Selection vs. Sorting

The selection problem is: Given an array $A$ of $n$ numbers, and $0 \leq k < n$, find the element in position $k$ of the sorted array.

**Observation:** the $k$th largest element is the element at position $n - k$.

Best heap-based algorithm had time cost $\Theta(n + k \log n)$.
For median selection, $k = \lfloor \frac{n}{2} \rfloor$, giving cost $\Theta(n \log n)$.
This is the same cost as our best sorting algorithms.

**Question:** Can we do selection in linear time?
The quick-select algorithm answers this question in the affirmative.

**Observation:** Finding the element at a given position is tough, but finding the position of a given element is simple.

Crucial Subroutines

quick-select and the related algorithm quick-sort rely on two subroutines:

- **choose-pivot($A$):** Choose an index $i$ such that $A[i]$ will make a good pivot (hopefully near the middle of the order).
- **partition($A, p$):** Using pivot $A[p]$, rearrange $A$ so that all items $\leq$ the pivot come first, followed by the pivot, followed by all items greater than the pivot.
Selecting a pivot

Ideally, we would select a median as the pivot. But this is the problem we’re trying to solve!

First idea: Always select first element in array

```python
choose-pivot1(A)
1. return 0
```

We will consider more sophisticated ideas later on.

---

Partition Algorithm

```python
partition(A, p)
A: array of size n, p: integer s.t. 0 ≤ p < n
1. swap(A[0], A[p])
2. i ← 1, j ← n − 1
3. loop
4. while i < n and A[i] ≤ A[0] do
5. i ← i + 1
7. j ← j − 1
8. if j < i then break
9. else swap(A[i], A[j])
10. end loop
11. swap(A[0], A[j])
12. return j
```

Idea: Keep swapping the outer-most wrongly-positioned pairs.

---

QuickSelect Algorithm

```python
quick-select1(A, k)
A: array of size n, k: integer s.t. 0 ≤ k < n
1. p ← choose-pivot1(A)
2. i ← partition(A, p)
3. if i = k then
4. return A[i]
5. else if i > k then
6. return quick-select1(A[0, 1, . . . , i − 1], k)
7. else if i < k then
8. return quick-select1(A[i + 1, i + 2, . . . , n − 1], k − i − 1)
```
Analysis of quick-select1

**Worst-case analysis:** Recursive call could always have size \( n - 1 \).

Recurrence given by \( T(n) = \begin{cases} T(n-1) + cn, & n \geq 2 \\ d, & n = 1 \end{cases} \)

Solution: \( T(n) = cn + c(n-1) + c(n-2) + \cdots + c \cdot 2 + d \in \Theta(n^2) \)

**Best-case analysis:** First chosen pivot could be the \( k \)th element
No recursive calls; total cost is \( \Theta(n) \).

Average-case analysis of quick-select1

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 for selecting \( k \)th item from size-\( n \) array:

\[
T(n, k) = cn + \frac{1}{n} \sum_{i=0}^{k-1} T(n-i-1, k-i-1) + \sum_{i=k+1}^{n-1} T(i, k)
\]

We could analyze this recurrence directly, or be a little lazier and still get the same asymptotic result.

For simplicity, define \( T(n) = \max_{0 \leq k < n} T(n, k) \).

Average-case analysis of quick-select1

The cost is determined by \( i \), the position of the pivot \( A[0] \).
For more than half of the \( n! \) permutations, \( \frac{3}{4} < i < \frac{3n}{4} \).

In this case, the recursive call will have length at most \( \lceil \frac{3n}{4} \rceil \), for any \( k \).

The average cost is then given by:

\[
T(n) \leq \begin{cases} cn + \frac{1}{n} \left( T(n) + T(\lceil 3n/4 \rceil) \right), & n \geq 2 \\ d, & n = 1 \end{cases}
\]

Rearranging gives:

\[
T(n) \leq 2cn + T(\lceil 3n/4 \rceil) \leq 2cn + 2c(3n/4) + 2c(9n/16) + \cdots + d
\]

\[
\leq d + 2cn \sum_{i=0}^{\infty} \left( \frac{3}{4} \right)^i \in O(n)
\]

Since \( T(n) \) must be \( \Omega(n) \) (why?), \( T(n) \in \Theta(n) \).
Randomized algorithms

A randomized algorithm is one which relies on some random numbers in addition to the input. The cost will depend on the input and the random numbers used.

Generating random numbers: Computers can’t generate randomness. Instead, some external source is used (e.g. clock, mouse, gamma rays, . . . )
This is expensive, so we use a pseudo-random number generator (PRNG), a deterministic program that uses a true-random initial value or seed. This is much faster and often indistinguishable from truly random.

Goal: To shift the probability distribution from what we can’t control (the input), to what we can control (the random numbers). There should be no more bad instances, just unlucky numbers.

Expected running time

Define $T(I, R)$ as the running time of the randomized algorithm for a particular input $I$ and the sequence of random numbers $R$.
The expected running time $T^{(exp)}(I)$ of a randomized algorithm for a particular input $I$ is the “expected” value for $T(I, R)$:
\[
T^{(exp)}(I) = E[T(I, R)] = \sum_R T(I, R) \cdot Pr[R]
\]
The worst-case expected running time is then
\[
T^{(exp)}(n) = \max_{\text{size}(I)=n} T^{(exp)}(I).
\]
For many randomized algorithms, worst-, best-, and average-case expected times are the same (why?).

Randomized QuickSelect

$\text{random}(n)$ returns an integer uniformly from $\{0, 1, \ldots, n-1\}$.

**First idea:** Randomly permute the input first using shuffle:

\begin{verbatim}
shuffle(A)
A: array of size n
1. for i ← 0 to n − 2 do
2. swap(A[i], A[i + random(n − i)])
\end{verbatim}

Expected cost becomes the same as the average cost, which is $\Theta(n)$. 
Randomized QuickSelect

**Second idea**: Change the pivot selection.

```plaintext
choose-pivot2(A)
1. \text{return } \text{random}(n)
```

```plaintext
quick-select2(A, k)
1. \text{p \leftarrow choose-pivot2}(A)
2. \ldots
```

With probability at least \( \frac{1}{2} \), the random pivot has position \( \frac{2}{3} \leq i < \frac{3n}{4} \), so the analysis is just like that for the average-case. The expected cost is again \( \Theta(n) \).

*This is generally the fastest quick-select implementation.*

---

Worst-case linear time

Blum, Floyd, Pratt, Rivest, and Tarjan invented the “medians-of-five” algorithm in 1973 for pivot selection:

```plaintext
choose-pivot3(A)
A: array of size \( n \)
1. \text{m \leftarrow } \lfloor \frac{n}{5} \rfloor - 1
2. \text{for } i \leftarrow 0 \text{ to } m \text{ do}
3. \quad j \leftarrow \text{index of median of } A[5i, \ldots, 5i+4]
4. \quad \text{swap}(A[i], A[j])
5. \text{return } \text{quick-select3}(A[0, \ldots, m], \lfloor m/2 \rfloor)
```

```plaintext
quick-select3(A, k)
1. \text{p \leftarrow choose-pivot3}(A)
2. \ldots
```

This *mutually recursive* algorithm can be shown to be \( \Theta(n) \) in the worst case, but it’s a little beyond the scope of this course.

---

QuickSort

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

```plaintext
quick-sort1(A)
A: array of size \( n \)
1. \text{if } n \leq 1 \text{ then return}
2. \text{p \leftarrow choose-pivot1}(A)
3. \text{i \leftarrow } \text{partition}(A, p)
4. \text{quick-sort1}(A[0, \ldots, i-1])
5. \text{quick-sort1}(A[i+1, \ldots, \text{size}(A)-1])
```

**Worst case**: \( T^{(\text{worst})}(n) = T^{(\text{worst})}(n-1) + \Theta(n) \)

Same as quick-select1; \( T^{(\text{worst})}(n) \in \Theta(n^2) \)

**Best case**: \( T^{(\text{best})}(n) = T^{(\text{best})}(n-1) + T^{(\text{best})}(i-1) + \Theta(n) \)

Similar to merge-sort; \( T^{(\text{best})}(n) \in \Theta(n \log n) \)
Average-case analysis of quick-sort1

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

Average cost over all permutations is given by:

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

It is possible to solve this recursion directly.

Instead, notice that the cost at each level of the recursion tree is $O(n)$.

So let's consider the height (or "depth") of the recursion, on average.

Average depth of recursion for quick-sort1

Define $H(n)$ as the average recursion depth for size-$n$ inputs. So

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

Let $i$ be the position of the pivot $A[0]$.

Again, $\frac{n}{3} \leq i < \frac{2n}{3}$ for more than half of all permutations.

Then larger recursive call has length at most $\lfloor \frac{3n}{4} \rfloor$.

This will determine the recursion depth for at least half of all inputs.

Therefore $H(n) \leq 1 + \frac{1}{2} (H(n) + H(\lfloor \frac{3n}{4} \rfloor))$ for $n \geq 2$,

which simplifies to $H(n) \leq 2 + H(\lfloor \frac{3n}{4} \rfloor)$.

So $H(n) \in O(\log n)$.

Average cost is $O(nH(n)) \in O(n\log n)$.

Since best-case is $\Theta(n \log n)$, average must be $\Theta(n \log n)$.

More notes on QuickSort

- We can randomize by using choose-pivot2, giving
  $\Theta(n \log n)$ expected time for quick-sort2.

- We can use choose-pivot3 (along with quick-select3) to get quick-sort3 with $\Theta(n \log n)$ worst-case time.

- We can use tail recursion to save space on one of the recursive calls. By making sure the other one is always smaller, the auxiliary space is $\Theta(\log n)$ in the worst case, even for quick-sort1.

- QuickSort is often the most efficient algorithm in practice.
Lower bounds for sorting

We have seen many sorting algorithms:

<table>
<thead>
<tr>
<th>Sort</th>
<th>Running time</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>Θ(n^2)</td>
<td>worst-case</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>Θ(n^2)</td>
<td>worst-case</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>Θ(n log n)</td>
<td>worst-case</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>Θ(n log n)</td>
<td>worst-case</td>
</tr>
<tr>
<td>quick-sort1</td>
<td>Θ(n log n)</td>
<td>average-case</td>
</tr>
<tr>
<td>quick-sort2</td>
<td>Θ(n log n)</td>
<td>expected</td>
</tr>
<tr>
<td>quick-sort3</td>
<td>Θ(n log n)</td>
<td>worst-case</td>
</tr>
</tbody>
</table>

**Question:** Can one do better than Θ(n log n)?

**Answer:** Yes and no! It depends on what we allow.
- No: Comparison-based sorting lower bound is Ω(n log n).
- Yes: Non-comparison-based sorting can achieve O(n).

The Comparison Model

In the *comparison model* data can only be accessed in two ways:
- comparing two elements
- moving elements around (e.g. copying, swapping)

This makes very few assumptions on the kind of things we are sorting. We count the number of above operations.

All sorting algorithms seen so far are in the comparison model.

Lower bound for sorting in the comparison model

**Theorem.** Any correct comparison-based sorting algorithm requires at least Ω(n log n) comparison operations.

**Proof.**
- A correct algorithm takes different actions (moves, swaps, etc.) for each of the n! possible permutations.
- The choice of actions is determined only by comparisons.
- The algorithm can be viewed as a decision tree.
  - Each internal node is a comparison, each leaf is a set of actions.
  - Each permutation must correspond to a leaf.
  - The worst-case number of comparisons is the longest path to a leaf.
  - Since the tree has at least n! leaves, the height is at least lg n!.
  - Therefore worst-case number of comparisons is Ω(n log n).
Non-comparison-based sorting

- Assume keys are numbers in base $R$ ($R$: radix)
  - $R = 2, 10, 128, 256$ are the most common.
- Assume all keys have the same number $m$ of digits.
  - Can achieve after padding with leading 0s.
- Can sort based on individual digits.

**MSD-Radix-sort**

$A$: array of size $n$, contains $m$-digit radix-$R$ numbers
$l, r, d$: integers, $0 \leq l, r \leq n - 1$, $1 \leq d \leq m$

1. if $l < r$
   1. partition $A[l..r]$ into bins according to $d$th digit
   2. if $d < m$
   3. for $i \leftarrow 0$ to $R - 1$ do
      4. let $l_i$ and $r_i$ be boundaries of $i$th bin
      5. MSD-Radix-sort($A, l_i, r_i, d + 1$)

To partition: Sort by counting.

Count Sort

$key-indexed-count-sort(A)$

$A$: array of size $n$ containing numbers in $\{0, \ldots, R - 1\}$

1. $C \leftarrow$ array of size $R$, filled with zeros
2. for $i \leftarrow 0$ to $n - 1$ do
3. increment $C[A[i]]$
4. $l \leftarrow$ array of size $R$, $l[0] = 0$
5. for $i \leftarrow 1$ to $R - 1$ do
6. $l[i] \leftarrow l[i - 1] + C[i - 1]$
7. $B \leftarrow$ copy($A$)
8. for $i \leftarrow 0$ to $n - 1$ do
10. increment $l[d]$

- **Time:** $\Theta(n + R)$.
- Optional: re-use $C$ rather than introducing $l$.

LSD-Radix-Sort

- Drawback of MSD-Radix-Sort: many recursions

$LSD-radix-sort(A)$

$A$: array of size $n$, contains $m$-digit radix-$R$ numbers

1. for $d \leftarrow m$ down to 1 do
2. Sort $A$ by $d$th digit

- Sort-routine must be stable: equal items stay in original order.
  - CountSort, InsertionSort, MergeSort are (usually) stable.
  - HeapSort, QuickSort not stable as implemented.
- Loop-invariant: $A$ is sorted w.r.t. digits $d, \ldots, m$ of each entry.
- **Time cost:** $\Theta(m(n + R))$ if we use CountSort
- **Auxiliary space:** $\Theta(n + R)$
Summary of sorting

- Randomized algorithms can eliminate "bad cases"
- Best-case, worst-case, average-case, expected-case can all differ
- Sorting is an important and very well-studied problem
- Can be done in $\Theta(n \log n)$ time; faster is not possible for general input
- HeapSort is the only fast algorithm we have seen with $O(1)$ auxiliary space.
- QuickSort is often the fastest in practice
- MergeSort is also $\Theta(n \log n)$, selection & insertion sorts are $\Theta(n^2)$.
- CountSort, RadixSort can achieve $o(n \log n)$ if the input is special