12. Medians and Order Statistics

**Definition 12: Orders and Medians**

Let \( A \) be a set of \( n \) values from a totally ordered set. The \( i^{th} \) order statistic is the \( i^{th} \) smallest value. For instance, the first order statistic is the minimum of \( A \). The maximum of \( A \) is the \( n^{th} \) order statistic. The median is the “halfway point.” If \( n \) is odd, then the median occurs at \( m = (n + 1)/2 \). If \( n \) is even, then there are two medians one at \( m_0 = n/2 \) and one at \( m_1 = n/2 + 1 \).

The *Selection Problem* is to find the \( i^{th} \) order statistic for a given \( i \), where \( 1 \leq i \leq n \).

**Problem 6: The Selection Problem**

Let \( n \) be a positive integer and let \( A \) be a set of \( n \) values from a totally ordered set.

*Decision Problem:* Is the value \( x \in A \) larger than exactly \( i - 1 \) other elements in \( A \)?

*Function Problem:* Let \( 1 \leq i \leq n \). Find the element \( x \in A \) that is larger than exactly \( i - 1 \) other elements in \( A \).

As an example, the elements in the set \( A \) have orders indicated below.

\[
A = \{7, \ 12, \ 5, \ 17, \ 9, \ 1, \ 14, \ 8, \ 18\}
\]

\[
\text{Order} = \{3, \ 6, \ 2, \ 8, \ 5, \ 1, \ 7, \ 4, \ 9\}
\]

A simple linear time algorithm that solves the minimum problem is shown below. It assumes \( A \) is represented as a 0-indexed array of \( n \) integers.

**Listing 35: Imperative Minimum**

```c
(Minimum algorithm 123)=
    int minimum(int A[], int n) {
        int min = A[0];
```
The time complexity of the above algorithm is $\Theta(n)$. The for loops $n - 1$ times taking a few cycles each time to decide whether or not to update the minimum. Since every element must be examined to determine the minimum, there can be no faster, deterministic algorithm.

A functional implementation to compute the maximum of a list might look like this:

The first line declares that the type of maximum to be a function that maps a list $[a]$ of orderable values to a value of type $a$. The two base cases are: an empty list has no maximum and a singleton list has the value of the single element. Then, recursively, the maximum of a longer list is the head of the list if the head is larger maximum of the tail, otherwise it is the maximum of the list's tail.

Listing 36: Functional Maximum

```haskell
maximum :: (Ord a) => [a] -> a
maximum [] = error "maximum of empty list"
maximum [x] = x
maximum (x:xs)
    | x > maxTail = x
    | otherwise    = maxTail
    where maxTail = maximum xs

- max and min are in class Ord of the Haskell Prelude
- in
max x y
    | x <= y = y
    | otherwise = x
min x y
    | x <= y = x
    | otherwise = y

{- And these functions in the PreludeList module:

Here the foldl ("left reduce") function maps a list to its maximum value. That is,
foldl max [a, b, c, d] = max ((max (max a b) c) d)
-}
maximum, minimum :: (Ord a) => [a] -> a
maximum [] = error "Prelude.maximum: empty list"
```
Randomized Selection

Using a randomized implementation of the partition function, described in the notes on Quicksort, an average case linear time algorithm can be developed for the i-th order problem. The randomizing heuristic is to swap the head of an array with a randomly selected element.

Listing 37: Imperative Randomized Partition

```c
#include <stdlib.h>
#include <stdio.h>

// partition is O(n) when lo = 0 and hi = n-1
// test need: A[j] > v for all j < hi
int partition(int A[], int lo, int hi) {
    int v, i, j, tmp;
    v = A[hi]; i = lo - 1; j = hi;
    for (;;) {
        while (A[++i] < v) ;
        while (A[--j] > v) ;
        if (i >= j) break;
        // swap A[i] and A[j]
        tmp = A[i];
        A[i] = A[j];
        A[j] = tmp;
    }
    // swap A[i] and A[hi]
    tmp = A[i];
    A[i] = A[hi];
    A[hi] = tmp;
    return i;
}

// randomPartition is O(1)
int randomPartition(int A[], int lo, int hi) {
    int k = rand() % (hi - lo + 1) + lo;
    int tmp = A[lo];
    A[lo] = A[k];
    A[k] = tmp;
```
```c
return partition(A, lo, hi);
}

t
```
Two functions are needed: one to partition about the head, and a second to make the head random.

127a \(\langle\text{Define partition about the head 127a}\rangle\) ≡

\[
\text{partition} :: \text{Ord } a \Rightarrow [a] \rightarrow ([a], \text{Int})
\]

\[
\text{partition} \ [\] = ([\], 0)
\]

\[
\text{partition} \ [p] = ([p], 1)
\]

\[
\text{partition} \ (p : xs) = (\text{before} ++ [p] ++ \text{after}, \text{length before})
\]

where \[
\text{before} = [x \mid x \leftarrow xs, x \leq p]
\]

\[
\text{after} = [x \mid x \leftarrow xs, x > p]
\]

127b \(\langle\text{Make the head random, then partition 127b}\rangle\) ≡

\[
\text{randomPartition} :: \text{Ord } a \Rightarrow [a] \rightarrow ([a], \text{Int})
\]

\[
\text{randomPartition} \ [\] = ([\], 0)
\]

\[
\text{randomPartition} \ [x] = ([x], 1)
\]

\[
\text{randomPartition} \ xs =
\]

let \( (k, _) = \text{randomR} \ (0, \text{length} \ xs) \) \((\text{mkStdGen} \ 10) :: (\text{Int}, \text{StdGen}) \)

\[
\in \text{let} \ (\text{first}, \text{second}) = \text{splitAt} \ k \ xs
\]

\[
in \text{partition} \ (\text{second} ++ \text{first})
\]

---

**Example: Randomize Partition**

*Given an array and index*

\[
A = \langle 7, 12, 5, 17, 9, 1, 14, 8, 18 \rangle \quad \text{and index} \quad i = 6
\]

The sought value, the \(6^{\text{th}}\) small element in the list has value 12.

Randomized selection might work something like this:

- Pretend the random partition was about index 4, value 9 creating the array

\[
\langle 7, 5, 1, 8, 9, 12, 17, 14, 18 \rangle
\]

- The value 9 occurs at the fifth order statistic, which is less than 6.

- Therefore, call random partition of the tail \(\langle 12, 17, 14, 18 \rangle\) and pretend 17 is randomly chosen as the pivot.

This results in the list \(\langle 12, 14, 17, 18 \rangle\) where 17 is of order 3 in the sub-list and order 5 + 3 = 8 in the original list.

- Next, because 6 < 8, call random partition on the list \(\langle 12, 14 \rangle\). Here 14 will be the pivot. It has order 2 and \(8 - 2 = 6\). Therefore, 12 is the \(6^{\text{th}}\) order statistic.
Listing 40: Randomized Selection

```c
#include <stdlib.h>

int randomSelect(int A[], int lo, int hi, int i) {
    if (lo == hi) { return A[lo]; }
    int q = randomPartition(A, lo, hi);
    int k = q - lo + 1;
    if (i == k) { return A[q]; }
    else {
        if (i < k) {
            return randomSelect(A, lo, q-1, i);
        }
        else {
            return randomSelect(A, q+1, hi, i-k);
        }
    }
}
```

In the best case, the array is partitioned at the halfway point each time. This leads to the recurrence relation

$$T(n) = T(n/2) + n, \quad T(1) = 0$$

which unrolls as:

$$T(n) = T(n/2) + n = T(n/4) + n/2 + n = T(n/8) + n/4 + n/2 + n \ldots = T(1) + n/2^{p-1} + \cdots + n/4 + n/2 + n \quad \text{for some } p = \log n$$

$$= 2n \left( 1 - \frac{1}{n} \right) = \Theta(n)$$

In the worst case, the array is always partitioned into a singleton and the rest of the array. This leads to the recurrence relation

$$T(n) = T(n - 1) + (n - 1), \quad T(1) = 0$$

which unrolls to the sum of the first $n$ natural numbers, that is

$$T(n) = \sum_{i=1}^{n-1} i = \binom{n-1}{2} = \Theta(n^2)$$

The textbook (Corman et al., 2009) contains a detailed analysis concluding that the average case time complexity is $O(n)$. It goes something like this.
Assume randomSelection returns any of the values 1 ≤ k ≤ n with equal likelihood, 1/n. It calls itself sub-array of size q or n−q−1. In the worst case, assume the call is always to the largest sub-array. Then,

\[ T(n) \leq \frac{1}{n} \sum_{q=1}^{n-2} T(\max(q, n-q-1)) + O(n) \]

Note q > n−q−1 implies q > (n−1)/2 and q ≤ n−q−1 implies q ≤ (n−1)/2

\[ \max(q, n-q-1) = \begin{cases} q & \text{if } q > \lceil (n-1)/2 \rceil \\ n-q-1 & \text{if } q \leq \lceil (n-1)/2 \rceil \end{cases} \]

For instance, if n is even, say n = 6, the the terms T(n−2), T(n−3),..., T([n−1]/2) occur twice in the sum.

And, when n is odd, say n = 7, the the terms T(n−2), T(n−3),..., T([n−1]/2 + 1) occur twice and T([n−1]/2) occurs once in the sum.

In all cases

\[ T(n) \leq \frac{2}{n} \sum_{q=\lceil n-1/2 \rceil}^{n-2} T(q) + O(n) \]

Assume T(n) ≤ cn for some constant c, and the O(n) term is an for some a. Then

\[ T(n) \leq \frac{2c}{n} \sum_{q=1}^{n-2} q - \sum_{q=1}^{\lceil n-1/2 \rceil - 1} q + an \]

\[ = \frac{2c}{n} \left( \frac{(n-2)(n-1)}{2} - \frac{\lceil (n-1)/2 \rceil - 1}{2} \lfloor (n-1)/2 \rfloor \right) + an \]

\[ \leq \frac{2c}{n} \left( \frac{(n-2)(n-1)}{2} - \frac{(n-1)/2 - 2}{2} \lfloor (n-1)/2 \rfloor \right) + an \]

\[ = cn - \left( \frac{cn}{2} - c + \frac{9c}{2n} - an \right) \]

\[ \leq cn - \left( \frac{cn}{2} - c - an \right) \]

Which is less than or equal to cn if

\[ \left( \frac{cn}{2} - c - an \right) \geq 0 \]

or

\[ n \geq \frac{2c}{c - 2a} \]

Thus, if T(n) = O(1) for n < 2c/(c - 2a) then the average case time complexity of random select is O(n).

Assume n = 6

max 1 (n−2) = 4
max 2 (n−3) = 3
max 3 (n−4) = 3
max 4 (n−5) = 4

Now assume n = 7

max 1 (n−2) = 5
max 2 (n−3) = 4
max 3 (n−4) = 3
max 4 (n−5) = 4
max 5 (n−6) = 5

The textbook (Corman et al., 2009) gives a more complex selection algorithm with worst case time complexity that is O(n).


Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition. [page 9], [page 14], [page 18], [page 21], [page 31], [page 33], [page 39], [page 53], [page 93], [page 105], [page 123], [page 141], [page 163], [page 169], [page 178], [page 187], [page 193], [page 201]


Haskell programming language, 20, 31, 44, 116, 137
Java programming language, 31

Ackermann function, 174
Axioms
Peano, 192

Bézout, Étienne, 47

C programming language, 30, 31, 42, 43, 55, 109, 152, 153
Calendar, 7
Cartesian product, 142
Catalan numbers, 156, 161
Computability, 191
Computable function, 28
Computer Language Benchmark Game, 30
Contradiction, 193

Dijkstra's algorithm, 183
DNA, 146

Empty
String λ, 61

Fibonacci, 44, 46, 142
Floating point
Machine epsilon, 50
Function
Ackermann, 174
Functional programming, 31

Gödel, Kurt, 192
Golden ratio, 142
Greatest common divisor, 47
Gutenberg Project, 84

Hash Tables, 93
Hofstadter, Douglas, 25

Kleene, Stephen, 53
Knuth, Donald, 30

Lamé, Gabriel, 44
λ calculus, 26, 28
Linear congruence, 47
Listing
C
Matrix Chain Multiplication, 158

Listings
C
A Minimum Algorithm, 135
aho-Corasick, 89
Boyer-Moore Pattern Matching, 72
Brute-force Left-to-Right Pattern Matching, 56
Bubble Sort, 109
Build a Heap, 126
Counting Sort, 128
Direct Address Table Operations, 94
Edit Distance, 152
Find Set with Path Compression, 177
Greatest Common Divisor, 42
Hash Table with Chaining, 95
Heap Sort, 127
Heapify a List, 126
Inner Product, 155
Insertion Sort, 112
Knuth–Morris–Pratt Pattern Matching, 67
Maximum Subsequence Sum (Cubic), 34
Maximum Subsequence Sum (Linear), 35
Merge Sort, 118
Morris–Pratt Pattern Matching, 58
Partition about a pivot, 120
Quick Sort, 120
Radix Exchange Sort, 132
Radix Sort, 129
Randomized Partition, 136
Randomized Selection, 138
Rational Knapsack, 167
Selection Sort, 114
Shell Sort, 116
Union by rank, 177

Haskell
Ackermann function, 174
Apply a Function Repeatedly, 50
Brute-force Left-to-Right Pattern Matching, 57
Bubble Function, 108
Bubble Sort, 109
Extended Euclidean Algorithm, 48
Fold from the left, 37
Greatest Common Divisor, 44
Inner Product, 155
Insertion Sort, 112
Matrix Multiplication, 155
Maximum Subsequence Sum, 37
Merge Sort, 117
Merging Sorted Lists, 117
Newton's Square Root Method, 51
Newton's Square Root Recurrence, 50
Quicksort, 120
Randomized Partition, 137
Relative Error Convergence Test, 50
Sample Code Header, 31
Second of a pair, 36
Selection Sort, 114
Sorted Decision Problem, 106