Divide and Conquer: Finding Median in Linear Time[1]

- In this lecture we see a beautiful application of the divide and conquer paradigm. We see an $O(n)$ time algorithm to find the median of n numbers. This algorithm was published in a paper[2] in 1973; an interesting feature of this paper is that four out of the five authors have won the Turing Award (though this result isn't what they won it for).

- We actually a look at the more general problem of *selection*. The input is an unsorted array/list $A[1:n]$ of (distinct) integers/reals, and a parameter $1 \leq k \leq n$. The objective is to find the kth smallest number. There is a trivial $O(nk)$ time algorithm, and in a previous lecture, we saw a faster algorithm. However, when $k = \Theta(n)$, that algorithm still took $O(n \log n)$ time; and an $O(n \log n)$ algorithm is trivial by sorting. We see an algorithm for solving the selection problem for any k in $O(n)$ time.

- **Idea 1: Pivoting to reduce space.** The first idea is one from a different sorting algorithm called QuickSort: this idea is pivoting. To illustrate this, let us pick an element of the array $a = A[i]$; think of i right now as *arbitrary*. The PIVOT operation takes A and a and generates two lists B and C (all this can be done in-place) such that B contains all the elements $< a$ and C contains all the elements $> a$ (we are assuming distinct elements). This can be done with one-scan and takes $O(n)$ time.

```
procedure PIVOT(A[1 : n], a):
    ▷ Return B and C which contains elements of A which are < a and > a respectively.
    for 1 ≤ j ≤ n do:
        if A[j] < a then:
            Add A[j] to B
        else if A[j] > a then:
            Add A[j] to C
    return (B, C).
```

- What does this buy us? First it tells us the *rank* of this particular element a. In particular, if $|B| = r$, then the rank of a is $(r+1)$; it is the $(r+1)$th smallest element.

  And what does this buy us for the selection problem? If we were *extremely* lucky and $r+1$ happened to be k, then a is the kth smallest element and there was nothing more to do. However, if $r+1 < k$, then (i) the kth smallest element belongs in present in C, and furthermore, (ii) it is the $(k-r-1)$th smallest in C. And, if $r+1 > k$, well then (i) the kth smallest element is in B, and (ii) it is the kth smallest element there. Therefore, in either case, we are handed a selection problem on a smaller array.

  How much smaller are the arrays? This depends on the size of B and C which itself depends on *what the pivot is*; we haven't discussed this process at all. For now, imagine there is a sub-routine

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 24th Jan, 2025*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

[2] Blum, M.; Floyd, R. W.; Pratt, V. R.; Rivest, R. L.; Tarjan, R. E. (August 1973). *"Time bounds for selection"*. Journal of Computer and System Sciences. 7 (4): 448–461.

FINDPIVOT which takes an array $A[1:n]$ and returns a pivot $A[i]$. For example, it could be as trivial as returning $i = 1$ all the time. But if we had our hands such a routine, here is the recursive algorithm for selection that arises out of this pivoting idea.

```
1: procedure SELECT(A[1 : n], k):  ▷ assume 1 ≤ k ≤ n
2:     ▷ Assumes existence of FINDPIVOT(A)
3:     if n = 1 then: ▷ k = 1 also then
4:         return A[1]
5:     p ←FINDPIVOT(A) ▷ find a pivot
6:     (B, C) ←PIVOT(A, p)
7:     if |B| = k − 1 then:
8:         return p
9:     else if |B| < k − 1 then:
10:        SELECT(C, k − |B| − 1)
11:    else: ▷ ie, |B| ≥ k
12:        SELECT(B, k)
```

- **Analysis of SELECT.**

  Let's try to analyze SELECT; in this process we will discover what a *good* FINDPIVOT-routine would look like. For now, we proceed as we have proceeded before. As usual, we say $T(n)$ is the worst running time of SELECT when run on *any* A of length at most n and *any* $1 \leq k \leq n$.

  We see that Line 5 takes time which depends on the pivot finding algorithm. Let's call this $P(n)$ for now; it's a function whose properties will dictate the runtime of $T(n)$. Line 6 takes $O(n)$ time. The recursive call at Line 10, if run, takes $T(|C|)$ time, and the recursive call at Line 12 takes $T(|B|)$ time. Then the recurrence becomes

$$T(1) = O(1); \quad \forall n \geq 2, \quad T(n) \leq \; P(n) \; + \; O(n) \; + \max\left(T(|B|), T(|C|)\right) \tag{1}$$

- Let us engage in some wishful thinking: what would be great for us. Firstly, since we are shooting for an overall $O(n)$ runtime, the function $P(n)$ should be $O(n)$. Let's say this is true. Indeed, the simpleminded algorithm which returns the first element as the pivot is an $O(1)$ time algorithm; so such candidates do exist.

  What is actually the problematic bit is $\max\left(T(|B|), T(|C|)\right)$. We know that $|B| + |C| < n$. If both of these were "roughly equal" in size, then $|B|$ and $|C|$ would both $\leq n/2$. Then, we would get that (1) becomes

$$T(n) \leq O(n) + T(n/2) \quad \text{which is great since it evaluates to} \quad O(n)$$

  Indeed even if $\max(|B|, |C|) \leq 9n/10$, *even then* we would be in great shape: the recurrence would be $T(n) \leq O(n) + T(9n/10)$ which also, by Master theorem, evaluates to $O(n)$. This lets us crystallize what the properties of a *good* FINDPIVOT would be.

  a. It should run in $P(n) = O(n)$ time.

  b. It should return a pivot p such that PIVOT(A, p) returns lists which are both $\Theta(n)$ sized.

- Before we describe how the paper mentioned above does it, let's actually say a simple procedure which gets both of the above *with high probability*: simply pick a pivot at *random* from A. (a) is obvious, and (b) occurs with constant probability. More precisely, the chance that we get a pivot which falls in the "middle third" is $1/3$ and if it does so, then $|B|$ and $|C|$ are both of size $\leq 2n/3$. And that's great. However, this is a randomized algorithm; can we obtain such a nice pivot *deterministically*?

- **Idea 2: Good** FINDPIVOT **by Median-of-Medians.** To recap, we wish to design a deterministic algorithm which for any array finds a pivot element which (a) breaks array into "balanced" pieces (the sizes B and C are both $\Theta(n)$ size), and (b) one can find this pivot in $O(n)$ time. To solve problem (a), the "median-of-median" algorithm by Blum, Floyd, Pratt, Rivest and Tarjan uses *recursion* again! In retrospect, the idea is simple. The run time would be $O(n)$ plus a recursive call...but that won't matter as you will see.

  We divide the array A into $n/5$ "quintets" each with 5 elements. In each piece, we find the median using brute force; this takes $O(n)$ time. Let M be the set of the array of these medians; note that M has $n/5$ elements. FINDPIVOT returns the median of M by calling SELECT on it (with $k = n/10$).

  Why is this a good idea? Let m be the median of M. There are $\approx n/10$ elements of M which are smaller than m. Furthermore, for each of these elements of M, there are 2 more elements smaller than it (coming from the corresponding quintet). So, all in all, there are *at least* $3n/10$ elements of A smaller than m. And so, the rank of m is *at least* $3n/10$. An analogous argument also shows at least $3n/10$ elements of A larger than m; and so the rank of m is *at most* $7n/10$. And this means that if we call PIVOT(A, m) to get (B, C), both pieces are at most $7n/10$ in size.

  Here is the algorithm in its full glory.

```
1: procedure LINEARSELECT(A[1 : n], k): ▷ assume 1 ≤ k ≤ n
2:     if n = 1 then:▷ k = 1 also then
3:         return A[1]
4:     Break A into n/5 quintets A_1, . . . , A_{n/5}
5:     M ← [ ].
6:     for 1 ≤ t ≤ n/5 do: ▷ O(n) time for-loop
7:         Find median of A_t in O(1) time and put in M.
8:     m ← LINEARSELECT(M, n/10) ▷ Recursively find median of M
9:     (B, C) ←PIVOT(A, m)
10:    if |B| = k − 1 then:
11:        return m
12:    else if |B| < k − 1 then:
13:        LINEARSELECT(C, k − |B| − 1)
14:    else:▷ ie, |B| ≥ k
15:        LINEARSELECT(B, k)
```

- **Recurrence Inequality and Solution.** The recurrence inequality governing the running time is found as follows. Fix any array $A[1:n]$ and k. Line 4 to Line 7 takes $O(n)$ time. The recursive call in Line 8 takes $\leq T(n/5)$ time since $|M| = n/5$. Line Line 9 takes $O(n)$ time. As explained above, by design, $|B|$ and $|C|$ are both of size $\leq 7n/10$. Therefore, either of the lines, Line 13 and Line 15, takes at most $T(7n/10)$ time. Together, we get

$$T(1) = O(1); \quad \forall n \geq 2, \quad T(n) \leq \; O(n) \; + T\left(\frac{n}{5}\right) \; + \; T\left(\frac{7n}{10}\right) \tag{2}$$

The above can't be solved by the master theorem, but the kitty method shows that it evaluates to $O(n)$. Indeed, it's not hard to establish this inductively. Suppose $T(1) \leq C$, and $T(n) \leq T(n/5) + T(7n/10) + Cn$. Then,

Claim 1. $T(n) \leq 10Cn$.

Proof. Base case is obvious. Assume the above is true for all $1 \leq k \leq n-1$, and we need to prove $T(n) \leq 10Cn$. By the above recurrence, we get

$$T(n) \leq \quad T(n/5) + T(7n/10) + Cn \quad \underbrace{\leq}_{\text{Induction Hypothesis}} \quad 10C \cdot n/5 + 10C \cdot 7n/10 + Cn = 10n \quad \square$$

- **Final Remarks.** One may ask what the coefficient in front of n is if we are only interested in the *number of comparisons*? The above analysis would give a coefficient which is ≈ 20. In their paper, Blum and others actually showed a more detailed procedure with this coefficient under 6. In 1976, a paper[3] by Schönage, Paterson, and Pippenger described an algorithm making at most $3n$ comparisons. This remained the state of affairs till a paper[4] by Dor and Zwick which gave a $\leq 2.95n$ query algorithm to find the median.

  There are known *lower bounds* too. Bent and John, in 1985, showed[5] that any correct algorithm for finding the median needs to make at least $2n$ comparisons. Dor and Zwick, in a different paper[6], improved that to show there exists a *constant* ε_0 such that any correct median finding algorithm must make at least $(2 + \varepsilon_0)n$ comparisons. In their paper, Dor and Zwick establish this for $\varepsilon_0 \approx 2^{-80}$, although the main message is that 2 is *not* the correct coefficient. This is the current state of the art as far as I know.

---

[3] A. Schönhage, M. Paterson, and N. Pippenger. *Finding the median.* Journal of Computer and System Sciences, 13:184–199, 1976

[4] D. Dor and U. Zwick. *Selecting the Median*, SIAM Journal on Computing, 28, 1722-1758, 1999

[5] S. W. Bent and J. W. John, *Finding the median requires* $2n$ *comparisons*, in Proceedings of the 17th Annual ACM Symposium on Theory of Computing, Providence, RI, 1985, pp. 213–216.

[6] D. Dor and U. Zwick, *Median Selection Requires* $(2 + \epsilon)n$ *Comparisons*, SIAM Journal on Discrete Mathematics, 14(3):312–325, 2001