Announcements

Seminar announcement

In-class midterm on Thursday. Closed book. No devices. No notes. I’ll provide scratch paper. **Be sure to bring a pen or pencil.**

The midterm will take place in 3 different rooms across campus. Your room depends on your last name:

- Last names starting with A-F go to **Stiteler Hall room B26**
- Last names starting with G-L go to **Claire Fagin Hall, room 118**
- Last names starting with M-Z go to **Towne 100 (here)**

The TAs will lead a midterm review session tonight at 8pm in Wu and Chen.
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? is selection as hard as sorting?
- $N$ upper bound? 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**

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

$k = 5$
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></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>0</td>
<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>
</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

<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 $k$.

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

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

<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>33</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 \( k \).

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>33</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**

<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>32</td>
<td>28</td>
<td>33</td>
<td>39</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 $k$.

**stop: partitioning item is at index** $k$
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:
  \[
  C_N = 2N + 2k \ln (N/k) + 2(N-k) \ln (N/(N-k))
  \leq (2 + 2 \ln 2) N
  \]
- Ex: \((2 + 2 \ln 2) N \approx 3.38 N\) compares to find median \((k = N/2)\).
Theoretical context for selection


Use theory as a guide.

- Still worthwhile to seek practical linear-time (worst-case) algorithm.
- Until one is discovered, use quick-select (if you don’t need a full sort).

**Remark.** Constants are high \(\Rightarrow\) not used in practice.
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
Bug. A `qsort()` call that should have taken seconds was taking minutes.

Why is `qsort()` so slow?

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>i</th>
<th>j</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></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>
<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>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>A</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>A</td>
<td>A</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>A</td>
<td>A</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>A</td>
<td>A</td>
<td>A</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>A</td>
<td>A</td>
<td>A</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>A</td>
<td>A</td>
<td>A</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>A</td>
<td>A</td>
<td>A</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>A</td>
<td>A</td>
<td>A</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>A</td>
<td>A</td>
<td>A</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>A</td>
<td>A</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>A</td>
<td>A</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>A</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>A</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>A</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>A</td>
</tr>
</tbody>
</table>
Duplicate keys: partitioning strategies

**Bad.** Don't stop scans on equal keys.
\[
\text{[ } \sim \frac{1}{2} N^2 \text{ compares when all keys equal } \]

\begin{align*}
\text{B A A B A B B B C C C} & \quad \text{A A A A A A A A A A A} \\
\end{align*}

**Good.** Stop scans on equal keys.
\[
\text{[ } \sim N \lg N \text{ compares when all keys equal } \]

\begin{align*}
\text{B A A B A B C C B C B} & \quad \text{A A A A A A A A A A A} \\
\end{align*}

**Better.** Put all equal keys in place. How?
\[
\text{[ } \sim N \text{ compares when all keys equal } \]

\begin{align*}
\text{A A A B B B B B C C C} & \quad \text{A A A A A A A A A A A} \\
\end{align*}
**Problem.** [Edsger Dijkstra] Given an array of $N$ buckets, each containing a red, white, or blue pebble, sort them by color.

Operations allowed.

- $\text{swap}(i, j)$: swap the pebble in bucket $i$ with the pebble in bucket $j$.
- $\text{color}(i)$: color of pebble in bucket $i$.

Requirements.

- Exactly $N$ calls to $\text{color}()$.
- At most $N$ calls to $\text{swap}()$.
- Constant extra space.
3-way partitioning

**Goal.** Partition array into three parts so that:

- Entries between \(lt\) and \(gt\) equal to the partition item.
- No larger entries to left of \(lt\).
- No smaller entries to right of \(gt\).

![Diagram showing 3-way partitioning](image)

**Dutch national flag problem.** [Edsger Dijkstra]

- Conventional wisdom until mid 1990s: not worth doing.
- Now incorporated into C library `qsort()` and Java 6 system sort.
Dijkstra 3-way partitioning demo

- Let v be partitioning item a[lo].
- Scan i from left to right.
  - (a[i] < v): exchange a[lt] with a[i]; increment both lt and i
  - (a[i] > v): exchange a[gt] with a[i]; decrement gt
  - (a[i] == v): increment i

![Diagram showing partitioning process with '<', '=', '>', and 'lt', 'i', 'gt' indicators]
Dijkstra 3-way partitioning demo

- Let v be partitioning item a[lo].
- Scan i from left to right.
  - (a[i] < v): exchange a[lt] with a[i]; increment both lt and i
  - (a[i] > v): exchange a[gt] with a[i]; decrement gt
  - (a[i] == v): increment i
Dijkstra 3-way partitioning demo

- Let $v$ be partitioning item $a[10]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$; increment both $lt$ and $i$
  - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$; decrement $gt$
  - $(a[i] == v)$: increment $i$
Dijkstra 3-way partitioning demo

- Let $v$ be partitioning item $a[lo]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$; increment both $lt$ and $i$
  - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$; decrement $gt$
  - $(a[i] == v)$: increment $i$
Dijkstra 3-way partitioning demo

- Let $v$ be partitioning item $a[lo]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$; increment both $lt$ and $i$
  - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$; decrement $gt$
  - $(a[i] == v)$: increment $i$

```
A  B  P  X  W  P  P  V  P  D  P  C  Y  Z
```

- less
- equal
- unknown
Dijkstra 3-way partitioning demo

- Let $v$ be partitioning item $a[l_0]$.
- Scan $i$ from left to right.
  
  - $(a[i] < v)$: exchange $a[l_t]$ with $a[i]$; increment both $l_t$ and $i$
  
  - $(a[i] > v)$: exchange $a[g_t]$ with $a[i]$; decrement $g_t$
  
  - $(a[i] == v)$: increment $i$
Dijkstra 3-way partitioning demo

- Let v be partitioning item a[lo].
- Scan i from left to right.
  - (a[i] < v): exchange a[lt] with a[i]; increment both lt and i
  - (a[i] > v): exchange a[gt] with a[i]; decrement gt
  - (a[i] == v): increment i

---

```
A B P Y W P P V P D P C Z X
```

<table>
<thead>
<tr>
<th>less</th>
<th>equal</th>
<th>unknown</th>
<th>greater</th>
</tr>
</thead>
</table>

Diagram with partitioning item P highlighted.
Dijkstra 3-way partitioning demo

- Let $v$ be partitioning item $a[10]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$; increment both $lt$ and $i$
  - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$; decrement $gt$
  - $(a[i] == v)$: increment $i$
Dijkstra 3-way partitioning demo

- Let $v$ be partitioning item $a[lo]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$; increment both $lt$ and $i$
  - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$; decrement $gt$
  - $(a[i] == v)$: increment $i$
Dijkstra 3-way partitioning demo

- Let \( v \) be partitioning item \( a[lo] \).
- Scan \( i \) from left to right.
  - \((a[i] < v)\): exchange \( a[lt] \) with \( a[i] \); increment both \( lt \) and \( i \)
  - \((a[i] > v)\): exchange \( a[gt] \) with \( a[i] \); decrement \( gt \)
  - \((a[i] == v)\): increment \( i \)
Let $v$ be partitioning item $a[lo]$.

Scan $i$ from left to right.

- $(a[i] < v)$: exchange $a[lt]$ with $a[i]$; increment both $lt$ and $i$
- $(a[i] > v)$: exchange $a[gt]$ with $a[i]$; decrement $gt$
- $(a[i] == v)$: increment $i$
Dijkstra 3-way partitioning demo

- Let $v$ be partitioning item $a[\text{lo}]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[\text{lt}]$ with $a[i]$; increment both $\text{lt}$ and $i$
  - $(a[i] > v)$: exchange $a[\text{gt}]$ with $a[i]$; decrement $\text{gt}$
  - $(a[i] == v)$: increment $i$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>V</th>
<th>P</th>
<th>D</th>
<th>W</th>
<th>Y</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>less</td>
<td>equal</td>
<td>unknown</td>
<td>greater</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra 3-way partitioning demo

- Let v be partitioning item a[lo].
- Scan i from left to right.
  - (a[i] < v): exchange a[lt] with a[i]; increment both lt and i
  - (a[i] > v): exchange a[gt] with a[i]; decrement gt
  - (a[i] == v): increment i
Dijkstra 3-way partitioning demo

- Let v be partitioning item a[lo].
- Scan i from left to right.
  - (a[i] < v): exchange a[lt] with a[i]; increment both lt and i
  - (a[i] > v): exchange a[gt] with a[i]; decrement gt
  - (a[i] == v): increment i
Dijkstra 3-way partitioning demo

- Let v be partitioning item a[lo].
- Scan i from left to right.
  - (a[i] < v): exchange a[lt] with a[i]; increment both lt and i
  - (a[i] > v): exchange a[gt] with a[i]; decrement gt
  - (a[i] == v): increment i
Dijkstra 3-way partitioning demo

- Let v be partitioning item a[lo].
- Scan i from left to right.
  - (a[i] < v): exchange a[lt] with a[i]; increment both lt and i
  - (a[i] > v): exchange a[gt] with a[i]; decrement gt
  - (a[i] == v): increment i
Dijkstra 3-way partitioning demo

- Let v be partitioning item a[lo].
- Scan i from left to right.
  - (a[i] < v): exchange a[lt] with a[i]; increment both lt and i
  - (a[i] > v): exchange a[gt] with a[i]; decrement gt
  - (a[i] == v): increment i

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>P</th>
<th>V</th>
<th>W</th>
<th>Y</th>
<th>Z</th>
<th>X</th>
</tr>
</thead>
</table>

\[\text{less}\quad\text{equal}\quad\text{greater}\]
Dijkstra 3-way partitioning demo

- Let \( v \) be partitioning item \( a[lo] \).
- Scan \( i \) from left to right.
  - \((a[i] < v)\): exchange \( a[lt] \) with \( a[i] \); increment both \( lt \) and \( i \)
  - \((a[i] > v)\): exchange \( a[gt] \) with \( a[i] \); decrement \( gt \)
  - \((a[i] == v)\): increment \( i \)

\[
\begin{array}{cccccccc}
\uparrow & \uparrow & \uparrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow
\end{array}
\]

**invariant**

\[
\begin{array}{cccc}
< v & = v & \text{gray} & > v \\
\uparrow & \uparrow & \uparrow & \uparrow
\end{array}
\]
3-way quicksort: Java implementation

private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    Comparable v = a[lo];
    int i = lo;
    while (i <= gt) {
        int cmp = a[i].compareTo(v);
        if (cmp < 0) exch(a, lt++, i++);
        else if (cmp > 0) exch(a, i, gt--);
        else i++;
    }
    sort(a, lo, lt - 1);
    sort(a, gt + 1, hi);
}
3-way quicksort: visual trace

equal to partitioning element
2.3 **QUICKSORT**

- quicksort
- selection
- duplicate keys
- system sorts
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.
  
  ...
Bentley-McIlroy quicksort.

- Cutoff to insertion sort for small subarrays.
- Partitioning item: median of 3 or Tukey’s ninther.
- Partitioning scheme: Bentley-McIlroy 3-way partitioning.

Very widely used. C, C++, Java 6, ....
Replacement of quicksort in java.util.Arrays with new dual-pivot quicksort

Hello All,

I'd like to share with you new Dual-Pivot Quicksort which is faster than the known implementations (theoretically and experimental). I'd like to propose to replace the JDK's Quicksort implementation by new one.

...

The new Dual-Pivot Quicksort uses *two* pivots elements in this manner:

1. Pick an elements P1, P2, called pivots from the array.
2. Assume that P1 <= P2, otherwise swap it.
3. Reorder the array into three parts: those less than the smaller pivot, those larger than the larger pivot, and in between are those elements between (or equal to) the two pivots.
4. Recursively sort the sub-arrays.

The invariant of the Dual-Pivot Quicksort is:

[ < P1 | P1 <= & <= P2 } > P2 ]

...
Replacement of quicksort in java.util.Arrays with new dual-pivot quicksort

Date: Thu, 29 Oct 2009 11:19:39 +0000
Subject: Replace quicksort in java.util.Arrays with dual-pivot implementation

Changeset: b05abb410c52
Author: alanb
Date: 2009-10-29 11:18 +0000
URL: http://hg.openjdk.java.net/jdk7/tl/jdk/rev/b05abb410c52

6880672: Replace quicksort in java.util.Arrays with dual-pivot implementation
Reviewed-by: jjb
Contributed-by: vladimir.yaroslavskiy at sun.com, joshua.bloch at google.com, jbentley at avaya.com

! make/java/java/FILES_java.gmk
! src/share/classes/java/util/Arrays.java
+ src/share/classes/java/util/DualPivotQuicksort.java

http://mail.openjdk.java.net/pipermail/compiler-dev/2009-October.txt
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!
## Sorting summary

<table>
<thead>
<tr>
<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>$\frac{1}{2}N^2$</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>$N$</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>$N \log_3 N$</td>
<td>?</td>
<td>$cN^{3/2}$</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</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>$N$</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>$N \log N$</td>
<td>$2N \ln N$</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>$N$</td>
<td>$2N \ln N$</td>
<td>$\frac{1}{2}N^2$</td>
<td>improves quicksort when duplicate keys</td>
</tr>
<tr>
<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
- memory
Types of analyses

**Best case.** Lower bound on cost.
- Determined by “easiest” input.
- Provides a goal for all inputs.

**Worst case.** Upper bound on cost.
- Determined by “most difficult” input.
- Provides a guarantee for all inputs.

**Average case.** Expected cost for random input.
- Need a model for “random” input.
- Provides a way to predict performance.

**Ex 1.** Array accesses for brute-force 3-Sum.
- Best: $\sim \frac{1}{2} N^3$
- Average: $\sim \frac{1}{2} N^3$
- Worst: $\sim \frac{1}{2} N^3$

**Ex 2.** Compares for binary search.
- Best: $\sim 1$
- Average: $\sim \lg N$
- Worst: $\sim \lg N$
Types of analyses

**Best case.** Lower bound on cost.
**Worst case.** Upper bound on cost.
**Average case.** “Expected” cost.

**Actual data might not match input model?**
- Need to understand input to effectively process it.
- Approach 1: design for the worst case.
- Approach 2: randomize, depend on probabilistic guarantee.
Theory of algorithms

Goals.

- Establish “difficulty” of a problem.
- Develop “optimal” algorithms.

Approach.

- Suppress details in analysis: analyze “to within a constant factor.”
- Eliminate variability in input model: focus on the worst case.

Upper bound. Performance guarantee of algorithm for any input.

Lower bound. Proof that no algorithm can do better.

Optimal algorithm. Lower bound = upper bound (to within a constant factor).
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><strong>Big Theta</strong></td>
<td>asymptotic order of growth</td>
<td>$\Theta(N^2)$</td>
<td>$\frac{1}{2} N^2$</td>
<td>classify algorithms</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$10 N^2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$5 N^2 + 22 N \log N + 3N$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\vdots$</td>
<td></td>
</tr>
<tr>
<td><strong>Big O</strong></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></td>
<td></td>
<td></td>
<td>$100 N$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$22 N \log N + 3N$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\vdots$</td>
<td></td>
</tr>
<tr>
<td><strong>Big Omega</strong></td>
<td>$\Theta(N^2)$ and larger</td>
<td>$\Omega(N^2)$</td>
<td>$\frac{1}{2} N^2$</td>
<td>develop lower bounds</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$N^5$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$N^3 + 22 N \log N + 3N$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\vdots$</td>
<td></td>
</tr>
</tbody>
</table>
Theory of algorithms: example 1

Goals.
- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 1-SUM = “Is there a 0 in the array?”

Upper bound. A specific algorithm.
- Running time of the optimal algorithm for 1-SUM is $O(N)$.

Lower bound. Proof that no algorithm can do better.
- Ex. Have to examine all $N$ entries (any unexamined one might be 0).
- Running time of the optimal algorithm for 1-SUM is $\Omega(N)$.

Optimal algorithm.
- Lower bound equals upper bound (to within a constant factor).
- Ex. Brute-force algorithm for 1-SUM is optimal: its running time is $\Theta(N)$. 
Theory of algorithms: example 2

Goals.
- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 3-Sum.

Upper bound. A specific algorithm.
- Ex. Brute-force algorithm for 3-Sum.
- Running time of the optimal algorithm for 3-Sum is $O(N^3)$. 
Theory of algorithms: example 2

Goals.
- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 3-SUM.

Upper bound. A specific algorithm.
- Ex. Improved algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is $O(N^2 \log N)$.

Lower bound. Proof that no algorithm can do better.
- Ex. Have to examine all $N$ entries to solve 3-SUM.
- Running time of the optimal algorithm for solving 3-SUM is $\Omega(N)$.

Open problems.
- Optimal algorithm for 3-SUM?
- Subquadratic algorithm for 3-SUM?
- Quadratic lower bound for 3-SUM?
Algorithm design approach

Start.

- Develop an algorithm.
- Prove a lower bound.

Gap?

- Lower the upper bound (discover a new algorithm).
- Raise the lower bound (more difficult).

Golden Age of Algorithm Design.

- 1970s–.
  - Steadily decreasing upper bounds for many important problems.
  - Many known optimal algorithms.

Caveats.

- Overly pessimistic to focus on worst case?
- Need better than “to within a constant factor” to predict performance.
## 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><strong>Tilde</strong></td>
<td>leading term</td>
<td>$\sim 10N^2$</td>
<td>$10N^2$</td>
<td>provide approximate model</td>
</tr>
<tr>
<td><strong>Big Theta</strong></td>
<td>asymptotic order of growth</td>
<td>$\Theta(N^2)$</td>
<td>$\frac{1}{2}N^2$ $10N^2$ $5N^2 + 22N\log N + 3N$</td>
<td>classify algorithms</td>
</tr>
<tr>
<td><strong>Big Oh</strong></td>
<td>$\Theta(N^2)$ and smaller</td>
<td>$O(N^2)$</td>
<td>$10N^2$</td>
<td>develop upper bounds</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$100N$ $22N\log N + 3N$</td>
<td></td>
</tr>
<tr>
<td><strong>Big Omega</strong></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>

**Common mistake.** Interpreting big-Oh as an approximate model.

**This course.** Focus on approximate models: use Tilde-notation
## Sorting summary

<table>
<thead>
<tr>
<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>$\frac{1}{2} N^2$</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>$N$</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>$N \log_3 N$</td>
<td>?</td>
<td>$c N^{3/2}$</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>✔️</td>
<td>$\frac{1}{2} N \log N$</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>$N$</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>$N \log N$</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>$N$</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>$N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>