Linear-Time Selection

Order statistic

Consider a finite (totally-ordered) set S of n distinct elements and a number k , for $k, n \in \mathbb{N}$. An element $x \in S$ is the k-th smallest element of S, aka the k-th order statistic, if $|\{s \in S : s < x\}| = k - 1$. If $k = \lceil \frac{n}{2} \rceil$ then the k-th smallest element of S is also called the median of S.

Problem: Selection

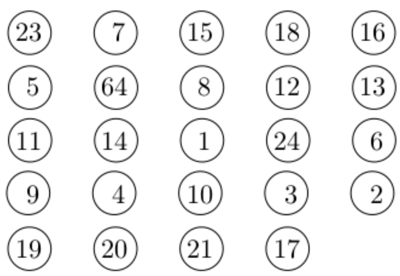
Given: A set S of n distinct (real) numbers and a number k, for $k, n \in \mathbb{N}$.

Compute: The k-th smallest element of S.

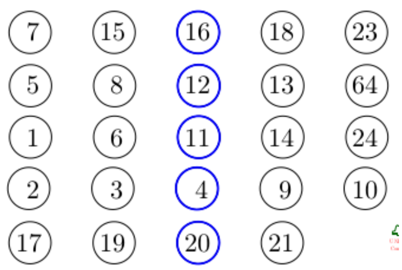
Not that, if $k = 1$ or $k = n$ then Selection can be solved easily using $n - 1$ comparisons. Furthermore, if the numbers of S are arranged in sorted order then the k-th smallest element can be found in $O(n)$ time (or even faster).

Deterministic Linear-Time Selection Algorithm

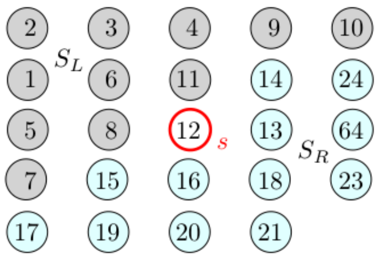
1. Divide the n elements of S into $\lfloor n/5 \rfloor$ groups of 5 elements each and (at most) one group containing the remaining n mod 5 elements.



2. Sort each group and compute its median.



3. Recursively find the median s of the $\lceil n/5 \rceil$ medians found in the previous step.
4. Partition S relative to the median-of-medians s into S_L and S_R (and $\{s\}$) such that all elements of S_L are smaller than s and all elements S_R are greater than s.



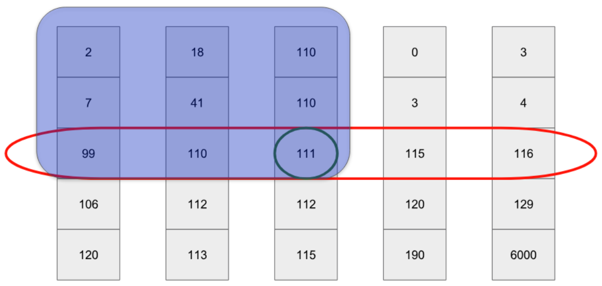
5. Let $m := |S_L \cup \{s\}|$. If $k = m$ then return s. Otherwise, if $k < m$ then recurse on S_L to find the k-th smallest element, else (if $k > m$) recurse on S_R to find the (k-m)-th smallest element.

Theorem (105)

Selection among n distinct numbers can be solved in $O(n)$ time, for any $n, k \in \mathbb{N}$.

** Proof:**

1. Grouping of the elements into sets of 5 can be in $O(n)$
2. Sorting and finding the median for each group can be done in $O(n)$
3. Finding the median-of-medians can be done in $O(\lceil n/5 \rceil)$
4. Partitioning of all elements can be done in $O(n)$
5. Recursion: Complexity is determined by the size of S_L and S_R. Therefore, we need to calculate how many elements can be in both sets.



As we can see the top-left rectangle will be less than or equal to s.

Hence, $m = |S_L| + 1 \geq \lceil \frac{1}{2} \lceil \frac{3}{5} \cdot n \rceil \rceil$ elements are smaller than s. Obviously, $m \geq \frac{3}{10}n$ and, thus, $|S_L| \geq \frac{3}{10}n - 1$.

Hence, $|S_R| \leq \frac{7}{10}n$. Similarly, $|S_R| \geq \frac{3}{10}n + O(1)$ and $S_L \leq \frac{7}{10}n + O(1)$, resulting in the

recurrence relation:

$$T(n) \leq T(\lceil \tfrac{n}{5} \rceil) + T(\tfrac{7n}{10}) + O(n)$$

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

$T(n) \leq c \cdot \lceil \frac{n}{5} \rceil + c \cdot \frac{7n}{10} + an$

$T(n) \leq cn/5 + c + c \cdot \frac{7n}{10} + an$

$T(n) \leq \frac{9cn}{10} + c + an \qquad\qquad T \in O(n)$

- Unfortunately, the constant hidden in the O-term is fairly large: Depending on details of the actual implementation, this algorithm requires about 50n comparisons! Hence, linear-time selection is too slow to be useful in practice.
- Worst-Case Complexity: $T \in O(n)$

Expected Linear-Time Selection

1. Pick an element s uniformly at random from S
2. Partition S relative to s into S_L and S_R such that all elements of S_L ares smaller than s and all elements of S_R are greater than s.
3. Let $m := |S_L \cup \{s\}$. If $k = m$ then return s. Otherwise, if $k < m$ then recurse on S_L to find the k-th smallest element, else (if $k > m$) recurse on S_R to find the (k-m)-th smallest element.

What is the complexity of an randomized algorithm?

Worst case: If s is the smallest or largest element of S then S shrinks by only one element, and we get $O(n^2)$ complexity. The probability of consistently picking an element of S which currently is the smallest or largest is

$$\frac{2}{n} \cdot \frac{2}{n-1} \cdot \frac{2}{n-2} \cdot \ldots \cdot \frac{2}{3} \cdot \frac{2}{2} = \frac{2^{n-1}}{n!}$$

Best case: The element s turns out to be the k-th smallest element with probability $1/n$.

Expected complexity:

- Let $T(n)$ be an upper bound on the expected time to process a set S with n (or fewer) elements.
- Call s lucky if $|S_L| \leq \frac{3n}{4}$ and $< |S_R| \leq \frac{3n}{4}$
- Hence, s is lucky if it lies between 25th and the 75th percentile of S, which happens with probability $1/2$
- This gives us:

  $T(n) \leq \text{Time to partition} + \text{Maximum expected time for recursion}$

  $T(n) \leq n + \Pr(\text{s is lucky}) \cdot T(\frac{3n}{4}) + \Pr(\text{s is unlucky}) \cdot T(n)$

  $= n + \frac{1}{2}T(\frac{3n}{4}) + \frac{1}{2}T(n)$

Theorem (106)

A simple randomized algorithm solves Selection in expected linear time.

Counting Sort

- Counting Sort can be used for sorting an array A of n elements whose keys are integers within the range [0,k-1], for some, $n, k \in \mathbb{N}$.
- It is stable, but not in-place.

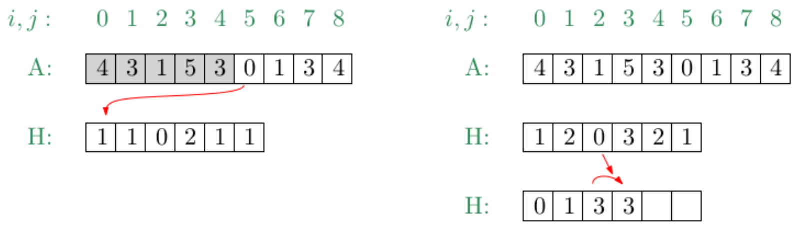
Stable: A sorting algorithm is said to be **stable** if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

In-place: An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only)

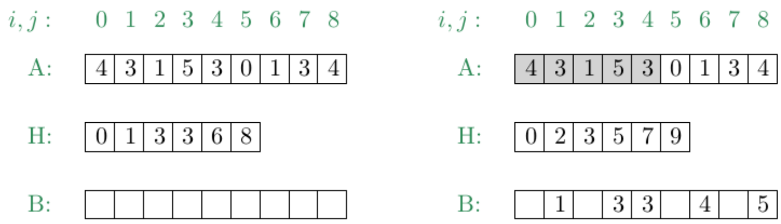
- It uses indices into an array and, thus, is not a comparison sort.

Algorithm:

1. Compute a histogram H of the number of times each element occurs within A.
2. For all possible keys, do a prefix sum computation on H to compute the starting index in the output array of the elements which have that key.



3. Move each element to its sorted position in the output array B.




```
CountingSort(array A[], array B[], array H[], int n, int k):

    for (i=0; i<k; ++i) H[i]=0
    for (j=0; j<k; ++j) H[A[j]] += 1

    total = 0
    for (i=0; i<k; ++i):
        oldCount = H[i]
        H[i] = total
        total += oldCount

    for (j=0; j<n; ++j):
        B[H[A[j]]] = A[j]
        H[A[j]] += 1
```

Theorem (107)

Counting Sort is a stable sorting algorithm that sorts an array of n elements whose keys are integers within the range [0, k − 1], for some $n, k \in \mathbb{N}$, within $O(n + k)$ time and space.

Radix Sort

- Radix Sort can be used for sorting an array A of n elements whose keys are d-digit (non-negative) integers, for some $n, d \in \mathbb{N}$.
- It compares keys on a per-digit basis and, thus, is NOT a comparison sort.
- It is stable, but not in-place

```
RadixSort(array A[], int n, int d):
    for (i = 1; i <= d; ++i):
        use stable sort to sort (counting sort) A[] relative to digit i
```

Theorem (108)

Radix Sort is a stable sorting algorithm that can be implemented to sort an array of n elements whose keys are formed by the Cartesian product of d digits, with each digit out of the range [0, k − 1], within $O(d(n+k))$ time and $O(n+k)$ space, for $n, d, k \in \mathbb{N}$.

Complexity:

- The for-loop runs d times (d digits). Hence, $O(d)$
- In every for-loop we apply counting sort. The complexity of counting sort is $O(n+k)$.

Hence, the overall complexity becomes $O(d(n+k))$.

Correctness proof:

IB:

Suppose that we have a number with $d = 1$ digits. Obviously, if we sort such numbers, we will obtain the right sorting.

IH:

After the k-th loop, where $k = 1, 2, \ldots, d$, the sequence is sorted on the lower k digits.

IS:

We will now proof it for the $(k+1)$-th step. Now, based on the induction hypthesis we assume that the numbers are sorted according to the first k-th numbers after the k-th loop. We now need to show that after the $(k+1)$-th loop, we know that the sequence is sorted according to the $(k+1)$-th number.

To prove this, let a_i, a_j two elements from A with the lower k digits of $a_i < a_j$.

- If the $(k+1)$-th digits of a_i and a_j are different, then the k-th digit of a_i must appear before a_j.
- If the $(k+1)$-th digits of a_i and a_j are the same, the right sorting depends on the other k elements. Our induction hypthesis guarantees that a_i appears before a_j. Since Counting sort is *stable* the order is preserved. a_i still appears before a_j after the $(k+1)$-th loop.

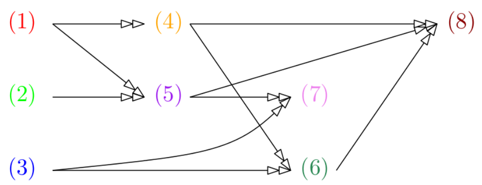
Therefore, we can conclude that the numbers are sorted correctly after the $(k+1)$-th loop.

Topological Sorting

Problem: TopologicalSorting

Problem: A directed graph $G = (V, E)$

Compute: A linear ordering of the vertices of V - if it exists - such that for all u,v$\in V$ the vertex u comes before the vertix v if E contains the directed edge uv.



$$\mathcal{L}: \quad (3) \to (2) \to (1) \to (5) \to (7) \to (4) \to (6) \to (8)$$
$$\mathcal{L}: \quad (1) \to (2) \to (3) \to (4) \to (5) \to (6) \to (7) \to (8)$$

Lemma (109)

A directed graph G admits a linear ordering of its vertices according to topological sorting if and only if G does not contain a directed cycle, i.e., if and only if G is a DAG.

Theorem (110)

In time $O(|V| + |E|)$ we can compute a linear ordering of the vertices of a directed graph $G = (V, E)$ according to topological sorting, or determine that the graph contains a directed cycle.

Algorithm

```
TopologicalSort(graph G=(V,E)) {
    L = {};
    S = list of all nodes of V with no incoming edges;
    while (S != {}) {
        remove front node u from S;
        add u to end of L;
        for (each edge e=(uv)) {
            remove edge e from E;
            if (v has no other incoming edges) {
                add v to end of S;
            }
        }
    }
```

```
    if (E != {})
        return error("graph has directed cycle");
    else
        return L;
}
```

- A depth-first search is an alternative to Kahn's algorithm
- Topological sorting can be used to solve the single-source shortest-path problem in a weighted directed graph in $O(|V| + |E|)$ time.