Quick-sort review

- Assume the pivot is selected as an arbitrary element (e.g., the first item in the array)
- The worst-case running time is?
Quick-sort review

• Assume the pivot is selected as an arbitrary element (e.g., the first item in the array)

• The worst-case running time is?
  • It is $\Theta(n^2)$ when the pivots are always the smallest/largest items.
Quick-sort review

- Assume the pivot is selected as an arbitrary element (e.g., the first item in the array)
- The worst-case running time is?
  - It is $\Theta(n^2)$ when the pivots are always the smallest/largest items.
- The best-case running time is?
Quick-sort review

- Assume the pivot is selected as an arbitrary element (e.g., the first item in the array)

- The worst-case running time is?
  - It is $\Theta(n^2)$ when the pivots are always the smallest/largest items.

- The best-case running time is?
  - It is $\Theta(n \log n)$, when there are a linear number (e.g., roughly half) of items on each side of pivots.
Quick-sort review

- Assume the pivot is selected as an arbitrary element (e.g., the first item in the array)

- The worst-case running time is?
  - It is $\Theta(n^2)$ when the pivots are always the smallest/largest items.

- The best-case running time is?
  - It is $\Theta(n \log n)$, when there are a linear number (e.g., roughly half) of items on each side of pivots.

- The average-case running time is?
  - When the input is shuffled, the running time is $\Theta(n \log n)$. 
A sorting algorithm is **comparison-based** if it can sort any array of **objects** by just pairwise comparison of them.

- E.g., you want to sort a bag of potatoes using a balance scale.

- It is known that any comparison-based sorting algorithm runs in $\Omega(n \log n)$ in the worst-case.
A sorting algorithm is **comparison-based** if it can sort any array of objects by just pairwise comparison of them.

- E.g., you want to sort a bag of potatoes using a balance scale.

It is known that any comparison-based sorting algorithm runs in $\Omega(n \log n)$ in the worst-case.

Can we improve the worst-case running time $\Theta(n^2)$ of Quick-sort to $\Theta(n \log n)$?

- This relates to the **selection problem**.
Selection & order statistics

- The $i$'th order statistic of a set of comparable elements is the $i$'th smallest value in the set.
  - The $\lceil n/2 \rceil$'th order statistic among $n$ items is called **median**.
  - The $\lceil n/4 \rceil$'th order statistic among $n$ items is called **quartile**.

- How can we find the 0'th or ($n - 1$)'th order statistic in $\Theta(n)$.
The $i$’th order statistic of a set of comparable elements is the $i$’th smallest value in the set.

- The $\lceil n/2 \rceil$’th order statistic among $n$ items is called **median**.
- The $\lceil n/4 \rceil$’th order statistic among $n$ items is called **quartile**.

How can we find the $0$’th or $(n - 1)$’th order statistic in $\Theta(n)$.

- Finding min/max $\rightarrow$ a linear scan is sufficient!
Selection & order statistics

- The $i$’th order statistic of a set of comparable elements is the $i$’th smallest value in the set.
  - The $\lceil n/2 \rceil$’th order statistic among $n$ items is called **median**.
  - The $\lceil n/4 \rceil$’th order statistic among $n$ items is called **quartile**.
- How can we find the 0’th or $(n - 1)$’th order statistic in $\Theta(n)$.
  - Finding min/max $\rightarrow$ a linear scan is sufficient!
- **Selection problem:** find the $i$’th order statistics:
  - The input is a set of $n$ comparable objects (e.g., integers) and an integer $i$
  - The output is the element at index $i$ of the sorted array ($i + 1$’th smallest item)
Selection algorithms

- Attempt I: sort $A$ and return the element at index $i$ in the sorted array.
  - E.g., use Merge-sort; sorting takes $\Theta(n \log n)$ and accessing the element in sorted array takes $\Theta(1)$. 

- Can we do better?
  - Attempt II: apply `heapify` on $A$ and `extract-min` $i + 1$ times (we assume indices start at 0).
    - `Heapify` takes $\Theta(n)$ and each `extract-min` operation takes $\Theta(\log n)$.
    - Select takes $\Theta(n + i \log n)$, which is $\Theta(n \log n)$ when $i \in \Theta(n)$.

The running time is $\Theta(n)$ for $i \in O(n / \log n)$.

What is the minimum time required for selection?

We need to read the whole input, i.e., the running time of any algorithm is $\Omega(n)$. Can we select in $\Theta(n)$?
Selection algorithms

- Attempt I: sort $A$ and return the element at index $i$ in the sorted array.
  - E.g., use Merge-sort; sorting takes $\Theta(n \log n)$ and accessing the element in sorted array takes $\Theta(1)$.
  - Can we do better?
Selection algorithms

- Attempt I: sort $A$ and return the element at index $i$ in the sorted array.
  - E.g., use Merge-sort; sorting takes $\Theta(n \log n)$ and accessing the element in sorted array takes $\Theta(1)$.
  - Can we do better?

- Attempt II: apply heapify on $A$ and extract-min $i + 1$ times (we assume indices start at 0).
  - Heapify takes $\Theta(n)$ and each extract-min operation takes $\Theta(\log n)$
  - Select takes $\Theta(n + i \log n)$, which is $\Theta(n \log n)$ when $i \in \Theta(n)$.
  - The running time is $\Theta(n)$ for $i \in O(n/\log n)$. 

What is the minimum time required for selection?
We need to read the whole input, i.e., the running time of any algorithm is $\Omega(n)$. Can we select in $\Theta(n)$?
Selection algorithms

- **Attempt I:** sort $A$ and return the element at index $i$ in the sorted array.
  - E.g., use Merge-sort; sorting takes $\Theta(n \log n)$ and accessing the element in sorted array takes $\Theta(1)$.
  - Can we do better?

- **Attempt II:** apply heapify on $A$ and extract-min $i + 1$ times (we assume indices start at 0).
  - Heapify takes $\Theta(n)$ and each extract-min operation takes $\Theta(\log n)$
  - Select takes $\Theta(n + i \log n)$, which is $\Theta(n \log n)$ when $i \in \Theta(n)$.
  - The running time is $\Theta(n)$ for $i \in O(n/\log n)$.

- What is the minimum time required for selection?
Selection algorithms

- Attempt I: sort $A$ and return the element at index $i$ in the sorted array.
  - E.g., use Merge-sort; sorting takes $\Theta(n \log n)$ and accessing the element in sorted array takes $\Theta(1)$.
  - Can we do better?

- Attempt II: apply `heapify` on $A$ and `extract-min $i + 1` times (we assume indices start at 0).
  - Heapify takes $\Theta(n)$ and each `extract-min` operation takes $\Theta(\log n)$
  - Select takes $\Theta(n + i \log n)$, which is $\Theta(n \log n)$ when $i \in \Theta(n)$.
  - The running time is $\Theta(n)$ for $i \in O(n/\log n)$.

- What is the minimum time required for selection?
  - We need to read the whole input, i.e., the running time of any algorithm is $\Omega(n)$. 
Selection algorithms

- Attempt I: sort $A$ and return the element at index $i$ in the sorted array.
  - E.g., use Merge-sort; sorting takes $\Theta(n \log n)$ and accessing the element in sorted array takes $\Theta(1)$.
  - Can we do better?

- Attempt II: apply heapify on $A$ and extract-min $i + 1$ times (we assume indices start at 0).
  - Heapify takes $\Theta(n)$ and each extract-min operation takes $\Theta(\log n)$
  - Select takes $\Theta(n + i \log n)$, which is $\Theta(n \log n)$ when $i \in \Theta(n)$.
  - The running time is $\Theta(n)$ for $i \in O(n / \log n)$.

- What is the minimum time required for selection?
  - We need to read the whole input, i.e., the running time of any algorithm is $\Omega(n)$.
  - Can we select in $\Theta(n)$?
Selection algorithms

- Quick-select: similar to Quick-sort, but for selection
- Select a pivot, partition around it, and recurs on the one side that contains the $i$’th element
QuickSelect Algorithm

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

• Here the pivot is selected arbitrarily (e.g., the first item in the array)
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:
$T(n) = cn + c(n - 1) + c(n - 2) + \cdots + c \cdot 2 + d \in \Theta(n^2)$
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-select

Assume all $n!$ permutations are equally likely.

Define $T(n, i)$ as average cost for selecting $i$th item from size-$n$ array:

$$T(n, i) = cn + \frac{1}{n} \left( \sum_{j=0}^{i-1} T(n-j-1, i-j-1) + \sum_{j=i+1}^{n-1} T(j, i) \right)$$

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-select

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

In this case, the recursive call will have length at most $\left\lfloor \frac{3n}{4} \right\rfloor$, for any $k$.

The average cost is then given by:

$$T(n) \leq \begin{cases} cn + \frac{1}{2} \left( T(n) + T(\left\lfloor \frac{3n}{4} \right\rfloor) \right), & n \geq 2 \\ d, & n = 1 \end{cases}$$
Average-case analysis of quick-select

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

In this case, the recursive call will have length at most $\left\lfloor \frac{3n}{4} \right\rfloor$, for any $k$.

The average cost is then given by:

$$T(n) \leq \begin{cases} \frac{1}{2} \left( T(n) + T(\left\lfloor \frac{3n}{4} \right\rfloor) \right), & n \geq 2 \\ d, & n = 1 \end{cases}$$

Rearranging gives:

$$T(n) \leq 2cn + T(\left\lfloor \frac{3n}{4} \right\rfloor) \leq 2cn + 2c(3n/4) + 2c(9n/16) + \cdots + d$$

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

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

COMP 3170 - Analysis of Algorithms & Data Structures
Linear-time selection

- Although Quick-select runs in $O(n)$ on average, in the worst-case it is still super-linear.
- Is there any selection algorithm that runs in $O(n)$ in the worst-case?

  - The answer is Yes; **Median of medians** algorithms!
  - It is a twist to Quick-select in which the pivot is selected a bit smarter!
Median of five algorithm

- The input is an array $A$ of $n$ objects (assume $n$ is divisible by 5).
- Divide $A$ into $n/5$ blocks of size 5.
- Recursively find the median of the medians; denote it by $x$.
- Partition the whole array using $x$ as the pivot.
- Recurs on the corresponding subarray as in Quick-select.