# 12. Medians and Order Statistics

Please read chapter 8 Medians and Order Statistics in the textbook (Corman et al., 2009).

**Definition 12: Orders and Medians**

*Let $\mathbb{A}$ 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 $\mathbb{A}$. The* maximum *of $\mathbb{A}$ 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 $1 \leq i \leq n$.

Describe an $\Theta(n \lg n)$ algorithm that solves the selection problem.

**Problem 6: The Selection Problem**

*Let $n$ be a positive integer and let $\mathbb{A}$ be a set of $n$ values from a totally ordered set.*
*<u>Decision Problem:</u> Is the value $x \in \mathbb{A}$ larger than exactly $i - 1$ other elements in $\mathbb{A}$?*
*<u>Function Problem:</u> Let $1 \leq i \leq n$. Find the element $x \in \mathbb{A}$ that is larger than exactly $i - 1$ other elements in $\mathbb{A}$.*

As an example, the elements in the set $\mathbb{A}$ have orders indicated below.

$$\begin{array}{rcccccccccc} \mathbb{A} = & \{7, & 12, & 5, & 17, & 9, & 1, & 14, & 8, & 18\} \\ \text{Order} = & \{3, & 6, & 2, & 8, & 5, & 1, & 7, & 4, & 9\} \end{array}$$

A simple linear time algorithm that solves the *minimum* problem is shown below. It assumes $\mathbb{A}$ is represented as a 0-indexed array of $n$ integers.

Describe an $\Theta(n)$ algorithm that simultaneously solve both the minimum and maximum problems.

**Listing 35: Imperative Minimum**

123 ⟨*Minimum algorithm* 123⟩≡

```
int minimum(int A[], int n) {
  int min = A[0];
```

```
    for (k = 1; k < n; k++) {
      if (min > A[k]) { min = A[k]; }
    }
    return min;
}
```

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**

124 ⟨*Maximum algorithm* 124⟩≡

```
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"
```

```
maximum xs = foldl1 max xs
minimum [] = error "Prelude.minimum: empty list"
minimum xs = foldl1 min xs
```

### *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**

125 ⟨*Randomized Partition* 125⟩≡

```
#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;
```

```
    return partition(A, lo, hi);
  }

  int main () {
     int A[10] = {21, 8, 13, 55, 34, 5, 3, 17, 12, 0};
     int p = randomPartition(A, 0, 9);
     printf("Partition about A[%d] = %d\n", p, A[p]);
     for (int i = 0; i < 10; i++) {
        printf("A[%d] = %d \n", i, A[i]);
     }
     return 0;
  }
```

This partition computation can be completed in $O(n)$ time in a language that support direct access to array elements.

For a list-based language such as HASKELL, swapping elements can take take linear time. The `splitAt k xs` splits xs at index `k` returning two list

Why is $O(k)$ the time complexity of `splitAt k xs`?

(xs[0..(k-1)], xs[k..n])

The time complexity of `splitAt k xs` is $O(k)$. Forming `(head list2):list1` and appending it `[x]` cost $O(1+k)$, and then another $O(k+1)$ steps to appending `tail list2`.

**Listing 38: Swap head with the `xs[k]`**

126a ⟨*Swapping head with k-th element* 126a⟩≡

```
import Data.List

swapElem :: Int -> [a] -> [a]
swapElem _ [] = []
swapElem _ [x] = [x]
swapElem k (x:xs) = (head list2):list1 ++ [x] ++ tail list2
   where (list1, list2) = splitAt (k-1) xs
```

A functional implementation of randomized partition might look like this.

**Listing 39: Functional Randomized Partition**

126b ⟨*Functional Randomized Partition* 126b⟩≡

```
import System.Random

⟨Define partition about the head 127a⟩
⟨Make the head random, then partition 127b⟩
```

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

127a ⟨*Define partition about the head* 127a⟩≡

```
partition :: Ord a => [a] -> ([a], Int)
partition [] = ([], 0)
partition [p] = ([p], 1)
partition (p:xs) = (before ++ [p] ++ after, length before)
            where before = [x | x <- xs, x <= p]
                  after  = [x | x <- xs, x > p]
```

127b ⟨*Make the head random, then partition* 127b⟩≡

```
randomPartition :: Ord a => [a] -> ([a], Int)
randomPartition []  = ([], 0)
randomPartition [x] = ([x], 1)
randomPartition xs  =
   let (k, _) = randomR (0, length xs) (mkStdGen 10) :: (Int, StdGen)
   in let (first, second) = splitAt k xs
      in partition (second ++ first)
```

**Example: Randomize Partition**

*Given an array and index*

$$\mathbb{A} = \langle 7, 12, 5, 17, 9, 1, 14, 8, 18 \rangle \quad \textit{and index} \quad i = 6$$

*The sought value, the* $6^{th}$ *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* $6^{th}$ *order statistic.*

**Listing 40: Randomized Selection**

128 ⟨*Randomized Selection* 128⟩≡

```
#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:

$$\begin{aligned}
T(n) &= T(n/2) + n \\
&= T(n/4) + n/2 + n \\
&= T(n/8) + n/4 + n/2 + n \\
&= \vdots \\
&= T(1) + n/2^{p-1} + \cdots + n/4 + n/2 + n \quad \text{for some } p = \lg n \\
&= 2n\left(1 - \frac{1}{n}\right) \\
&= O(n)
\end{aligned}$$

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

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

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` selection returns any of the values $1 \leq k \leq n$ with equal liklihood, $1/n$. 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 $q \leq n-q-1$ implies $q \leq (n-1)/2$

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

Assume $n = 6$

$\max 1\ (n-2) = 4$
$\max 2\ (n-3) = 3$
$\max 3\ (n-4) = 3$
$\max 4\ (n-5) = 4$

For instance, if $n$ is even, say $n = 6$, the the terms $T(n-2)$, $T(n-3), \ldots, T(\lceil (n-1)/2 \rceil)$ occur twice in the sum.

And, when $n$ is odd, say $n = 7$, the the terms $T(n-2)$, $T(n-3), \ldots, T(\lceil (n-1)/2 \rceil + 1)$ occur twice and $T(\lceil (n-1)/2 \rceil)$ occurs once in the sum.

Now assume $n = 7$

$\max 1\ (n-2) = 5$
$\max 2\ (n-3) = 4$
$\max 3\ (n-4) = 3$
$\max 4\ (n-5) = 4$
$\max 5\ (n-6) = 5$

In all cases

$$T(n) \leq \frac{2}{n}\sum_{q=\lfloor n-1 \rfloor/2}^{n-2} T(q) + O(n)$$

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

$$\begin{aligned}
T(n) &\leq \frac{2c}{n}\sum_{q=1}^{n-2} q - \sum_{q=1}^{\lfloor n-1 \rfloor/2-1} q + an \\
&= \frac{2c}{n}\left(\frac{(n-2)(n-1)}{2} - \frac{(\lfloor (n-1)/2 \rfloor - 1)(\lfloor (n-1)/2 \rfloor)}{2}\right) + an \\
&\leq \frac{2c}{n}\left(\frac{(n-2)(n-1)}{2} - \frac{((n-1)/2-2)((n-1)/2-1)}{2}\right) + an \\
&= cn - \left(\frac{cn}{2} - c + \frac{9c}{2n} - an\right) \\
&\leq cn - \left(\frac{cn}{2} - c - an\right)
\end{aligned}$$

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 $O(n)$.

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

# *Bibliography*


Agrawal, M., Kayal, N., and Saxena, N. (2002). Primes is in P. http://www.cse.iitk.ac.in/news/primality.html. [page 205]

Aho, A. V. and Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340. [page 87]

Apostolico, A. and Galil, Z. (1997). *Pattern Matching Algorithms*. Oxford University Press. [page 53]

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press. [page 141]

Bentley, J. (1984a). Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873. [page 33]

Bentley, J. (1984b). Programming Pearls: How to sort. *Commun. ACM*, 27(4):287–ff.

Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772. [page 71]

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM. [page 199]

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]

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of Intractability*. W. H. Freeman. [page 193], [page 203]

Gödel, K. (1930). Die vollst andigkeit der axiome des logischen funktionenkalk uls. *Monatshefte f ur Mathematik und Physik*, 37:349–360. [page 194]

Gödel, K. (1992). *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover books on advanced mathematics. Dover Publications. [page 194]

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. EBL-Schweitzer. Cambridge University Press. [page 53]

Hoare, C. A. R. (1961). Quicksort. *Computer Journal*, 5(1):10–15.

Hofstadter, D. R. (1999). *Gödel, Escher, Bach: An Eternal Golden Braid. 20th Anniversary Edition*. Basic Books, New York, New York. [page 25]

Kernighan, B. W. and Ritchie, D. M. (1988). *The* C *Programming Language*. Prentice Hall, second edition. [page 31]

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2):97–111. [page 30]

Knuth, D. E. (1993). *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley, ACM Press. [page 147], [page 163]

Knuth, D. E., Morris, J. H., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):240–267. [page 55]

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition. [page 31]

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley. [page 193], [page 195], [page 206]

Patterson, D. A. and Hennessy, J. L. (1996). *Computer Architecture: A Quantitive Approach*. Morgan Kaufmann Publishers, Inc., second edition. [page 30]

Polya, G. (1945). *How To Solve It*. Princeton University Press. ISBN 0-692-02356-5. [page 26]

Rabhi, F. A. and Lapalme, G. (1999). *Algorithms: A Functional Programming Approach*. Addison-Wesley. [page 31]

Ramsey, N. (1994). Literate programming simplified. *IEEE Software*, 11(5):97 – 105. [page 30]

Sankoff, D. and Kruskal, J. B., editors (1983). *Time Warps, String Edits, and Macromolecules*. Addison-Wesley, Reading, Massachusetts. [page 146]

Sedgewick, R. (1977). The analysis of quicksort programs. *Acta Informatica*, 7:327–355.

Sedgewick, R. (1978). Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857.

Sedgewick, R. (2004). *Algorithms in Java*. Addison-Wesley. [page 31]

Shallit, J. (1994). Origins of the Analysis of the Euclidean Algorithm. *Historia Mathematica*, 21:401–419. [page 44]

Stansifer, R. (1984). Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department. [page 194]

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265. [page 197]

Wilf, H. (2006). *Generatingfunctionology: Third Edition*. AK Peters Series. Taylor & Francis. [page 19]

# *Index*