Linear Time Sorting and Selection

S. Halim YJ. Chang

School of Computing
National University of Singapore

CS3230 Lec12; Tue, 05 Nov 2024

Overview

Recap: Sorting Problems

Sorting: Given an array $A = [a_1, a_2, \ldots, a_n]$ of n elements, sort the elements in A in non-decreasing order, i.e., find a permutation $[a'_1, a'_2, \ldots, a'_n]$ of A such that $a'_1 \le a'_2 \le \ldots \le a'_n$.

In CS1101S/eq, CS2040S/eq, and now CS3230,
we have learned many sorting algorithms,
see `https://visualgo.net/en/sorting`.

Using decision tree model, we can show that the lower bound of *comparison-based* sorting algorithms is $\Omega(n \log n)$.

Can we break this lower bound... if we do not compare?

Counting Sort - unstable version

This algorithm is very easy to explain and to implement.
Assumption: Each element in A are integers in range $[0..k]$.
Use `https://visualgo.net/en/sorting?mode=Counting`
Click 'Sort' and then click 'Simple'.

Very short Python code

PS: DAT is “Direct Addressing Table”.

```
C = [0]*(k+1) # create the frequency DAT [0..k], O(k)
for Ai in A: # O(n), at the end, C[i] = frequency of i
    C[Ai] += 1
for i in range(k+1): # O(k+n)
    print(*[i] * C[i], end=' ') # set format yourself
```

No comparisons between element $\rightarrow$ not comparison-based sorting.
We trade-off memory (DAT) to break the $\Omega(n \log n)$ lower bound.
If $k \in O(n)$, then the $O(n+k)$ Counting Sort takes $O(n)$ time.
PS: All analysis can be made tighter from O to Θ.

Main Issue of Counting Sort

It requires big memory for the frequency counter DAT C.
If k is big, we may not be able to create a DAT C to cover $[0..k]$.

We want to set k big enough (vs the available working memory), but not too big that we cannot even run the algorithm.

Counting Sort - the stable version

For completeness, Counting Sort can be made to be stable.
Use `https://visualgo.net/en/sorting?mode=Counting`
Click ‘Sort’ and then click ‘Stable’.

```
C = [0]*(k+1) # create the frequency DAT [0..k], O(k)
for Ai in A: # O(n), at the end, C[i] = frequency of i
    C[Ai] += 1

for i in range(1, k+1): # O(k), C[i] = num ints <= i
    C[i] += C[i-1] # this is also called the 'prefix sum'
B = [0] * n # output (sorted) array
for i in range(n-1, -1, -1): # O(n), from back to front
    C[A[i]] -= 1 # decrease first (0-based indexing)
    B[C[A[i]]] = A[i] # A[i] is placed at C[A[i]] in B
print(*B) # B is the sorted version of A, and stable
```

Radix Sort - Iterated Counting Sort

Herman Hollerith's card-sorting machine for 1890 US census
Key idea: Sort digits by digits

Hollerith's 'bad' idea: sort Most Significant Digit (MSD) first.
Better idea: Sort Least Significant Digit (LSD) first.

The sorting of each digit requires a **fast stable sort**.
Each pass uses the stable version of Counting Sort.

Radix Sort - Pseudocode and Animation

```
// suppose there are d digits
for i <- 1 to d // note: LSD is from right to left
    sort by the i-th LSD (using stable Counting Sort)
```

Use `https://visualgo.net/en/sorting?mode=Radix`
Click Sort.

Proof of Correctness

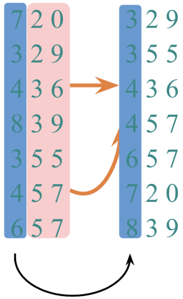
It may not be immediately obvious that Radix Sort is correct.

We can use proof by induction:
When we have sorted the i-th LSD, then the integers are sorted according to their values on the i-th LSD.

$P(1)$ holds, as Radix Sort first pass will correctly sort the LSD (we are sure the stable Counting Sort sub-routine is correct).

Suppose $P(i-1)$ holds, we can show $P(i)$ holds too:
The integers are sorted based on the i-th LSD on the i-th pass.
Now, due to the stable sort subroutine used, within the group of integers that have the *same* i-th LSD, the stable sort algorithm does *not* change their relative position!
Thus, these integers are correctly sorted!

Proof of Correctness - Illustration



Question 1 at VisuAlgo Online Quiz

What if we use stable Merge Sort as sub-routine of Radix Sort?

A). It works, and faster

B). It works, but slower

C). It does not work

Radix Sort - Analysis (1)

If we have d digits, Radix Sort runs d iterations of $O(n+k)$ Counting Sort, thus the overall time complexity in $O(d \cdot (n+k))$.

Setting $k = 10$ (digit [0..9], or base 10/Decimal), as illustrated in the default VisuAlgo sorting visualization and in many other Computer Science textbooks/websites as it aids early understanding of Radix Sort, is often **not** the best setup.

If b-bit word is broken into $\frac{b}{r}$ groups of r-bit words, then:

- There are only $d = \frac{b}{r}$ passes now.
- Each pass takes time $O(n + 2^r)$, as $k = 2^r$.
- The total time is thus $O(\frac{b}{r}(n + 2^r))$.

Radix Sort - Analysis (2)

Choose r to minimize $O(\frac{b}{r}(n+2^r))$.

The optimal $r \approx \log_2 n$ (so that $(n+2^r)$ balances).

The total time is thus $O(\frac{b}{\log_2 n} \cdot (n + 2^{\log_2 n}) = O(\frac{b \cdot n}{\log_2 n})$

Now, if the integers are in the range of $[0..n^d]$,
then $b = \log_2 n^d$ bits, or $b = d \cdot \log_2 n$ bits.

Thus, Radix Sort runs in time $O(\frac{d \cdot \log_2 n \cdot n}{\log_2 n}) = O(d \cdot n)$ – linear time

Natural Break Point

Selection/Order Statistics - Definition

The selection problem, or order statistics, is defined as follows: Given an *unsorted* array A without duplicates, find the *index of the* i-th smallest/i-th order statistics/rank i element in A.

A few special cases:

- $i = 1$, find the minimum element
- $i = \lfloor \frac{n+1}{2} \rfloor$ (or $i = \lceil \frac{n+1}{2} \rceil$), find the median
- $i = n$, find the maximum element

Example, suppose the array $A = \{13, 5, 8, 7, 4\}$ with $n = 5$:

- $i = 1 : 5$, as $A[5] = 4$ is the minimum element
- $i = 3 : 4$, as $A[4] = 7$ is the median (3-rd smallest) element
- $i = 5 : 1$, as $A[1] = 13$ is the maximum element
- $i = 4 :$ (can you find it?)

Selection - Avoiding Duplicates

In this lecture, we will assume that all elements are distinct to simplify our discussion and analysis.

We can transform an array that contains duplicates into one that is distinct (how?)

Selection - Returning Index vs Actual Element

We reckon that **Select(A, i)** should return the index k of the i-th smallest element of A, instead of returning the i-th smallest element x itself.

This is because returning index is a bit more meaningful, as we can infer that $x = A[k]$ whereas we cannot easily do the reverse (infer k given x – recall that A is an unsorted array).

Selection - Naive Solution(s)

If we do not sort the array first, we may need $O(i \cdot n)$ algorithm.

Go to `https://visualgo.net/en/array?mode=sorted-array` and click ‘Select’.

If we first sort the array using a comparison-based sorting algorithm, we can solve the selection problem in $\Theta(n \log n)$

Can we do better than $\Theta(n \log n)$?
PS: assume we can only compare elements
(so, we rule-out linear-time sort).

Lower bound of Selection is $\geq n$ steps

Selecting the i-th element in an unsorted array requires $\geq n$ steps.

Proof by contradiction: Suppose that we can find the i-th smallest element in an unsorted array using $< n$ steps. (can you arrive at contradiction?).

Can we find the min/max in $o(n \log n)$

YES!

```
function FindMin(A[1..n]) // FindMax(A[1..n]) is similar
    i = 1 // assume A[1] is the minimum
    for j = 2 to n do
        if A[j] < A[i] // A[j] is smaller
            i = j // remember this index
    return i
```

The above algorithm runs in $\Theta(n)$, same as the lower bound, and thus it is the best possible (for an unsorted array).

Can we find the i-th smallest element in $o(n \log n)$?

The answer is also **YES!**.

We will present two solutions:

The first one is much easier to explain and analyze. It is a *randomized* Divide and Conquer (D&C) algorithm based on the randomized partition algorithm of the randomized Quicksort. It is called Quickselect. As it is simpler, we will only show the outline and defer the analysis in the last tutorial 11.

The second one has a worst-case linear time guarantee.

Quickselect - Pseudocode

```
function QuickSelect(A, l, r, i)
    if l == r, return l
    k = Randomized_Partition(A, l, r) // random pivot
    if i == k, then return k // found
    else if i < k, then return QuickSelect(A, l, k-1, i)
    else, then return QuickSelect(A, k+1, r, i)
```

Quickselect is invented by the same author of Quicksort, Tony Hoare (1961).

Go to `https://visualgo.net/en/array?mode=array` and click ‘Select’, set k[1], and then ‘Quickselect’.

[1] In some literature, we select k-th, not i-th, smallest element.

Quickselect - Expected Linear-Time Selection

The best case for Quickselect is $O(n)$, i.e., in one pass of `Randomized_Partition`, we find that pivot has rank $k = i$.

If the partition sub-routine is *not* randomized, the worst case for Quickselect is similar with the worst case QuickSort, i.e., $O(n^2)$.

The analysis of Quickselect that uses `Randomized_Partition` is deferred to tut11.

Question 2 at VisuAlgo Online Quiz

Who is the master of algorithms pictured below?

A). Robert Tarjan

B). Robert Floyd

C). Richard Karp

D). Tony Hoare

Worst-case Linear-Time Select

The worst-case linear-time algorithm to select the rank-i element is due to Blum, Floyd, Pratt, Rivest, and Tarjan (1973), ≈ 12 years after the invention of Quickselect.

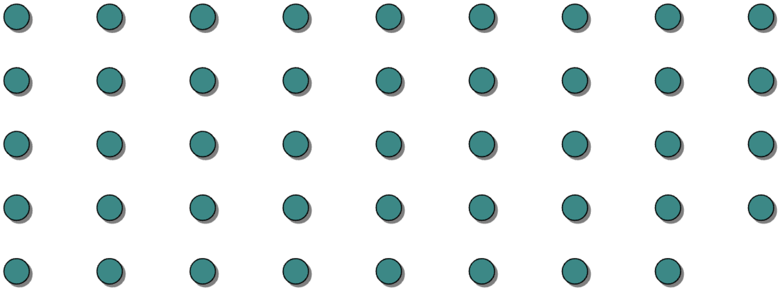
It is sometimes called the 'median of medians' selection algorithm.

The top side of algorithm is different,
but the bottom side is identical with QuickSelect.

Worst-case Linear-Time Select - Pseudocode

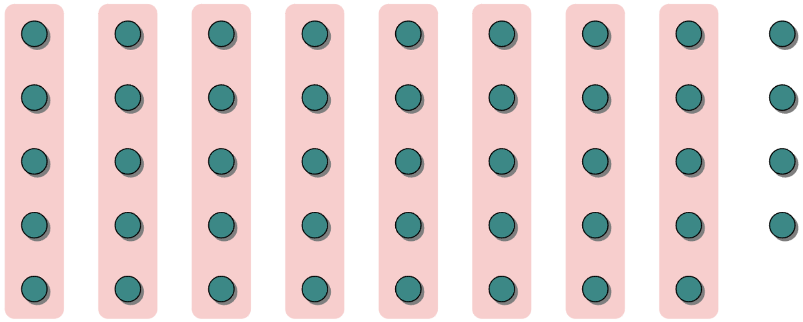
```
function Select(A, n, i)
    // step 1
    divide A into ceil(n/5) groups of 5 elements each
    find the median of each 5-elements group in O(1)
    // step 2
    Let B be ceil(n/5) medians of each groups
    Select(B, |B|, |B|/2)) // find median of medians
    // step 3
    k = Partition(A, x) // use this specific pivot x
    Let A' / A'' be the elements < x / > x, respectively
    // step 4-5-6, similar as Quickselect
    if i == k, then return k // found
    else if i < k, then return Select(A', k-1, i)
    else, return Select(A'', n-k, i-k)
```

Choosing the pivot (1)



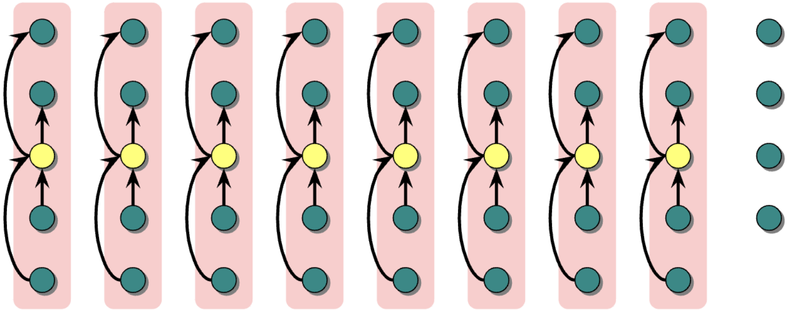
Suppose there are n elements, e.g., $n = 44$ elements in this picture.
How to find a 'good pivot' among these n elements?

Choosing the pivot (2)



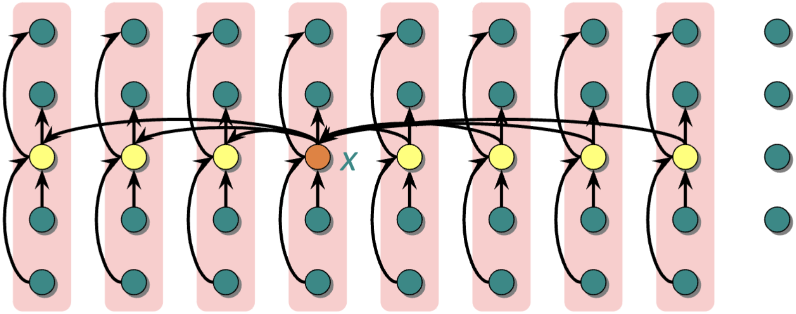
Divide the n elements into groups of 5.
There are $\lfloor \frac{n}{5} \rfloor$ such groups, e.g., there are $\lfloor \frac{44}{5} \rfloor = 8$ groups of 5.

Choosing the pivot (3)



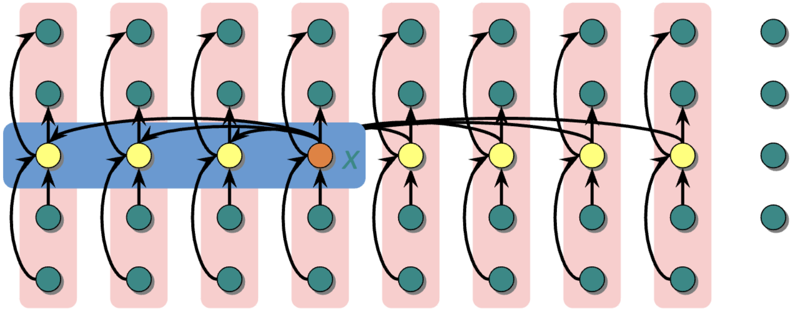
Find the median of each group in $O(1)$, e.g., sort and get index 2. Sorting 5 elements is in $O(1)$. There are $\lfloor \frac{n}{5} \rfloor$ such medians.

Choosing the pivot (4)



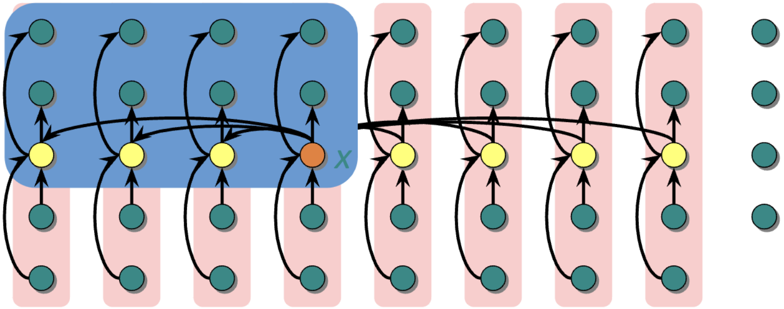
Recursively Select the median x of the $\lfloor \frac{n}{5} \rfloor$ group medians.
This x, the median of medians, will be the pivot.

Analysis (1)



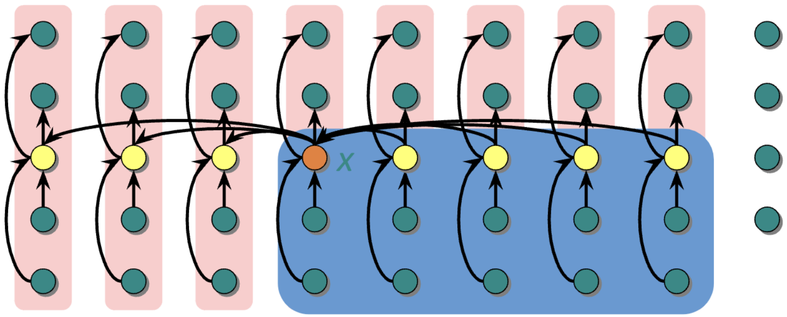
At least half of the group of 5 medians are $\leq x$, which is at least $\lfloor \lfloor \frac{n}{5} \rfloor / 2 \rfloor = \lfloor \frac{n}{10} \rfloor$ elements, e.g., $\lfloor \frac{44}{10} \rfloor = 4$ in this example.

Analysis (2)



Therefore, at least $3 \cdot \lfloor \frac{n}{10} \rfloor$ elements are $\leq x$ (see the highlight).
There are $\geq 3 \cdot \lfloor \frac{44}{10} \rfloor = 12$ in this example.

Analysis (3)



Similarly, at least $3 \cdot \lfloor \frac{n}{10} \rfloor$ elements are $\geq x$ (see the highlight).
There are $\geq 3 \cdot \lfloor \frac{44}{10} \rfloor = 12$ (actually 15) in this example.

Analysis (4)

For large n, since at least $3 \cdot \lfloor \frac{n}{10} \rfloor$ elements are $\leq x$ (or $\geq x$), after partitioning A around x, the recursive call to Select (step 5-6) is executed recursively on at most $n - \frac{3n}{10} = \frac{7n}{10}$ elements.

Thus, the recurrence for running time $T(n)$ takes time of not more than $T(\frac{n}{5}) + T(\frac{7n}{10}) + \Theta(n)$ in the worst case.

For small n, we can compute $T(n) \in \Theta(1)$.

Worst-case Linear-Time Select - the recurrence

```
function Select(A, n, i)
    // step 1, Theta(n)
    divide A into ceil(n/5) groups of 5 elements each
    find the median of each 5-elements group in O(1)
    // step 2, T(n/5)
    Let B be ceil(n/5) medians of each groups
    Select(B, |B|, |B|/2)) // find median of medians
    // step 3, Theta(n)
    k = Partition(A, x) // use this specific pivot x
    Let A' / A'' be the elements < x / > x, respectively
    // step 4-5-6, similar as Quickselect, T(7n/10)
    if i == k, then return k // found
    else if i < k, then return Select(A', k-1, i)
    else, return Select(A'', n-k, i-k)
```

Solving the recurrence

Overall, we have $T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + \Theta(n)$.

We can use recursion tree to analyze the recurrence (exercise), or
We can use substitution method, i.e., we see if $T(n) \leq c \cdot n$?

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

$T(n) \leq c \cdot \frac{9n}{10} + \Theta(n)$

$T(n) \leq c \cdot n - (c \cdot \frac{n}{10} - \Theta(n))$

$T(n) \leq c \cdot n$,
if c is large enough to handle both the $\Theta(n)$ and initial conditions.

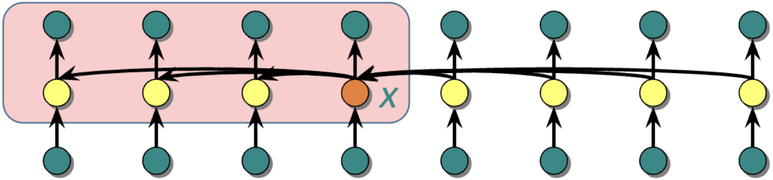
Conclusion

The **worst-case linear-time** selection is only good in theory.
In practice, this algorithm is a bit slow due to the constant c of $c \cdot n$ is large.

The **expected linear-time** Quickselect is far more practical.

Question 3 at VisuAlgo Online Quiz

The worst-case linear-time selection algorithm that we have just seen divides the array into groups of 5. Now suppose, we divide the array into groups of 3 instead. Will this modified algorithm works in worst-case linear-time too?



A). YES

B). Unfortunately no

Selection - Dynamic Order Statistics

In this lecture, we assume that the elements of array A are static.
But, in practice, new elements can be added and existing elements can be deleted or updated, thus disrupting existing order statistics.
Example: a new smallest element $\rightarrow$ existing elements get $+1$ rank.
This is called the *Dynamic* Order Statistics problem.

A potential solution: Balanced BST (bBST) augmentation (a.k.a. Order Statistics Tree). A bBST has $O(\log n)$ insertion/deletion (and update). With proper size-of-subtree augmentation, we can also do $O(\log n)$ select and rank operations.
Review `https://visualgo.net/en/avl` and click ‘Select(k)’.

Acknowledgement

The slides are modified from previous editions of this course and similar course elsewhere

List of credits: Erik D. Demaine, Charles E. Leiserson, Surender Baswana, Leong Hon Wai, Lee Wee Sun, Ken Sung, Arnab Battacharya, Diptarka Chakraborty, Sanjay Jain.