COMP 3170 - Analysis of Algorithms & Data Structures

Shahin Kamali

Lecture 6 - Jan. 18, 2019
CLRS 7.1, 7-4, 9.1, 9.3
University of Manitoba
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 $O(n \log n)$. 
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.

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.
Comparison-based algorithms

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 \([n/2]\)’th order statistic among \(n\) items is called median.
  - The \([n/4]\)’th order statistic among \(n\) items is called quartile.
- How can we find the 0’th or \((n - 1)\)’th order statistic in \(\Theta(n)\).
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 & 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 → 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 $\Theta(1)$.
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 \( i \in O(n/\log n) \).
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 Review

**quick-select1**\((A, i)\)
\(A: \text{ array of size } n, \quad i: \text{ integer s.t. } 0 \leq i < n\)
1. \(p \leftarrow \text{choose-pivot1}(A)\)
2. \(j \leftarrow \text{partition}(A, p)\)
3. \(\text{if } j = i \text{ then}\)
   4. \(\text{return } A[j]\)
5. \(\text{else if } j > i \text{ then}\)
   6. \(\text{return } \text{quick-select1}(A[0, 1, \ldots, j - 1], i)\)
7. \(\text{else if } j < i \text{ then}\)
   8. \(\text{return } \text{quick-select1}(A[j + 1, j + 2, \ldots, n - 1], i - j - 1)\)

- If pivot is at position \(j\), the cost of recursive call parameters will be:
  - None if \(j = i\).
  - \((j, i)\) if \(j > i\) (recursing on the left subarray).
  - \((n - j - 1, i - j - 1)\) if \(j < i\) (recursing on the right subarray).
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:

The cost for recursive calls (RC) is

\[
RC = \begin{cases} 
0 & j = i \\
T(j, i) & j > i \\
T(n - j - 1, i - j - 1) & j < i
\end{cases}
\]

For simplicity, define \( T(n) = \max_{0 \leq k < n} T(n, k) \).
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: The cost for recursive calls (RC) is

\[
RC = \begin{cases} 
0 & j = i \\
T(j, i) & j > i \\
T(n - j - 1, i - j - 1) & j < i 
\end{cases}
\]

Shuffled input → it is equally likely for the pivot to be at any position:

\[
T(n, i) = \left( cn \right)_{\text{partition}} + \frac{1}{n} \left( (\text{RC if } j=0) + (\text{RC if } j=1) + \ldots + (\text{RC if } j=n-1) \right)
\]

\[
= \left( cn \right)_{\text{partition}} + \frac{1}{n} \left( \sum_{j=0}^{i-1} T(n - j - 1, i - j - 1) + \sum_{j=i+1}^{n-1} T(j, i) \right)
\]

For simplicity, define \( T(n) = \max_{0 \leq k < n} T(n, k) \).
Average-case analysis of quick-select

\[ T(n) \leq \underbrace{cn}_{\text{partition}} + \frac{1}{n} \left( \sum_{j=0}^{i-1} T(n - j - 1) + \sum_{j=i+1}^{n-1} T(j) \right) \]
Average-case analysis of quick-select

\[ T(n) \leq cn \underbrace{\text{partition}}_{\text{partition}} + \frac{1}{n} \left( \sum_{j=0}^{i-1} T(n-j-1) + \sum_{j=i+1}^{n-1} T(j) \right) \]

- We say that a pivot is **good** if the arrays on both sides have size at least \( n/4 \)
  - This happens when pivot index \( j \) is in \([n/4, 3n/4)\).
  - Half of possible pivots are good and the rest are bad.
Average-case analysis of quick-select

\[ T(n) \leq \begin{cases} 
    cn + \frac{1}{n} \left( \sum_{j=0}^{i-1} T(n-j-1) + \sum_{j=i+1}^{n-1} T(j) \right) & \text{partition} \\
    \frac{1}{n} \left( T(n) \right) \quad & n = 1 
\end{cases} \]

- We say that a pivot is **good** if the arrays on both sides have size at least \( n/4 \)
  - This happens when pivot index \( j \) is in \([n/4, 3n/4)\).
  - Half of possible pivots are good and the rest are bad.

- The recursive cost for a good pivot is at most \( T(3n/4) \).
- The recursive cost for a bad pivot is at most \( T(n) \).

The average cost is then given by:

\[ T(n) \leq \begin{cases} 
    cn + \frac{1}{2} \left( T(n) + T(\lfloor 3n/4 \rfloor) \right), & n \geq 2 \\
    d & n = 1 
\end{cases} \]
The average cost is then given by:

\[ T(n) \leq \begin{cases} 
  cn + \frac{1}{2} \left( T(n) + T(\lfloor 3n/4 \rfloor) \right), & n \geq 2 \\
  d, & n = 1 
\end{cases} \]
The average cost is then given by:

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

Rearranging gives:

\[
T(n) \leq 2cn + T(\lfloor 3n/4 \rfloor) \leq 2cn + 2c(3n/4) + 2c(9n/16) + \cdots + d \\
\leq d + 2cn \sum_{i=0}^{\infty} \left( \frac{3}{4} \right)^i \in O(n)
\]

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

- A variant of Quick-select in which the pivot is selected more carefully.
- 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 $x$.
  - $x$ will be the pivot for quick-select
- Partition the whole array using $x$ as the pivot
- Recurs on the corresponding subarray as in Quick-select
Median of five example

........ 2 54 44 4 25 ...............  
........ 5 5 32 18 39 ................  
........ 9 87 21 26 47 ................  
........ 19 9 13 16 56 ................  
........ 24 10 2 19 71 ...............
Median of five example

\[
\begin{array}{ccccc}
2 & 5 & 2 & 4 & 25 \\
5 & 9 & 13 & 16 & 39 \\
9 & 10 & 21 & 18 & 47 \\
19 & 54 & 32 & 19 & 56 \\
24 & 87 & 44 & 26 & 71 \\
\end{array}
\]

Median of each group
Median of five example

Find X, the median of medians
Median of five algorithm

- Pivot $x$ is median of medians $\rightarrow$ half of blocks have median $< x$.
  - This implies half of blocks include at least 3 elements $< x$.
  - So, there will be at least $n/5 \cdot 1/2 \cdot 3 = 3n/10$ elements smaller than $x$

- Similarly, there will be at least $3n/10$ elements larger than $x$.

- We assume distinct items; when pivot is equal to multiple items, you can update the partition algorithm so that the pivot is the ‘best’ among items with the same key.

- Hence, the size of recursive call is always in the range $(3n/10, 7n/10)$.
  - $x$ is always a ‘good’ pivot
Median of five algorithm

- Pivot $x$ is median of medians $\rightarrow$ half of blocks have median $< x$.
  - This implies half of blocks include at least 3 elements $< x$.
  - So, there will be at least $n/5 \cdot 1/2 \cdot 3 = 3n/10$ elements smaller than $x$

- Similarly, there will be at least $3n/10$ elements larger than $x$.

- We assume distinct items; when pivot is equal to multiple items, you can update the partition algorithm so that the pivot is the ‘best’ among items with the same key.

- Hence, the size of recursive call is always in the range $(3n/10, 7n/10)$.
  - $x$ is always a ‘good’ pivot

- In the worst case, the size of recursive call is always $7n/10$.

\[
T(n) \leq \begin{cases} \\
\frac{T(n/5)}{d} + \frac{cn}{partition\ around\ x} + T(7n/10), & n \geq 2 \\
T(n/5) + cn & n = 1 \\
\end{cases}
\]
Median of five algorithm

\[ T(n) \leq \begin{cases} 
T(n/5) + \frac{cn}{5} + T(7n/10), & n \geq 2 \\
T(n) & n = 1 
\end{cases} \]

- We guess that \( T(n) \in O(n) \) and use strong induction to prove it.
- We prove there is a value \( M \) s.t. \( T(n) \leq Mn \) for all \( n \geq 1 \).
- For the base we have \( T(1) = d \leq M \) as long as \( M \geq d \).
- For any value of \( n \) we can state:

\[
T(n) \leq T(n/5) + T(7n/10) + cn \quad \text{(definition)} \\
\leq M \cdot n/5 + M \cdot 7n/10 + cn \quad \text{(induction hypothesis)} \\
= (9M/10 + c)n \\
\leq M \cdot n \quad \text{as long as } M \geq 9M/10 + c, \text{i.e., } M \geq 10c
\]

so, we showed for \( M = \max\{10c, d\} \) we have \( T(n) \leq M \cdot n \) for \( n \geq 1 \). So, \( T(n) \in O(n) \).
Quick-sort revisit

Theorem

It is possible to select the i’th smallest item in a list of n numbers in time $\Theta(n)$

- Quick-sort in $O(n \log n)$ time:
  - Using select algorithm to choose the pivot as the median of n items in $O(n)$ time
  - Partition around pivot in $O(n)$ time (selecting pivot as $n/c$’th smallest item for constant $c$ gives the same result)
  - Sort the two sides of pivot recursively in time $2T(n/2)$.

  The cost will be $T(n) = 2T(n/2) + \Theta(n)$, which gives $T(n) = \Theta(n \log n)$ [case II of Master theorem]

Theorem

A smart selection of pivot, using linear-time select, results in quick-sort running in $\Theta(n \log n)$
QuickSelect Algorithm

\[
\text{quick-select1}(A, i)
\]

\(A\): array of size \(n\), \(i\): integer s.t. \(0 \leq i < n\)

1. \(p \leftarrow \text{choose-pivot1}(A)\)
2. \(j \leftarrow \text{partition}(A, p)\)
3. \textbf{if } j = i \textbf{ then}
4. \hspace{2em} \textbf{return } A[j]
5. \textbf{else if } j > i \textbf{ then}
6. \hspace{2em} \textbf{return } \text{quick-select1}(A[0, 1, \ldots, j - 1], i)\)
7. \textbf{else if } j < i \textbf{ then}
8. \hspace{2em} \textbf{return } \text{quick-select1}(A[j + 1, j + 2, \ldots, n - 1], i - j - 1)\)

- Here the pivot is selected arbitrarily (e.g., the first item in the array)
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:

\[
T(n) = \begin{cases} 
T(n - 1) + cn, & n \geq 2 \\
\phantom{T(n - 1) + c} d, & n = 1
\end{cases}
\]

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:

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

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

Rearranging gives:

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

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

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