallest number in a list (such a number is called the \(k^{th}\) order statistic).

Selection Problem
(Find \(k^{th}\) Element: Selection by sorting)

One simple way of finding the \(k^{th}\) element is to sort the list and return the \(k^{th}\) element. This can be done in \(O(n \log n)\) time, e.g...

```
Select_kth(A, k)
{
    MergeSort(A)
    Return (A[k - 1])
}
```

Is there better algorithm?

Selection Problem
(Find \(k^{th}\) Element: Nonlinear general selection algorithm)

Using the same ideas used in the selection sort algorithm, we can construct a simple, but inefficient general algorithm for finding the \(k^{th}\) smallest or \(k^{th}\) largest item in a list, requiring \(O(n)\) time, which is effective when \(k\) is small.
Selection Problem
(Find k\textsuperscript{th} Element: in expected linear time)

- The general selection problem appears more difficult than the simple problem of finding a minimum or maximum.
- There are asymptotically linear time algorithms for the general selection problem that use a divide and conquer approach.
- Randomized select uses the quicksort idea.
- In quicksort, the input is partitioned into two parts and quicksort is recursively called for both parts. In randomized select, however, we only need to process one of the partitions.

Randomized select uses the quicksort idea. In quicksort, the input is partitioned into two parts and quicksort is recursively called for both parts. In randomized select, however, we only need to process one of the partitions.

Selection Problem
(Find k\textsuperscript{th} Element: in “expected” linear time)

```java
Partition(A, p, r) {
    x = A[r]; // A[r] used as a pivot
    i = p - 1;
    for j = p to r - 1 {
        if A[j] ≤ x {
            i = i + 1;
            Swap(A[i], A[j]);
        }
    }
    swap(A[i+1], A[r]);
    return (i + 1);
}
```

Randomized select uses the quicksort idea. In quicksort, the input is partitioned into two parts and quicksort is recursively called for both parts. In randomized select, however, we only need to process one of the partitions.

Selection Problem
(Find k\textsuperscript{th} Element: in “expected” linear time)

```java
Randomized_Partition(A, p, r) {
    i = RANDOM(p, r);
    Swap(A[r], A[i]);
    return Partition(A, p, r);
}
```

Selection Problem
(Find k\textsuperscript{th} Element: in “expected” linear time)

Randomized select uses the quicksort idea. In quicksort, the input is partitioned into two parts and quicksort is recursively called for both parts. In randomized select, however, we only need to process one of the partitions.

Selection Problem
(Find k\textsuperscript{th} Element: in “expected” linear time)

Randomized select uses the quicksort idea. In quicksort, the input is partitioned into two parts and quicksort is recursively called for both parts. In randomized select, however, we only need to process one of the partitions.

Selection Problem
(Find k\textsuperscript{th} Element: in “expected” linear time)

```
Randomized_Select(A, p, r, i) {
    if (p == r) then // then only one element in sublist
        return A[p];
    q = Randomized_Partition(A, p, r)
    k = q - p + 1
    if (i == k) then // then the pivot value is the answer
        return A[q];
    else if (i < k) then
        return Randomized_Select(A, p, q - 1, i)
    else
        return Randomized_Select(A, q + 1, r, i - k);
}
```

Running time of Randomized\_Select

- Best case: Partition always divides the list evenly
  \[ T(n) = T\left(\frac{n}{2}\right) + n = O(n) \]
- Worst case: Partition always divides the list into a list with 1 element and a second with n-1 elements
  \[ T(n) = T(n - 1) + n = O(n^2) \]
- Average case: Partition divides the list into n-k and k-1 elements
  \[ T(n) = T(max(n - k, k - 1)) + n = O(n) \]