Overview

- Sorting in Simple Way
- Lower Bound on Comparison Sorting
- Heapsort
- Quicksort
- Sorting in Linear Time
Selection Sort

- Repeatedly extract an entry with minimum value from the set and append it to the sorted sequence.

15 45 7 13 5 42 2 99 8 32
Selection Sort

- Repeatedly extract an entry with minimum value from the set and append it to the sorted sequence.

\[
\begin{array}{cccccccc}
15 & 45 & 7 & 13 & 5 & 42 & 2 & 99 & 8 & 32 \\
\end{array}
\]

\[
T(n) = \sum_{i=1}^{n} i = \frac{(n+1)n}{2}
\]
**Insertion Sort**

- Build up the sorted sequence incrementally, by inserting one entry at a time in an appropriate place.

```
15 45 7 13 5 42 2 99 8 32
```
**Insertion Sort**

- Build up the sorted sequence incrementally, by inserting one entry at a time in an appropriate place.

\[
\begin{array}{cccccccc}
15 & 45 & 7 & 13 & 5 & 42 & 2 & 99 & 8 & 32 \\
\end{array}
\]

\[
T(n) = \sum_{i=1}^{n} i = \frac{(n + 1)n}{2}
\]
Bubble Sort

- Bring the smallest element to the front of the list, bring the second smallest element to the second position, etc.

\[
\begin{array}{cccccccc}
15 & 45 & 7 & 13 & 5 & 42 & 2 & 99 & 8 & 32 \\
\end{array}
\]
Algorithms Sorting

Bubble Sort

- Bring the smallest element to the front of the list, bring the second smallest element to the second position, etc.

\[
T(n) = \sum_{i=1}^{n} (i - 1) = \frac{n(n - 1)}{2}
\]
Lower Bound

- Any comparison-based sorting algorithm needs $\Omega(n \log n)$ time.
- Binary tree of height $h$ has at most $2^h$ leaves (by induction).
- Decision tree in any comparison sorting must satisfy: $2^h \geq n!$

$$
\frac{h}{2^h} \\
\geq \log n! \\
> \log \left(\frac{n}{e}\right)^n \\
= n \log n - n \log e \\
= \Omega(n \log n)
$$

- using Stirling’s approximation (P. 55)
Merge Sort

- **Divide** the set into two equally large subsets.
- **Sort** each subset recursively using **MERGE** sort.
- **Merge** the two sorted subsets.

<table>
<thead>
<tr>
<th>15</th>
<th>45</th>
<th>7</th>
<th>13</th>
<th>5</th>
<th>42</th>
<th>2</th>
<th>99</th>
<th>8</th>
<th>32</th>
</tr>
</thead>
</table>
Merge Sort

- **Divide** the set into two equally large subsets.
- **Sort** each subset recursively using **MERGE** sort.
- **Merge** the two sorted subsets.

\[
\begin{array}{cccccccc}
15 & 45 & 7 & 13 & 5 & 42 & 2 & 99 & 8 & 32
\end{array}
\]

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{otherwise}
\end{cases}
\]

- \( T(n) = O(n \log n) \) (see p. 63)
Binary Heaps

- Can be represented as simple arrays.
- Children of the node stored in entry $i$ can be found in entries $2i$ and $2i + 1$. 
Heapify

[Diagram of a heap with numbers and arrows indicating the heapify process.]
Algorithms

Heapify

Sorting
Algorithms

Sorting

Heapify

```
45
 /   \
2  42  3
   /   /
  11 15 2
 /   /   /
5  8  7  6
```

5  8  7  6  11  15  2

5  8  7  11  15  2
Building a Heap

- A heap can be built by applying \texttt{HEAPIFY} to the nodes with children, starting with the one stored in the highest entry.
Building a Heap

- A heap can be build by applying `HEAPIFY` to the nodes with children, starting with the one stored in the highest entry.
Building a Heap

- A heap can be build by applying \texttt{HEAPIFY} to the nodes with children, starting with the one stored in the highest entry.
Building a Heap

- A heap can be build by applying **HEAPIFY** to the nodes with children, starting with the one stored in the highest entry.
Building a Heap

- A heap can be build by applying `HEAPIFY` to the nodes with children, starting with the one stored in the highest entry.
Building a Heap

- A heap can be build by applying `HEAPIFY` to the nodes with children, starting with the one stored in the highest entry.

- Building a heap requires $O(n \log n)$

- This analysis is not tight!
**Heapsort**

- Swap the root and the node in the last entry.
- Remove last entry.
- Heapify the root.
**Heapsort**

- Swap the root and the node in the last entry.
- Remove last entry.
- Heapify the root.
Heapsort

- Swap the root and the node in the last entry.
- Remove last entry.
- Heapify the root.
Heapsort

- Swap the root and the node in the last entry.
- Remove last entry.
- Heapify the root.
Priority Queues

- INSERT(S, x)
- MAXIMUM(S)
- EXTRACT-MAX(S)
Heap Insert
Heap Insert

```
99
/  \
/   \
45   32
/
/  \
/   \
13   42
/
/  \
/   \
8    15
/
/
5
```

```
5
/  \
/   \
8    7
/
/
66
```
Heap Insert
Heap Insert
**Quicksort**

- **Divide** Partition the set into two subsets such that each element in one subset is smaller than or equal to each element in the second subset.

- **Conquer** Sort each subset by using `QUICKSORT` recursively.

- **Combine** Do nothing!

```plaintext
QUICKSORT(A, p, r)
if p < r then
    q=PARTITION(A, p, r);
    QUICKSORT(A, p, q);
    QUICKSORT(A, q+1, r);
```
Algorithms

Sorting

Quicksort

15 45 7 13 5 42 2 99 8 32
## Quicksort

<table>
<thead>
<tr>
<th>15</th>
<th>45</th>
<th>7</th>
<th>13</th>
<th>5</th>
<th>42</th>
<th>2</th>
<th>99</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>7</td>
<td>13</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>32</td>
<td>99</td>
<td>42</td>
</tr>
</tbody>
</table>
Quicksort

15 45 7 13 5 42 2 99 8 32
8 45

15 7 13 5 2 32 99 42

7 5 2 8 13 15 32 42 45 99
Algorithms       Sorting

Quicksort

```
15 45 7 13 5 42 2 99 8 32
```

```
15 7 13 5 2 8 32 99 42 45
```

```
7 5 8 13 32 45
```

```
2 7 5 8 13 15 32 42 45 99
```
Quicksort

15 45 7 13 5 42 2 99 8 32

15 7 13 5 2 8 32 99 42 45

7 5 2 8 13 15 32 42 45 99

5 13

2 7 8 15 32 42 45 99
Quicksort

15 45 7 13 5 42 2 99 8 32

15 7 13 5 2 8 32 99 42 45

7 5 2 8 13 15 32 42 45 99

2 7 5 8 13 15 32 45 99

5

2 7 5 8 13 15 32 45 99

2 5 7 8 13 15 32 45 99
Advantages of Quicksort

- Expected running time is $\Theta(n \log n)$
- Sorts in place.
- Quicksort: Worst-case: $\Theta(n^2)$ (P. 149)
- Quicksort: Best-case: $O(n \log n)$ (P. 150)
- Quicksort: Partitioning with constant proportionality: $\Theta(n \log n)$
Randomized Quicksort

- Randomly permute the elements, or
- Select the partitioning number in a randomly chosen entry.

**RANDOMIZED-PARTITION** \((A, p, r)\)

\[ i = \text{RANDOM}(p, r); \]
\[ \text{swap}(A, r, i); \]
\[ \text{return PARTITION}(A, p, r); \]

**RANDOMIZED-QUICKSORT** \((A, p, r)\)

\[ \text{if } p < r \text{ then} \]
\[ q = \text{RANDOMIZED-PARTITION}(A, p, r); \]
\[ \text{RANDOMIZED-QUICKSORT}(A, p, q); \]
\[ \text{RANDOMIZED-QUICKSORT}(A, q+1, r); \]
Randomized Quicksort - Analysis, Worst-Case

- \( T(n) = \max_{0 \leq q \leq n-1} \{T(q) + T(n - q - 1)\} + \Theta(n) \)

- Substitution method: Guess:

\[
T(n) \leq cn^2
\]

\[
T(n) \leq \max_{0 \leq q \leq n-1} \{cq^2 + c(n - q)^2\} + \Theta(n)
\]

\[
= c \max_{0 \leq q \leq n-1} \{q^2 + (n - q - 1)^2\} + \Theta(n)
\]

\[
= cn^2 - 2c(n - 1) + \Theta(n)
\]

\[
\leq cn^2
\]

- since \( c \) can be chosen so that \( 2c(n - 1) \) dominates \( \Theta(n) \).
Randomized Quicksort - Analysis, Average-Case

- It is shown in the book (P. 156-158) that $T(n) = O(n \log n)$, i.e., average running time of \texttt{QUICKSORT} is $O(n \log n)$
Counting Sort

- Count the number of elements smaller than each given element.
- Elements to be sorted are integers between 1 and $k$.
- Requires $O(k)$ working space.
- $O(k + n)$ time.
- Stable.
Algorithms

Sorting

Radix Sort

- Sort digit for digit?
- Most significant digit first?
- Least significant digit first?
Radix Sort

- Sort digit for digit?
- Most significant digit first?
- Least significant digit first?

```plaintext
*   *   *
289 142 142 142
488 289 234 234
142 234 289 289
441 321 321 321
234 488 432 432
478 441 441 441
445 478 445 445
321 445 478 475
475 475 475 478
432 432 488 488
```
Radix Sort

- Sort digit for digit?
- Most significant digit first?
- Least significant digit first?

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>289</td>
<td>142</td>
<td>142</td>
</tr>
<tr>
<td>488</td>
<td>289</td>
<td>234</td>
</tr>
<tr>
<td>142</td>
<td>234</td>
<td>289</td>
</tr>
<tr>
<td>441</td>
<td>321</td>
<td>321</td>
</tr>
<tr>
<td>234</td>
<td>488</td>
<td>432</td>
</tr>
<tr>
<td>478</td>
<td>441</td>
<td>441</td>
</tr>
<tr>
<td>445</td>
<td>478</td>
<td>445</td>
</tr>
<tr>
<td>321</td>
<td>445</td>
<td>478</td>
</tr>
<tr>
<td>475</td>
<td>475</td>
<td>475</td>
</tr>
<tr>
<td>432</td>
<td>432</td>
<td>488</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>289</td>
<td>441</td>
<td>321</td>
</tr>
<tr>
<td>488</td>
<td>321</td>
<td>432</td>
</tr>
<tr>
<td>142</td>
<td>142</td>
<td>234</td>
</tr>
<tr>
<td>441</td>
<td>432</td>
<td>441</td>
</tr>
<tr>
<td>234</td>
<td>234</td>
<td>142</td>
</tr>
<tr>
<td>478</td>
<td>445</td>
<td>441</td>
</tr>
<tr>
<td>445</td>
<td>475</td>
<td>475</td>
</tr>
<tr>
<td>321</td>
<td>488</td>
<td>478</td>
</tr>
<tr>
<td>475</td>
<td>488</td>
<td>289</td>
</tr>
<tr>
<td>432</td>
<td>289</td>
<td>488</td>
</tr>
</tbody>
</table>
**Bucket Sort**

- Divide the interval into $n$ equal sized subintervals and distribute the elements into these subintervals. Use insertion sort on each of $n$ buckets.
Summary

- Sorting by comparison requires at least $\Omega(n \log n)$.
- There are sorting algorithms by comparison that are optimal.
- Heapsort is optimal and sorts in-place.
- Quicksort is $\Theta(n^2)$ but $O(n \log n)$ on average. Small constants, faster even for quite large problem instances.
- Additional assumptions about input or stronger computational models make it possible to sort in $\Theta(n)$ time.