2.3 QuickSort

- quicksort
- selection
- duplicate keys
- system sorts
Selection

**Goal.** Given an array of \( N \) items, find the \( k^{th} \) smallest item.

**Ex.** Min \((k = 0)\), max \((k = N - 1)\), median \((k = N/2)\).

**Applications.**

- Order statistics.
- Find the "top \( k \)."

**Use theory as a guide.**

- Easy \( N \log N \) upper bound. How?
- Easy \( N \) upper bound for \( k = 1, 2, 3 \). How?
- Easy \( N \) lower bound. Why?

**Which is true?**

- \( N \log N \) lower bound? \( \text{is selection as hard as sorting?} \)
- \( N \) upper bound? \( \text{is there a linear-time algorithm?} \)
Quick-select

**Partition array so that:**

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

**Repeat** in one subarray, depending on `j`; finished when `j` equals `k`.

```java
public static Comparable select(Comparable[] a, int k) {
    StdRandom.shuffle(a);
    int lo = 0, hi = a.length - 1;
    while (hi > lo) {
        int j = partition(a, lo, hi);
        if (j < k) lo = j + 1;
        else if (j > k) hi = j - 1;
        else return a[k];
    }
    return a[k];
}
```
Quick-select demo

Partition array so that:

- Entry $a[j]$ is in place.
- No larger entry to the left of $j$.
- No smaller entry to the right of $j$.

Repeat in one subarray, depending on $j$; finished when $j$ equals $k$.

select element of rank $k = 5$

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>21</td>
<td>28</td>
<td>65</td>
<td>39</td>
<td>59</td>
<td>56</td>
<td>22</td>
<td>95</td>
<td>12</td>
<td>90</td>
<td>53</td>
<td>32</td>
<td>77</td>
<td>33</td>
</tr>
</tbody>
</table>

$k = 5$
Quick-select demo

Partition array so that:

- Entry $a[j]$ is in place.
- No larger entry to the left of $j$.
- No smaller entry to the right of $j$.

Repeat in one subarray, depending on $j$; finished when $j$ equals $k$.

**partition on leftmost entry**

\[
\begin{array}{cccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline
50 & 21 & 28 & 65 & 39 & 59 & 56 & 22 & 95 & 12 & 90 & 53 & 32 & 77 & 33 \\
\end{array}
\]

$k = 5$
Quick-select demo

Partition array so that:

- Entry \( a[j] \) is in place.
- No larger entry to the left of \( j \).
- No smaller entry to the right of \( j \).

**Repeat** in one subarray, depending on \( j \); finished when \( j \) equals \( k \).

**partitioned array**

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>21</td>
<td>28</td>
<td>33</td>
<td>39</td>
<td>32</td>
<td>12</td>
<td>50</td>
<td>95</td>
<td>56</td>
<td>90</td>
<td>53</td>
<td>59</td>
<td>77</td>
<td>65</td>
</tr>
</tbody>
</table>
```

\( k = 5 \)
Quick-select demo

Partition array so that:

- Entry \( a[j] \) is in place.
- No larger entry to the left of \( j \).
- No smaller entry to the right of \( j \).

Repeat in one subarray, depending on \( j \); finished when \( j \) equals \( k \).

can safely ignore right subarray

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
22 & 21 & 28 & 33 & 39 & 32 & 12 & 50 & 95 & 56 & 90 & 53 & 59 & 77 & 65 \\
\end{array}
\]

\( k = 5 \)
Quick-select demo

Partition array so that:

- Entry \( a[j] \) is in place.
- No larger entry to the left of \( j \).
- No smaller entry to the right of \( j \).

Repeat in one subarray, depending on \( j \); finished when \( j \) equals \( k \).

partition on leftmost entry

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>21</td>
<td>28</td>
<td>33</td>
<td>39</td>
<td>32</td>
<td>12</td>
<td>50</td>
<td>95</td>
<td>56</td>
<td>90</td>
<td>53</td>
<td>59</td>
<td>77</td>
<td>65</td>
</tr>
</tbody>
</table>

\( k = 5 \)
Quick-select demo

Partition array so that:

- Entry $a[j]$ is in place.
- No larger entry to the left of $j$.
- No smaller entry to the right of $j$.

Repeat in one subarray, depending on $j$; finished when $j$ equals $k$.

partitioned array

\[
\begin{array}{ccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
12 & 21 & 22 & 33 & 39 & 32 & 28 & 50 & 95 & 56 & 90 & 53 & 59 & 77 & 65 \\
\end{array}
\]

$k = 5$
Quick-select demo

Partition array so that:

- Entry $a[j]$ is in place.
- No larger entry to the left of $j$.
- No smaller entry to the right of $j$.

Repeat in one subarray, depending on $j$; finished when $j$ equals $k$.
Quick-select demo

Partition array so that:
- Entry $a[j]$ is in place.
- No larger entry to the left of $j$.
- No smaller entry to the right of $j$.

Repeat in one subarray, depending on $j$; finished when $j$ equals $k$.

**partition on leftmost entry**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>21</td>
<td>22</td>
<td><strong>33</strong></td>
<td>39</td>
<td>32</td>
<td>28</td>
<td>50</td>
<td>95</td>
<td>56</td>
<td>90</td>
<td>53</td>
<td>59</td>
<td>77</td>
<td>65</td>
</tr>
</tbody>
</table>

$k = 5$
Quick-select demo

Partition array so that:

- Entry \( a[j] \) is in place.
- No larger entry to the left of \( j \).
- No smaller entry to the right of \( j \).

**Repeat** in one subarray, depending on \( j \); finished when \( j \) equals \( k \).

**partitioned array**

\[
\begin{array}{ccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
12 & 21 & 22 & 32 & 28 & 33 & 39 & 50 & 95 & 56 & 90 & 53 & 59 & 77 & 65 \\
\end{array}
\]

\( k = 5 \)
Quick-select demo

Partition array so that:

- Entry $a[j]$ is in place.
- No larger entry to the left of $j$.
- No smaller entry to the right of $j$.

Repeat in one subarray, depending on $j$; finished when $j$ equals $k$.

stop: partitioning item is at index $k$

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
12 21 22 32 28 33 39 50 95 56 90 53 59 77 65
k = 5
```
Quick-select: mathematical analysis

**Proposition.** Quick-select takes linear time on average.

**Pf sketch.**

- Intuitively, each partitioning step splits array approximately in half:

\[ N + N/2 + N/4 + \ldots + 1 \sim 2N \text{ compares.} \]

- Formal analysis similar to quicksort analysis yields:

\[
C_N = 2N + 2k \ln \left( \frac{N}{k} \right) + 2(N - k) \ln \left( \frac{N}{N - k} \right) \\
\leq (2 + 2 \ln 2) N
\]

- Ex: \((2 + 2 \ln 2) N \approx 3.38 N\) compares to find median \((k = \frac{N}{2})\).
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Duplicate keys

Often, purpose of sort is to bring items with equal keys together.

- Sort population by age.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical characteristics of such applications.

- Huge array.
- Small number of key values.
War story (system sort in C)

A beautiful bug report. [Allan Wilks and Rick Becker, 1991]

We found that qsort is unbearably slow on "organ-pipe" inputs like "01233210":

```c
main (int argc, char**argv) {
    int n = atoi(argv[1]), i, x[100000];
    for (i = 0; i < n; i++)
        x[i] = i;
    for ( ; i < 2*n; i++)
        x[i] = 2*n-i-1;
    qsort(x, 2*n, sizeof(int), intcmp);
}
```

Here are the timings on our machine:

$ time a.out 2000
real 5.85s

$ time a.out 4000
real 21.64s

$ time a.out 8000
real 85.11s
**War story (system sort in C)**

**Bug.** A `qsort()` call that should have taken seconds was taking minutes.

At the time, almost all `qsort()` implementations based on those in:

- Version 7 Unix (1979): quadratic time to sort organ-pipe arrays.
- BSD Unix (1983): quadratic time to sort random arrays of 0s and 1s.
Duplicate keys: stop on equal keys

Our partitioning subroutine stops both scans on equal keys.

Q. Why not continue scans on equal keys?
### Partitioning an array with all equal keys

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>
Duplicate keys: partitioning strategies

**Bad.** Don't stop scans on equal keys.

\[ \sim \frac{1}{2} N^2 \] compares when all keys equal

\[
\begin{array}{cccccccc}
\end{array}
\]

**Good.** Stop scans on equal keys.

\[ \sim N \lg N \] compares when all keys equal

\[
\begin{array}{cccccccc}
\end{array}
\]

**Better.** Put all equal keys in place. How?

\[ \sim N \] compares when all keys equal

\[
\begin{array}{cccccccc}
\end{array}
\]
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Sorting applications

Sorting algorithms are essential in a broad variety of applications:

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS feed in reverse chronological order.

- Find the median.
- Identify statistical outliers.
- Binary search in a database.
- Find duplicates in a mailing list.

- Data compression.
- Computer graphics.
- Computational biology.
- Load balancing on a parallel computer.

...
System sort in Java 7

Arrays.sort().

- Has one method for objects that are Comparable.
- Has an overloaded method for each primitive type.
- Has an overloaded method for use with a Comparator.
- Has overloaded methods for sorting subarrays.

Algorithms.

- Dual-pivot quicksort for primitive types.
- Timsort for reference types.

Q. Why use different algorithms for primitive and reference types?

Bottom line. Use the system sort!
Ineffective sorts

![Diagram of ineffective sorts with code snippets](http://xkcd.com/1185)
## Sorting summary

<table>
<thead>
<tr>
<th></th>
<th>inplace?</th>
<th>stable?</th>
<th>best</th>
<th>average</th>
<th>worst</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>✔️</td>
<td></td>
<td>$\frac{1}{2} N^2$</td>
<td>$\frac{1}{2} N^2$</td>
<td>$\frac{1}{2} N^2$</td>
<td>$N$ exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>✔️</td>
<td>✔️</td>
<td>$N$</td>
<td>$\frac{1}{4} N^2$</td>
<td>$\frac{1}{2} N^2$</td>
<td>use for small $N$ or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>✔️</td>
<td></td>
<td>$N \log_3 N$</td>
<td>?</td>
<td>$c N^{3/2}$</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>✔️</td>
<td></td>
<td>$\frac{1}{2} N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$ guarantee; stable</td>
</tr>
<tr>
<td>timsort</td>
<td>✔️</td>
<td></td>
<td>$N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>improves mergesort when preexisting order</td>
</tr>
<tr>
<td>quick</td>
<td>✔️</td>
<td></td>
<td>$N \log N$</td>
<td>$2 N \ln N$</td>
<td>$\frac{1}{2} N^2$</td>
<td>$N \log N$ probabilistic guarantee; fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔️</td>
<td></td>
<td>$N$</td>
<td>$2 N \ln N$</td>
<td>$\frac{1}{2} N^2$</td>
<td>improves quicksort when duplicate keys</td>
</tr>
<tr>
<td>?</td>
<td>✔️</td>
<td>✔️</td>
<td>$N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>
1.4 Analysis of Algorithms

- introduction
- observations
- mathematical models
- order-of-growth classifications
- theory of algorithms
# Commonly-used notations in the theory of algorithms

<table>
<thead>
<tr>
<th>notation</th>
<th>provides</th>
<th>example</th>
<th>shorthand for</th>
<th>used to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tilde</td>
<td>leading term</td>
<td>(~ 10 N^2)</td>
<td>10 (N^2)</td>
<td>provide approximate model</td>
</tr>
<tr>
<td>Big Theta</td>
<td>asymptotic order of growth</td>
<td>(\Theta(N^2))</td>
<td>(\frac{1}{2} N^2) (10 N^2) (5N^2 + 22N \log N + 3N)</td>
<td>classify algorithms</td>
</tr>
<tr>
<td>Big Oh</td>
<td>(\Theta(N^2)) and smaller</td>
<td>(O(N^2))</td>
<td>10 (N^2)</td>
<td>develop upper bounds</td>
</tr>
<tr>
<td>Big Omega</td>
<td>(\Theta(N^2)) and larger</td>
<td>(\Omega(N^2))</td>
<td>(\frac{1}{2} N^2) (N^5) (N^3 + 22N \log N + 3N)</td>
<td>develop lower bounds</td>
</tr>
</tbody>
</table>