Algorithms: Order Statistics

Amotz Bar-Noy

CUNY
Finding the Minimum Or the Maximum

**Input**

**Output**
- **Minimum**: A key $K$ from $A$ such that $K \leq A[i]$ for all $1 \leq i \leq n$.
- **Maximum**: A key $K$ from $A$ such that $K \geq A[i]$ for all $1 \leq i \leq n$.

**Method**
- By *comparisons* between any two keys from the array.

**Optimization goal**
- Minimize the number of key comparisons.

**Keys**
- Usually keys will be represented by numbers or integers.
Finding the Maximum

**Trivial algorithm**

- **Scan** the array to find the maximum key.
- **Maintain** a candidate key $K$ to be the maximum key.
- In each round, **compare** the candidate key with the next key from the array.
- **Update** the candidate if a larger key is found.
- **Terminate** after $n - 1$ rounds, after the candidate is compared with $A[n]$.
- **Return** the candidate key as the maximum key.
Finding the Maximum

Pseudocode

**Trivial-Find-Max**\((A[1], \ldots, A[n])\)

\[
\text{for } i = 2 \text{ to } n \text{ \{ } \text{if } K < A[i] \text{ \{ } \text{then } K := A[i] \text{ \}} \text{ \}} \text{ \}} \text{ \}	ext{ return } K
\]

Correctness

- By induction: \( K = \max \{A[1], \ldots, A[i + 1]\} \) after round \( i \).
- At the end: \( K = \max \{A[1], \ldots, A[n]\} \) after \( n - 1 \) rounds.

Complexity

- Exactly \( n - 1 \) comparisons.
Finding the Maximum

Adversary strategy

- **Key idea:** Any key in $A$ could potentially be the maximum.

Adversary data structure

- $\mathcal{H}$ - Set of candidates keys that could be the maximum.
- $\mathcal{R}$ - Set of keys that cannot be maximum.

Initially: $\mathcal{H} = \{A[1], \ldots, A[n]\}$ and $\mathcal{R} = \emptyset$.

At the end: $|\mathcal{H}| = 1$ and $|\mathcal{R}| = n - 1$.

Adversary answer rules

- $(R_1 : R_2) \Rightarrow$ Any consistent answer.
- $(H : R) \Rightarrow H > R$.
- $(H_1 : H_2) \Rightarrow H_1 < H_2$; transfer $H_1$ from $\mathcal{H}$ to $\mathcal{R}$.
Finding the Maximum

Theorem
- The adversary forces any algorithm that finds the maximum to make at least $n - 1$ comparisons.

Proof
- A **useful comparison** decreases the size of $B$.
- Only $(H_1 : H_2)$ is a **useful comparison**.
- Each **useful comparison** decreases the size of $H$ by 1.
- $n - 1$ **useful comparisons** are required to decrease the size of $H$ from $n$ to 1.

Optimality
- **Trivial-Find-Max** is an **optimal** algorithm to find the maximum.
Finding the Minimum Or the Maximum in Parallel

**Model**
- The search is done in **rounds**.
- Each round may contain several comparisons.
- **Mutual exclusion**: In each round, a key may participate in at most one comparison.

**Optimization goals**
- Minimize number of **rounds**.
- Minimize number of **comparisons**.
Finding the Minimum in Parallel

The Tournament Algorithm

- In parallel, compare $\lfloor n/2 \rfloor$ pairs of keys from the array $A$.
- Move the $\lfloor n/2 \rfloor$ smaller keys (and the extra un-compared key in case $n$ is odd) to the beginning of the array.
- Continue recursively with the first $\lceil n/2 \rceil$ keys in the array.
- Return $A[1]$ as the minimum key once the size of the array is 1.
Finding the Minimum in Parallel

Assumption

- $n = 2^k$ is a power of 2.

Pseudocode

```
Parallel-Find-Min(A[1], \ldots, A[n])
  if $n = 1$ then return $A[1]$
  for $i = 1$ to $n/2$
    if $A[i] > A[i + (n/2)]$ (* comparison *)
      then $A[i] \leftrightarrow A[i + (n/2)]$
  return Parallel-Find-Min(A[1], \ldots, A[n/2])
```
Finding the Minimum in Parallel

**Correctness**

- Initially the minimum key is one of $A[1], A[2], \ldots, A[n]$.
- After the first round, the minimum key is one of $A[1], A[2], \ldots, A[n/2]$.
- After the second round, the minimum key is one of $A[1], A[2], \ldots, A[n/4]$.
- By induction: after $r$ rounds the minimum key is one of $A[1], A[2], \ldots, A[n/2^r]$.
- After $r = k = \log_2(n)$ rounds, the minimum is $A[n/2^k] = A[1]$. 
Finding the Minimum in Parallel

Number of comparisons
- There are $\frac{n}{2^r}$ comparisons in the $r^{th}$ round.
- Total number of comparisons: $\frac{n}{2} + \frac{n}{4} + \cdots + 1 = n - 1$.
- The same as in Trivial-Find-Min.
- Optimal.

Number of rounds
- All the comparisons in each recursive call can be done in parallel.
- After $\log_2(n)$ recursive calls (rounds) the size of the array is 1.
- In Trivial-Find-Min there are $n - 1$ rounds.
- Optimal.
Finding the Minimum in Parallel

**Lower bound on number of rounds**

- The adversary can force any algorithm that finds the minimum to run at least $\lceil \log_2 n \rceil$ rounds.

**Proof**

- There could be at most $\lfloor |B/2| \rfloor$ useful comparisons per round since any key may participate in only one comparison.
- $\lceil \log_2 n \rceil$ rounds are required to decrease the size of $B$ from $n$ to 1 by halving.

**Optimality**

- **Parallel-Find-Min** is an optimal algorithm to find the minimum in both optimization goals: number of comparisons and number of rounds.
Finding the Minimum **And** the Maximum

**Trivial Algorithm**

\[
\text{Trivial-Find-Min-and-Max}(A[1], \ldots, A[n])
\]

\[
\begin{align*}
\text{Find-Min} & (A[1], \ldots, A[n]) \\
\text{Find-Max} & (A[1], \ldots, A[n])
\end{align*}
\]

**Correctness**

- By the correctness of **Find-Min** and **Find-Max**.

**Complexity**

- \(2(n - 1) = 2n - 2\) comparisons.
  - \((n - 1) + (n - 2) = 2n - 3\) comparisons by running **Find-Max** with only the \(n - 1\) non-minimum keys.

- At most \(2 \log_2 n\) rounds using **Parallel-Find-Min** and **Parallel-Find-Max**.
Finding the Minimum And the Maximum

Assumption

- \( n = 2^k \) is a power of 2.

The double tournament algorithm

- In parallel, compare \( n/2 \) pairs of keys from the array \( A \).
- Rearrange the array:
  - Move the \( n/2 \) smaller keys to the beginning of the array.
  - Move the \( n/2 \) larger keys to the end of the array.

- In parallel run
  - **Parallel-Find-Min** on the first \( n/2 \) keys in the array.
  - **Parallel-Find-Max** on the last \( n/2 \) keys in the array.

- Return \( A[1] \) as the minimum key and \( A[n] \) as the maximum key when the two tournaments terminate.
Finding the Minimum And the Maximum

Pseudocode

Parallel-Find-Min-and-Max(A[1],...,A[n])
for i = 1 to n/2
  if A[i] > A[i + (n/2)] (* comparison *)
  then A[i] ↔ A[i + (n/2)]

Parallel-Find-Min(A[1],...,A[n/2])
Parallel-Find-Max(A[n/2 + 1],...,A[n])

Correctness

- After the first round the minimum is in the first half of the array and the maximum is in the second half of the array.
- The rest follows by the correctness of Parallel-Find-Min and Parallel-Find-Max.
Finding the Minimum And the Maximum

Complexity for $n$ power of 2
- $\frac{n}{2} + 2 \left( \frac{n}{2} - 1 \right) = \frac{3n}{2} - 2$ comparisons.
- $1 + \log_2 \left( \frac{n}{2} \right) = \log_2 n$ rounds.

Complexity for an even $n$
- $\frac{n}{2} + 2 \left( \frac{n}{2} - 1 \right) = \frac{3n}{2} - 2$ comparisons.
- $1 + \left\lceil \log_2 \left( \frac{n}{2} \right) \right\rceil = \left\lceil \log_2(n) \right\rceil$ rounds.

Complexity for an odd $n$
- Add one round with one comparison with the extra key to partition the array into $\left\lfloor \frac{n}{2} \right\rfloor$ smaller keys and $\left\lceil \frac{n}{2} \right\rceil$ larger keys.
- $1 + \left\lfloor \frac{n}{2} \right\rfloor + (\left\lfloor \frac{n}{2} \right\rfloor - 1) + (\left\lceil \frac{n}{2} \right\rceil - 1) = \left\lceil \frac{3n}{2} \right\rceil - 2$ comparisons.
- $2 + \left\lceil \log_2 \left( \left\lfloor \frac{n}{2} \right\rfloor \right) \right\rceil = \left\lceil \log_2(n) \right\rceil + 1$ rounds.
Finding the Minimum And the Maximum

Adversary strategy

- **Key idea:** Any key in the array could potentially be the minimum or the maximum.

Adversary data structure

- $\mathcal{N}$ - Candidates for either maximum or minimum.
- $\mathcal{H}$ - Candidates only for maximum.
- $\mathcal{B}$ - Candidates only for minimum.
- $\mathcal{R}$ - Can be neither maximum nor minimum.

Initially: $\mathcal{N} = \{A[1], \ldots, A[n]\}$ and $\mathcal{H} = \mathcal{B} = \mathcal{R} = \emptyset$.

At the end: $|\mathcal{H}| = 1$, $|\mathcal{B}| = 1$, $|\mathcal{N}| = 0$, $|\mathcal{R}| = n - 2$. 
Finding the Minimum **And** the Maximum

**Adversary answer rules**

- \((R_1 : R_2) \Rightarrow A \text{ consistent answer.}\)
- \((R : H) \Rightarrow R < H.\)
- \((B : R) \Rightarrow B < R.\)
- \((N : R) \Rightarrow N < R \text{ and } N \rightarrow B.\)
- \((B : N) \Rightarrow B < N \text{ and } N \rightarrow H.\)
- \((N : H) \Rightarrow N < H \text{ and } N \rightarrow B.\)
- \((N_1 : N_2) \Rightarrow N_1 < N_2 \text{ and } N_1 \rightarrow B \text{ and } N_2 \rightarrow H.\)
- \((B : H) \Rightarrow B < H.\)
- \((B_1 : B_2) \Rightarrow B_1 < B_2 \text{ and } B_2 \rightarrow R.\)
- \((H_1 : H_2) \Rightarrow H_1 < H_2 \text{ and } H_1 \rightarrow R.\)
Theorem
- The adversary forces any algorithm that finds the minimum and the maximum to make at least $\lceil \frac{3n}{2} \rceil - 2$ comparisons.

Proof
- Non-max and non-min keys: $N \rightarrow \{B, H\} \rightarrow R$.
- Only $(N_1 : N_2)$, $(B_1 : B_2)$, and $(H_1 : H_2)$ are useful comparisons.
- $(N_1 : N_2)$ is better than $(N : R)$, $(B : N)$, $(N : H)$.
- Emptying $N$ requires at least $\lceil \frac{n}{2} \rceil$ useful comparisons.
- The fastest way to leave one key in both $B$ and $H$ requires at least $n - 2$ useful comparisons.

Optimality
- Parallel-Find-Min-and-Max is an optimal algorithm to find the minimum and the maximum.
Finding the First and the Second

Task

- **Output**: Keys $A[n]$ and $A[n - 1]$:
  - $A[n - 1] \geq A[i]$ for $1 \leq i \leq n - 2$.

Optimization Goal

- Minimize number of *comparisons* between keys.

Remark

- First and Second could also be the smallest and the second smallest keys.
Finding the First and the Second

**Trivial algorithm**

\[
\text{Trivial-Find-First-and-Second}(A[1], \ldots, A[n])
\]

\[
\begin{align*}
\text{Find-Max}(A[1], \ldots, A[n]) \\
\text{Find-Max}(A[1], \ldots, A[n-1]) \\
\text{return } (A[n] \geq A[n-1])
\end{align*}
\]

**Correctness**

- By the correctness of \textbf{Find-Max}.
- Note that the second run of \textbf{Find-Max} was done without the maximum key.

**Complexity**

\[
(n - 1) + (n - 2) = 2n - 3 \text{ comparisons}.
\]
**Finding the First and the Second**

**Key observation**
- Denote by $L$ the set of keys that were directly compared with *First*.
- Then only keys from $L$ could be *Second*.

**Proof**
- Let $x \notin L$ be a non-*First* key.
- $x$ must be smaller than another non-*First* key.
- $x$ cannot be *Second* since it is smaller than at least two keys.

**Size of $L$**
- Trivial algorithm: The size of $L$ could be $n - 1$.
- Tournament algorithm: The size of $L$ is at most $\lceil \log n \rceil$. 
Finding the First and the Second

Efficient algorithm

- **First:** Find the maximum among all the $n$ keys using the tournament algorithm.
- **Second:** Find the maximum among all the $\lceil \log_2 n \rceil$ keys in $L$ using a second tournament algorithm.

Complexity

- $(n - 1)$ comparisons to find First.
- $(\lceil \log_2 n \rceil - 1)$ comparisons to find Second.
- $n + \lceil \log_2 n \rceil - 2$ comparisons to find First and Second.

Optimality

There exists an adversary strategy that forces any algorithm that finds First and Second to perform at least $n + \lceil \log_2 n \rceil - 2$ comparisons.
The $k$-Selection Problem

**Task**

- **Input:**
  - An integer $k$, $1 \leq k \leq n$.

- **Output:** The key $A[i]$ that is the $k^{th}$ smallest key in $A$.

**Optimization goal**

- Minimize number of comparisons between keys.

**Special cases**

- **Maximum:** $k = n$.
- **Median:** $k = \lceil n/2 \rceil$ for an odd $n$.
- **Minimum:** $k = 1$. 
The \( k \)-Selection Problem

**Observation**
- The \( k \)th smallest key is the \((n + 1 - k)\)th largest key.

**Example with distinct keys**
- Input array: [21, 34, 8, 55, 13, 1, 5, 3, 89, 2, 144]
- Sorted array: [1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
- 5 is the 4th smallest and the 8th largest key.
- 13 is the median: the 6th smallest and the 6th largest key.
- 34 is the 8th smallest and the 4th largest key.
The \( k \)-Selection Problem

Observation

- The \( k \)th **smallest** key is the \( (n + 1 - k) \)th **largest** key.

Example with arbitrary keys

- Input array: \([8, 4, 4, 8, 2, 16, 4, 1, 2, 2, 4]\)
- Sorted array: \([1, 2, 2, 2, 4, 4, 4, 4, 8, 8, 16]\)
- 2 is the 2\(^{nd}\), 3\(^{rd}\), and 4\(^{th}\) **smallest**.
- 4 is the **median**: the 6\(^{th}\) **smallest** and the 6\(^{th}\) **largest**.
- 4 is also the 4\(^{th}\), 5\(^{th}\), and 7\(^{th}\) **largest** and the 5\(^{th}\), 7\(^{th}\), and 8\(^{th}\) **smallest**.
Algorithms for the $k$-Selection Problem

**Algorithm I:**
- Sort the array and find the $k^{th}$ smallest key.
- **Complexity:** $\Theta(n \log(n))$ comparisons.

**Algorithm II:**
- Repeat finding the minimum key $k$ times.
- **Complexity:** $\Theta(kn)$ comparisons:
  
  $$(n - 1) + (n - 2) + \cdots + (n - k) = kn - \frac{k(k+1)}{2}.$$  

Which algorithm is more efficient?
- Algorithm I **outperforms** Algorithm II for $k = \omega(\log(n))$.
- Algorithm II **outperforms** Algorithm I for $k = o(\log(n))$.
- For $k = \Theta(\log(n))$, both algorithms have complexity $\Theta(n \log(n))$.  

The $k$-Selection Problem

**Notations**
- $S_i$ the set of all indices $j$ such that $A[j]$ is smaller than $A[i]$:
- $G_i$ the set of all indices $j$ such that $A[j]$ is greater than $A[i]$:

**Observation**
- $A[i]$ is the $k^{th}$ smallest key iff $|S_i| \leq k - 1$ and $|G_i| \leq n - k$.

**Proof sketch**
- The $\Rightarrow$ direction: $A[i]$ cannot be larger than $k$ or more keys because it is the $k^{th}$ smallest key and cannot be smaller than $n - k + 1$ or more keys since it is $(n + 1 - k)^{th}$ largest key.
- The $\Leftarrow$ direction: Both $S_i$ and $G_i$ are too small to contain the $k^{th}$ smallest key. Therefore, $A[i]$ must be the $k^{th}$ smallest key.
The \( k \)-Selection Problem

**Example with distinct keys**

- **Input array:** \([21, 34, 8, 55, 13, 1, 5, 3, 89, 2, 144]\)
- \( k = 4 \).
  - The 4\(^{th}\) smallest key is \( A[7] = 5 \).
  - \( S_7 = \{6, 8, 10\} \Rightarrow |S_7| = 3 \leq k - 1 \).
  - \( G_7 = \{1, 2, 3, 4, 5, 9, 11\} \Rightarrow |G_7| = 7 \leq n - k \).
- \( k = 6 \) (**Median**).
  - The 6\(^{th}\) smallest key is \( A[5] = 13 \).
  - \( S_5 = \{3, 6, 7, 8, 10\} \Rightarrow |S_5| = 5 \leq k - 1 \).
  - \( G_5 = \{1, 2, 4, 9, 11\} \Rightarrow |G_5| = 5 \leq n - k \).
The $k$-Selection Problem

Example with arbitrary keys

- **Input array:** $[8, 4, 4, 8, 2, 16, 4, 1, 2, 2, 4]$
- **$k = 4$.**
  - The $4^{th}$ smallest key is $2 \in \{A[5], A[9], A[10]\}$.
  - $S = S_5 = S_9 = S_{10} = \{8\} \Rightarrow |S| = 1 \leq k - 1$.
  - $G = G_5 = G_9 = G_{10} = \{1, 2, 3, 4, 6, 7, 16\} \Rightarrow |G| = 7 \leq n - k$.

- **$k = 6$ (Median).**
  - $S = S_2 = S_3 = S_7 = S_{11} = \{5, 8, 9, 10\} \Rightarrow |S| = 4 \leq k - 1$.
  - $G = G_2 = G_3 = G_7 = G_{11} = \{1, 4, 6\} \Rightarrow |G| = 3 \leq n - k$. 
Pivot selection

- Select a pivot $p = A[i]$ for a some index $i$ from the range $[1..n]$.
- Partition the array into two sets:
  - $S$: the set of all indices $j$ such that $A[j]$ is smaller than $p$.
  - $G$: the set of all indices $j$ such that $A[j]$ is greater than $p$.

Recursive step

1. $|S| \geq k$: Recursively select the $k^{th}$ smallest key in $S$.
2. $|G| \geq n + 1 - k$: Recursively select the $(|G| - n + k)^{th}$ smallest key in $G$.
3. Otherwise, $(|S| \leq k - 1)$ AND $(|G| \leq n - k)$: Return $p$. 
A Pivot algorithm for the $k$-Selection Algorithm

Correctness

The sizes of $S$ and $G$ determine in which part of the array to look for the $k$ smallest key.

(1): The $k^{th}$ smallest key in $A$ is the $k^{th}$ smallest key in $S$.

(2): The $k^{th}$ smallest key in $A$ is the $(n + 1 - k)^{th}$ largest key in $A$ which is the $(n + 1 - k)^{th}$ largest key in $G$ which is the $(|G| - n + k)^{th}$ smallest key in $G$.

(3): The $k^{th}$ smallest key is not in $S \cup G \Rightarrow$ it is the pivot.
A Pivot algorithm for the $k$-Selection Algorithm

Example with distinct keys

- **Input:** $A = [21, 34, 8, 5, 55, 13, 1, 3, 89, 2, 144]$ and $k = 4$.
  - $p = 13$: $S = \{3, 4, 7, 8, 10\}$ and $G = \{1, 2, 5, 9, 11\}$.

- **Second instance:** $A = [8, 5, 1, 3, 2]$ and $k = 4$.
  - $p = 2$: $S = \{3\}$ and $G = \{1, 2, 4\}$.

- **Third instance:** $A = [8, 5, 3]$ and $k = 2$.
  - $p = 5$: $S = \{3\}$ and $G = \{8\}$.

- **Output:** The 4th smallest key in the original array $A$ is 5.
A Pivot algorithm for the $k$-Selection Algorithm

Example with arbitrary keys

- **Input:** $[8, 4, 4, 8, 2, 16, 4, 1, 2, 2, 4]$ and $k = 5$.
  - $p = 8$: $S = \{2, 3, 5, 7, 8, 9, 10, 11\}$ and $G = \{6\}$.

- **Second instance:** $A = [4, 4, 2, 4, 1, 2, 2, 4]$ and $k = 5$.
  - $p = 2$: $S = \{5\}$ and $G = \{1, 2, 4, 8\}$.

- **Third instance:** $A = [4, 4, 4, 4]$ and $k = 1$.
  - $p = 4$: $S = \emptyset$ and $G = \emptyset$.

- **Output:** The $5^{th}$ smallest key in the original array is $4$. 
A Pivot algorithm for the $k$-Selection Algorithm

**Complexity**

- The **partition** is done with $n - 1 = \Theta(n)$ comparison.
- For $n \geq 1$, denote by $T(n)$ the worst case number of comparisons.
- $T(1) = 0$.

**Worst case:** The pivot is always the **minimum** or the **maximum**.
  - Recursive formula: $T(n) = T(n - 1) + \Theta(n)$.
  - Solution: $T(n) = \Theta(n^2)$.

**Best case:** The pivot is always the **median**
  - Recursive formula: $T(n) = T(n/2) + \Theta(n)$.
  - Solution: $T(n) = \Theta(n)$.

**Remark**

- The complexity is **independent** of $k$. 
A Randomized $k$-Selection Algorithm

Definition:

- **Good pivot:** \((|S| \leq \frac{3n}{4}) \text{ AND } (|G| \leq \frac{3n}{4}).\)

Observations:

- \(p\) is a **good pivot** if \(p\) is the \(k^{th}\) **smallest** key for \(\frac{n}{4} \leq k \leq \frac{3n}{4}\).
- Half of the pivots are **good pivots**.

Randomized pivot selection:

- Repeat selecting a **pivot** \(p = A[i]\) for a random \(i\) from the range \([1..n]\) until finding a **good pivot**.
A Randomized $k$-Selection Algorithm

Probabilities facts
- With probability $1/2$ the random pivot is good pivot.
- Expected number of random selections to get a good pivot is 2.
- The expectation of a sum is the sum of expectations.

Notation
- $T(n)$: The expected number of comparisons for the Randomized $k$-Selection Algorithm on arrays of size $n$.

Expected number of comparisons
- $n - 1$ comparisons to do one partition.
- $2(n - 1) < 2n$ comparisons until a good partition is found.
- Recursive formula: $T(n) \leq T(3n/4) + 2n$.
  - Because with a good pivot the sizes of $S$ and $G$ are at most $3n/4$. 
A Randomized $k$-Selection Algorithm

Solving $T(n)$ using the master theorem

- $T(n) = T(3n/4) + \Theta(n)$.
  - $a = 1$.
  - $b = 4/3$.
  - $\log_b(a) = 0$.
  - $d = 1$.

- $d > \log_b(a) \implies T(n) = \Theta(n^d) = \Theta(n)$. 

Amotz Bar-Noy (CUNY) Analysis of Algorithms
A Randomized $k$-Selection Algorithm

Solving $T(n)$ with a top-down evolution

“For convenient” ignore floors and ceilings.

\[
T(n) \leq T\left(\frac{3n}{4}\right) + 2n
\]
\[
\leq T\left(\frac{9n}{16}\right) + 2\left(\frac{3}{4}\right)n + 2n
\]
\[
\leq T\left(\frac{27n}{64}\right) + 2\left(\frac{9}{16}\right)n + 2\left(\frac{3}{4}\right)n + 2n
\]
\[
\vdots
\]
\[
\leq 2n + 2\left(\frac{3}{4}\right)n + 2\left(\frac{9}{16}\right)n + \cdots + 2\left(\frac{3}{4}\right)^i n + \cdots
\]
\[
< 2n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i
\]
\[
= 2n \left(\frac{1}{1 - \frac{3}{4}}\right)
\]
\[
= 8n
\]
A Randomized $k$-Selection Algorithm

Solving $T(n)$ with a proof by induction

- **Recursive formula:** $T(n) \leq T(3n/4) + 2n$ and $T(1) = 0$.
- **Claim:** $T(n) \leq 8n$.
- **Proof:**

\[
T(n) \leq T\left(\frac{3n}{4}\right) + 2n \\
\leq 8\left(\frac{3n}{4}\right) + 2n \\
= 6n + 2n \\
= 8n
\]
A Deterministic $k$-Selection Algorithm

**Pivot selection**
- **Partition** the array into $n/5$ groups each with 5 keys.
  - Assume a “nice” value for $n$ and ignore ceilings and floors.
- **Find** the **medians** of each one of the $n/5$ groups.
- **Find** the **median** of the $n/5$ **medians** recursively.
- The **pivot** is the **median of the medians**.

$S = \{ \text{smaller than the pivot} \}$

$G = \{ \text{greater than the pivot} \}$

$\text{Medians} \rightarrow \text{Pivot}$

Amotz Bar-Noy (CUNY)
A Deterministic $k$-Selection Algorithm

**Pivot selection**
- **Partition** the array into $n/5$ groups each with 5 keys.
  - Assume a nice value for $n$ and ignore ceilings and floors.
- **Find** the *medians* of each one of the $n/5$ groups.
- **Find** the *median* of the $n/5$ *medians* recursively.
- The pivot is the *median of the medians*.

**Assumptions**
- All the keys are distinct.
- Floors and ceilings are ignored.
A Deterministic \( k \)-Selection Algorithm

**Observations**

- **\( S \)** contains the \( n/10 \) medians that are **smaller** than the pivot and the \( 2n/10 \) keys that are **smaller** than these \( n/10 \) medians.
  \[ \Rightarrow |S| \geq 3n/10 \Rightarrow |G| \leq 7n/10. \]

- **\( G \)** contains the \( n/10 \) medians that are **greater** than the pivot and the \( 2n/10 \) keys that are **greater** than these \( n/10 \) medians.
  \[ \Rightarrow |G| \geq 3n/10 \Rightarrow |S| \leq 7n/10. \]
A Deterministic $k$-Selection Algorithm

**Worst Case Number of Comparisons**
- $\Theta(n)$ to find the $n/5$ medians.
- $T(n/5)$ to find the median of the medians.
- $\Theta(n)$ for the partition.
- At most $T(7n/10)$ for the recursion.

**Recursive formula**
- $T(n) \leq T(7n/10) + T(n/5) + \Theta(n) = \Theta(n)$.
  - Because $7n/10 + n/5 = (1 - \varepsilon)n$ for a constant $\varepsilon$. 
A Deterministic $k$-Selection Algorithm

Solving the Recursive Formula

- **Formula:** $T(n) \leq T(7n/10) + T(n/5) + \alpha n$.
  - For some constant $\alpha$ that is independent of $n$.

- **Guess:** $T(n) \leq \beta n$.
  - For some constant $\beta$ that is independent of $n$.

- **Induction:** $T(n) \leq \beta (7n/10) + \beta (n/5) + \alpha n$
  
  \[= \left( (7\beta/10) + (\beta/5) + \alpha \right) n. \]

- **Set:** $\beta = 10\alpha \Rightarrow T(n) \leq \left( (7\beta/10) + (\beta/5) + (\beta/10) \right) n.$

- **Conclude:** $T(n) \leq \beta n \leq 10\alpha n.$
The Values of the Constants $\alpha$ and $\beta$

- Finding all the $n/5$ medians:
  - The median of 5 keys can be found with 6 comparisons.
  - $6(n/5) = 1.2n$ comparisons to find all the medians.

- $(2/5)n = 0.4n$ comparisons, only with the keys not in $S \cup G$, to finish the partition.

$\Rightarrow \alpha \leq 1.6.$
$\Rightarrow \beta \leq 10\alpha \leq 16.$
$\Rightarrow T(n) \leq \beta n \leq 16n.$
Why Not Groups of 3 keys?

- $S$ contains the $n/6$ medians that are smaller than the pivot and the $n/6$ keys that are smaller than these $n/6$ medians. 
  \[ \Rightarrow |S| \geq n/3 \Rightarrow |G| \leq 2n/3. \]
- Similarly, $|S| \leq 2n/3$.
- At most $T(2n/3)$ for the recursion.
- $T(n/3)$ to find the median of the medians.
- Therefore, $T(n) \leq T(2n/3) + T(n/3) + \Theta(n)$.
- The solution to this recursive formula is $T(n) = \Theta(n \log n)$. 

A Deterministic $k$-Selection Algorithm

Groups of $2k + 1$ keys for $k \geq 2$

- At most $T \left( \frac{(3k+1)n}{4k+2} \right)$ for the recursion.
- $T \left( \frac{n}{2k+1} \right)$ to find the median of the medians.
- $T(n) \leq T \left( \frac{(3k+1)n}{4k+2} \right) + T \left( \frac{n}{2k+1} \right) + \Theta(n) = \Theta(n)$.
- Therefore, $T(n) \leq \beta_k n$ for a constant $\beta_k$ that depends on $k$ but independent on $n$.
- The best $k$ is determined by the number of comparisons required to find all the $n/(2k + 1)$ medians and the number of comparisons needed to finish the partition.
The $k$-Selection Problem

Complexity summary

- **Lower bound**: $\Omega(n)$ comparisons are required for selecting the minimum or the maximum.
- **Randomized upper bound**: $O(n)$.
- **Deterministic upper bound**: $O(n)$.
- **Complexity**: $\Theta(n)$ average and worst case.

Known bounds

- **Upper bound for all $n$**: $T(n) \leq 5.43n$.
- **Asymptotic upper bound**: $T(n) \leq 2.95n + o(n)$.
- **Lower bound for all $n$**: $T(n) \geq 1.5n$.
- **Asymptotic lower bound**: $T(n) \geq 2n + o(n)$. 