Order Statistics

We often want to compute a **median** of a list of values.
(It gives a more accurate picture than the average sometimes.)

More generally, what element has position k in the sorted list?
(For example, for percentiles or trimmed means.)

Selection Problem

Given a list A of size n, and an integer k,
what element is at position k in the sorted list?

Sorting-Based Solutions

- First idea: Sort, then look-up

- Second idea: Cut-off selection sort

Heap-Based Solutions

- First idea: Use a size-k max-heap

- Second idea: Use a size-n min-heap

Algorithm Design

What **algorithm design paradigms** could we use to attack the selection problem?

- Reduction to known problem
  What we just did!
- Memoization/Dynamic Programming
  Would need a recursive algorithm first...
- Divide and Conquer
  Like binary search — seems promising. What's the problem?

A better "divide"

Consider this array: $A = [60, 43, 61, 87, 89, 87, 77, 11, 49, 45]$

- **Difficult**: Finding the element at a given position.
  For example, what is the 5th-smallest element in A?
- **Easier**: Finding the *position* of a given element.
  For example, what is the position of $x = 77$ in the sorted order?

Idea: Pick an element (the **pivot**), and sort around it.

Partition Algorithm

Input: Array A of size n. **Pivot** is in `A[0]`.
Output: Index p such that $A[p]$ holds the pivot, and
$A[a] \leq A[p] < A[b]$ for all $0 \leq a < p < b < n$.

```
def partition(A):
    n = len(A)
    i, j = 1, n-1
    while i <= j:
        if A[i] <= A[0]:
            i = i + 1
        elif A[j] > A[0]:
            j = j - 1
        else:
            swap(A, i, j)
    swap(A, 0, j)
    return j
```

Analysis of `partition`

- **Loop Invariant**: Everything before $A[i]$ is $\leq$ the pivot; everything after $A[j]$ is greater than the pivot.

- **Running time**: Consider the value of $j - i$.

Choosing a Pivot

The choice of pivot is really important!

- Want the partitions to be close to the same size.
- What would be the very best choice?

Initial "dumb" idea: Just pick the first element:

Input: Array A of length n
Output: Index of the pivot element we want

```
def choosePivot1(A):
    return 0
```


The Algorithm

Input: Array A of length n, and integer k
Output: Element at position k in the sorted array

```
def quickSelect1(A, k):
    n = len(A)
    swap(A, 0, choosePivot1(A))
    p = partition(A)
    if p == k:
        return A[p]
    elif p < k:
        return quickSelect1(A[p+1 : n], k-p-1)
    elif p > k:
        return quickSelect1(A[0 : p], k)
```

QuickSelect: Initial Analysis

- Best case:

- Worst case:

Average-case analysis

Assume all $n!$ permutations are equally likely.
Average cost is sum of costs for all permutations, divided by $n!$.

Define $T(n,k)$ as average cost of `quickSelect1(A,k)`:

$$T(n,k) = n + \frac{1}{n}\left(\sum_{p=0}^{k-1} T(n-p-1, k-p-1) + \sum_{p=k+1}^{n-1} T(p,k)\right)$$

See the book for a precise analysis, or...

Average-Case of `quickSelect1`

First simplification: define $T(n) = \max_k T(n,k)$

The key to the cost is the **position of the pivot**.

There are n possibilities, but can be grouped into:

- **Good pivots**: The position p is between $n/4$ and $3n/4$.
  Size of recursive call:

- **Bad pivots**: Position p is less than $n/4$ or greater than $3n/4$
  Size of recursive call:

Each possibility occurs $\frac{1}{2}$ of the time.

Average-Case of `quickSelect1`

Based on the cost and the probability of each possibility, we have:

$$T(n) \leq n + \frac{1}{2}T\left(\frac{3n}{4}\right) + \frac{1}{2}T(n)$$

(Assumption: every permutation in each partition is also equally likely.)

Drawbacks of Average-Case Analysis

To get the average-case we had to make some BIG assumptions:

- Every permutation of the input is equally likely
- Every permutation of each half of the partition is still equally likely

The first assumption is actually false in most applications!

Randomized algorithms

Randomized algorithms use a source of **random numbers**
in addition to the given input.

AMAZINGLY, this makes some things faster!

Idea: Shift assumptions on the *input distribution*
to assumptions on the *random number distribution*.
(Why is this better?)

Specifically, assume the function `random(n)` returns an integer between 0 and n-1 with uniform probability.

Randomized quickSelect

We could shuffle the whole array into a randomized ordering, or:

1. Choose the pivot element randomly:

Randomized pivot choice

```
def choosePivot2(A):
    # This returns a random number from 0 up to n-1
    return randrange(0, len(A))
```

2. Incorporate this into the quickSelect algorithm:

Randomized selection

```
def quickSelect2(A, k):
    swap(A, 0, choosePivot2(A))
    # ...  the rest is the same as quickSelect1
```


Analysis of `quickSelect2`

The **expected cost** of a randomized algorithm is the probability of each possibility, times the cost given that possibility.

We will focus on the **expected worst-case running time**.

Two cases: good pivot or bad pivot. Each occurs half of the time...
The analysis is exactly the same as the average case!

Expected worst-case cost of `quickSelect2` is $\Theta(n)$.
Why is this better than average-case?

Do we need randomization?

Can we do selection in linear time **without randomization**?

Blum, Floyd, Pratt, Rivest, and Tarjan figured it out in 1973.

But it's going to get a little complicated...

Median of Medians

Idea: Develop a divide-and-conquer algorithm for choosing the pivot.

1. Split the input into m sub-arrays
2. Find the median of each sub-array
3. Look at just the m medians, and take the median of those
4. Use the median of medians as the pivot

This algorithm will be **mutually recursive** with the selection algorithm. Crazy!

Note: q is a **parameter**, not part of the input. We'll figure it out next.

```
def choosePivot3(A):
    n = len(A)
    m = n // q

    # base case
    if m <= 1:
        return n // 2

    # Find median of each size-q group
    medians = []
    for i in range(0, m):
        medians.append(
          quickSelect3(A[i*q : (i+1)*q], q//2))

    # Find median of medians
    quickSelect3(medians, m//2)
    return m//2
```


Worst case of choosePivot3(A)

Assume all array elements are distinct.

Question: How unbalanced can the pivoting be?

- At least $\lceil m/2 \rceil$ medians must be $\leq$ the chosen pivot.
- At least $\lceil q/2 \rceil$ elements are $\leq$ each median.
- So the pivot must be greater than or equal to at least
  $$\left\lceil \frac{m}{2} \right\rceil \cdot \left\lceil \frac{q}{2} \right\rceil$$
  elements in the array, in the worst case.
- By the same reasoning, as many elements must be $\geq$ the chosen pivot.

Worst-case example, $q = 3$

$$A = [13, 25, 18, 76, 39, 51, 53, 41, 96, 5, 19, 72, 20, 63, 11]$$

Aside: "At Least Linear"

Definition

A function $f(n)$ is **at least linear** if and only if $f(n)/n$ is non-decreasing (for sufficiently large n).

- Any function that is $\Theta(n^c(\log n)^d)$ with $c \geq 1$ is "at least linear".
- You can pretty much assume that any running time that is $\Omega(n)$ is "at least linear".
- **Important consequence**: If $T(n)$ is at least linear, then $T(m) + T(n) \leq T(m + n)$ for any positive-valued variables n and m.

Analysis of `quickSelect3`

Since `quickSelect3` and `choosePivot3` are **mutually recursive**, we have to analyze them together.

- Let $T(n)$ = worst-case cost of `quickSelect3(A,k)`
- Let $S(n)$ = worst-case cost of `selectPivot3(A)`
- $T(n) =$
- $S(n) =$
- Combining these, $T(n) =$

Choosing q

- What if q is big? Try $q = n/3$.
- What if q is small? Try $q = 3$.

Choosing q

What about $q = 5$?

QuickSort

QuickSelect is based on a sorting method developed by Hoare in 1960:

```
def quickSort1(A):
    n = len(A)
    if n > 1:
        swap(A, 0, choosePivot1(A))
        p = partition(A)
        A[0 : p] = quickSort1(A[0 : p])
        A[p+1 : n] = quickSort1(A[p+1 : n])
    return A
```

QuickSort vs QuickSelect

- Again, there will be three versions depending on how the pivots are chosen.
- Crucial difference: QuickSort makes **two** recursive calls
- Best-case analysis:
- Worst-case analysis:
- We could ensure the best case by using `quickSelect3` for the pivoting. In practice, this is **too slow**.

Average-case analysis of `quickSort1`

Of all $n!$ permutations, $(n-1)!$ have pivot $A[0]$ at a given position i.

Average cost over all permutations:

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i) + T(n-i-1)\big) + \Theta(n), \qquad n \geq 2$$

Do you want to solve this directly?

Instead, consider the **average depth** of the recursion.
Since the cost at each level is $\Theta(n)$, this is all we need.

Average depth of recursion for `quickSort1`

$D(n) =$ average recursion depth for size-n inputs.

$$H(n) = \begin{cases} 0, & n \leq 1 \\ 1 + \frac{1}{n}\sum_{i=0}^{n-1} \max\big(H(i), H(n-i-1)\big), & n \geq 2 \end{cases}$$

- We will get a **good pivot** ($n/4 \leq p \leq 3n/4$) with probability $\frac{1}{2}$
- The *larger* recursive call will determine the height (i.e., be the "max") with probability at least $\frac{1}{2}$.

Summary of QuickSort analysis

- `quickSort1`: Choose $A[0]$ as the pivot.
  - Worst-case: $\Theta(n^2)$
  - Average case: $\Theta(n \log n)$
- `quickSort2`: Choose the pivot randomly.
  - Worst-case: $\Theta(n^2)$
  - **Expected** case: $\Theta(n \log n)$
- `quickSort3`: Use the median of medians to choose pivots.
  - Worst-case: $\Theta(n \log n)$

Sorting so far

We have seen:

- Quadratic-time algorithms:
  BubbleSort, SelectionSort, InsertionSort
- $n \log n$-time algorithms:
  HeapSort, MergeSort, QuickSort

$O(n \log n)$ is **asymptotically optimal** in the comparison model.

So how could we do better?

BucketSort

BucketSort is a general approach, not a specific algorithm:

1. Split the range of outputs into k groups or **buckets**
2. Go through the array, put each element into its bucket
3. Sort the elements in each bucket (perhaps recursively)
4. Dump sorted buckets out, in order

Notice: No comparisons!

countingSort(A,k)

Precondition $0 \leq A[i] < k$ for all indices i

```
def countingSort(A, k):
    C = [0] * k # size-k array filled with 0's
    for x in A:
        C[x] = C[x] + 1
    # Now C has the counts.
    # P will hold the positions.
    P = [0]
    for i in range(1, k):
        P.append(P[i-1] + C[i-1])
    # Now copy everything into its proper position.
    for x in copy(A):
        A[P[x]] = x
        P[x] = P[x] + 1
    return A
```


Analysis of CountingSort

- Time:
- Space:

Stable Sorting

Definition

A sorting algorithm is **stable** if elements with the same key stay in the same order.

- Quadratic algorithms and MergeSort are easily made stable
- QuickSort will require extra space to do **stable partition**.
- CountingSort is stable.

`radixSort(A,d,B)`

Input: Integer array A of length n, and integers d and B such that every $A[i]$ has d digits $A[i] = x_{d-1}x_{d-2}\cdots x_0$, to the base B.
Output: A gets sorted.

```
def radixSort(A, d, B):
    for i in range(0, d):
        countingSort(A, B) # based on the i'th digits
    return A
```

Works because CountingSort is stable!

Analysis:

Summary of Sorting Algorithms

Every algorithm has its place and purpose!

Algorithm	Analysis	In-place?	Stable?
SelectionSort	$\Theta(n^2)$ best and worst	yes	yes
InsertionSort	$\Theta(n)$ best, $\Theta(n^2)$ worst	yes	yes
HeapSort	$\Theta(n \log n)$ best and worst	yes	no
MergeSort	$\Theta(n \log n)$ best and worst	no	yes
QuickSort	$\Theta(n \log n)$ best, $\Theta(n^2)$ worst	yes	no
CountingSort	$\Theta(n+k)$ best and worst	no	yes
RadixSort	$\Theta(d(n+k))$ best and worst	yes	yes

Unit 5 Summary

- Selection problem
- Partition
- quickSelect and quickSort
- Average-case analysis
- Randomized algorithms and analysis
- Median of medians
- Non-comparison based sorting
- BucketSort, CountingSort, RadixSort
- Stable sorting