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”

- Finding the element at a given position is tough.

- But find the position of a given element is easy!

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

```plaintext
partition(A)
Input: Array A of size n. Pivot is in A[0].
Output: Index p such that A[p] holds the pivot, and
1  i := 1
2  j := n - 1
3  while i <= j do
4    if A[i] <= A[0] then
5      i := i + 1
6    else if A[j] > A[0] then
7      j := j - 1
8    else
9      swap (A[i], A[j])
10  end while
11  swap (A[0], A[j])
12  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:

```java
choosePivot1(A)
```

**Input**: Array $A$ of length $n$

**Output**: Index of the pivot element we want

```java
1 return 0
```

The Algorithm

```java
quickSelect1(A, k)
```

**Input**: Array $A$ of length $n$, and integer $k$

**Output**: Element at position $k$ in the sorted array

```java
1 swap (A[0], A[choosePivot1(A)])
2 p := partition(A)
3 if p = k then
4    return A[p]
5 else if p < k then
6    return quickSelect1(A[p+1..n-1], k-p-1)
7 else if p > k then
8    return quickSelect1(A[0..p-1], 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 $\text{quickSelect1}(A, k)$:

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

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

Average-Case of $\text{quickSelect1}$

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:
   
   ```python
   choosePivot2(A)
   1    return random(n)
   ```

2. Incorporate this into the quickSelect algorithm:
   
   ```python
   quickSelect2(A)
   1    swap (A[0], A[choosePivot2(A)])
   2    ...
   ```

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.

- Split the input into $m$ sub-arrays
- Find the median of each sub-array
- Look at just the $m$ medians, and take the median of those
- 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.
- quickSelect3(A,k) finds the element at position $k$ in the sorted array and re-arranges $A$ so that $A[k]$ is that element.

choosePivot3(A)

1. $m := \text{floor}(n/q)$
2. for $i$ from 0 to $m-1$
3.     // Find median of next group, move to front
4.     quickSelect3(A[i*q..(i+1)*q-1], floor(q/2))
5.     swap(A[i], A[i*q + floor(q/2)])
6. end for
7. // Find the median of medians
8. quickSelect3(A[0..m-1], floor(m/2))
9. return floor(m/2)

Worst case of choosePivot3(A)

Assume all array elements are distinct.

Question: How unbalanced can the pivoting be?

- Chosen pivot must be greater than $\lfloor m/2 \rfloor$ medians.
- Each median must be greater than $\lfloor q/2 \rfloor$ elements.
- Since $m = \lfloor n/q \rfloor$, the pivot must be greater than (and less than) approximately

$$\left\lceil \frac{n}{2q} \right\rceil \cdot \left\lfloor \frac{q}{2} \right\rfloor$$

elements in the worst case.
Worst-case example, \( q = 3 \)

\[
\]

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)`

\[
\begin{align*}
T(n) &= \\
S(n) &= \\
\text{Combining these, } T(n) &=
\end{align*}
\]
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:

quickSort1(A)

- **Input:** Array $A$ of size $n$
- **Output:** The array is sorted in-place.

\[
\begin{align*}
1 & \text{ if } n > 1 \text{ then} \\
2 & \text{ swap } (A[0], A[\text{choosePivot1}(A)]) \\
3 & p := \text{partition}(A) \\
4 & \text{quickSort1}(A[0..p-1]) \\
5 & \text{quickSort1}(A[p+1..n-1]) \\
6 & \text{end if}
\end{align*}
\]
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:

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

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

$$ D(n) = \text{average recursion depth for size-}n \text{ inputs.} $$

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

- 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 $\frac{1}{2}$. 

M. Agrawal and S. Das (2012)
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:

- Split the range of outputs into $k$ groups or buckets
- Go through the array, put each element into its bucket
- Sort the elements in each bucket (perhaps recursively)
- Dump sorted buckets out, in order

Notice: No comparisons!
countingSort(A,k)

**Input:** Integer array \( A \) of length \( n \), and integer \( k \) such that every \( A[i] \) satisfies \( 0 \leq A[i] < k \).

**Output:** \( A \) gets sorted.

1. \( C := \text{new array of size } k \)
2. for \( i \) from 0 to \( k \) do
3. \( C[i] := 0 \)
4. for \( i \) from 0 to \( n-1 \) do
5. \( C[A[i]] := C[A[i]] + 1 \)
6. for \( i \) from 1 to \( k-1 \) do
7. \( C[i] := C[i] + C[i-1] \)
8. \( B := \text{copy} (A) \)
9. for \( i \) from \( n-1 \) down to 0 do
10. \( C[B[i]] := C[B[i]] - 1 \)
11. \( A[C[B[i]]] := B[i] \)
12. end for

---

**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 integer d and k 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.

1 for i from 0 to d - 1 do
2 \hspace{1em} // Sort by the x_i's
3 countingSort(A,B) by every x_i

Works because CountingSort is stable!

Analysis:

Summary of Sorting Algorithms

Every algorithm has its place and purpose!

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Analysis</th>
<th>In-place?</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectionSort</td>
<td>Θ(n^2) best and worst</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>InsertionSort</td>
<td>Θ(n) best, Θ(n^2) worst</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>HeapSort</td>
<td>Θ(n \log n) best and worst</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>MergeSort</td>
<td>Θ(n \log n) best and worst</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>QuickSort</td>
<td>Θ(n \log n) best, Θ(n^2) worst</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>CountingSort</td>
<td>Θ(n + k) best and worst</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>RadixSort</td>
<td>Θ(d(n + k)) best and worst</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

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