Sorting

- Put data in order based on primary key
- Many methods
- Internal sorting:
  - data in arrays in main memory
- External sorting:
  - data in files on the disk
Methods

- Exchange sort
  - Bubble sort
  - Quicksort
- Selection sort
  - Straight selection sort
  - Binary Tree sort
- Insertion sort
  - Simple (linear) insertion sort
  - Shell sort
  - Address calculation sort
- Merge sort
  - Mergesort
  - Radix sort
Bubble Sort

- Bubble sort: pass through the array several times
  - compare \( x[i] \) with \( x[i-1] \)
  - swap \( x[i], x[i-1] \) if they are not in the proper order
  - the less efficient algorithm !!

```plaintext
for (i = 0, i < n - 1; i++)
    for (j = n - 1; j > i; j--)
        if (\( x[i] < x[i-1] \)) swap(\( x[i] \), \( x[i-1] \));
```
n - 1 passes
Observations

- After the first iteration the maximum element is in its proper position (last)
- After the $i$-th iteration all elements in positions greater than $(n-i+1)$ are in proper position
  - they need not to be examined again
  - if during an iteration, no element changes position, the array is already sorted
- $n-1$ iterations to sort the entire array
Complexity

- **Worst case**: \( n-i \) comparisons at each step \( i \)
  \[
  \sum_{i=1}^{n-1} (n-i) = \frac{n^2 - n}{2} \in O(n^2)
  \]

- **Average case**: \( k < n-1 \) iterations to sort the array
  \[
  \sum_{i=1}^{k} (n-i) = \frac{2kn - k^2 - k}{2} \in O(n^2)
  \]
  \( k \in O(n) \)

- **Best case**: the input is sorted
  - takes \( O(n) \) comparisons
Quicksort

- Each iteration puts an element \textit{(pivot)} in proper position
  - let \( a = x[0] \): pivot
  - if \( a \) is in proper position:
    - \( x[j] \leq a \), for all \( j \) in \([1,j-1]\)
    - \( X[j] > a \), for all \( j \) in \([j+1,n]\)

\[
\begin{array}{cccccccc}
25 & 57 & 48 & 37 & 12 & 92 & 86 & 33 \\
\end{array}
\]

\( a = x[0] \)

\[
\begin{array}{cccccccc}
(12) & 25 & 57 & 48 & 37 & 92 & 86 & 33 \\
\end{array}
\]
Algorithm

- Repeat the same for the two sub-arrays on the left and on the right of pivot a until the sub-arrays have size 1

quicksort(lb, ub) {
    if (lb < ub) {
        partition(lb, ub, a);
        quicksort(lb, a-1);
        quicksort(a+1, ub);
    }
}

- quicksort(1, n) sorts the entire array!
**input:** (25 57 48 37 12 92 86 33)  
(12) 25 (57 48 37 92 86 33)  
12 25 (57 48 37 92 86 33)  
12 25 (48 37 33) 57 (92 86)  
12 25 (37 33) 48 57 (92 86)  
12 25 (33) 37 48 57 (92 86)  
12 25 33 37 48 57 (92 86)  
12 25 33 37 48 57 (86) 92  
12 25 33 37 48 57 86 92  

**sorted array**
partition(int lb, int ub, int j) {
    int a=x[lb];  int up=ub;  int down=lb;
    do
        while (x[down] <= a) down++;
        while (x[up] > a) up--;
        if (up>down) swap (x[down],x[up]);
    while (up > down);
    swap(x[up],x[lb]);
}
a=\text{x[1]}b=25: pivot

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>57</td>
<td>48</td>
<td>37</td>
<td>12</td>
<td>92</td>
<td>86</td>
</tr>
<tr>
<td>down</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>up</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>57</td>
<td>48</td>
<td>37</td>
<td>12</td>
<td>92</td>
<td>86</td>
</tr>
<tr>
<td>down</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>up</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>57</td>
<td>48</td>
<td>37</td>
<td>12</td>
<td>92</td>
<td>86</td>
</tr>
<tr>
<td>down</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>up</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>57</td>
<td>48</td>
<td>37</td>
<td>12</td>
<td>92</td>
<td>86</td>
</tr>
<tr>
<td>down</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>up</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>57</td>
<td>45</td>
<td>37</td>
<td>12</td>
<td>92</td>
<td>86</td>
</tr>
<tr>
<td>up&gt;down</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>12</td>
<td>48</td>
<td>37</td>
<td>57</td>
<td>92</td>
<td>86</td>
</tr>
<tr>
<td>swap (12,57)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>25</td>
<td>12</td>
<td>48</td>
<td>37</td>
<td>57</td>
<td>92</td>
<td>86</td>
</tr>
<tr>
<td>down</td>
<td>up</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>down</td>
<td>up</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>down</td>
<td>up</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>down</td>
<td>up</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>up</td>
<td>down</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>up</td>
<td>down</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>up</td>
<td>down</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>up</td>
<td>down</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

up $<$ down? swap(12,25)
Complexity

- **Average case:** let $n = 2^m \Rightarrow m = \log n$ and assume that the pivot goes to the middle of the array
  - “partition” splits the array into equal size arrays
  - comparisons: $n + 2(n/2) + 4(n/4) + \ldots + n(n/n) = n \cdot m = n \log n$
  - complexity $O(n \log n)$

- **Worst case:** sorted input
  - “partition” splits the array into arrays of size $1, (n-1)$
  - comparisons: $n + (n-1) + \ldots + 2 \Rightarrow$ complexity $O(n^2)$

- **Best case:** none!

- **Advantages:** very fast, suitable for files
  - locality of reference
Selection Sort

- Iteratively select max element and put it in proper position

<table>
<thead>
<tr>
<th></th>
<th>25</th>
<th>47</th>
<th>48</th>
<th>37</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>47</td>
<td>12</td>
<td>37</td>
<td>48</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>37</td>
<td>12</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>12</td>
<td>37</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>25</td>
<td>37</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>25</td>
<td>37</td>
<td>47</td>
<td>48</td>
</tr>
</tbody>
</table>

- Complexity: $O(n^2)$ always!
Heapsort

- Put elements in priority queue, iteratively select max element and put it in proper position
- generalization of selection sort
  ```
  set pq;
  /* for i = 1 to n pqinsert(pq,x[i]); */ not needed for i = 1 to n pqmaxdelete(pq);
  ```
- **Complexity**: build a heap + select n elements from heap $O(n) + O(n \log n)$
Binary Tree Sort

- Put elements in binary search tree and “inorder”

```
4 8 10 12 30
```

- average case: build + traverse tree $O(n \log n) + O(n)$
- worst case: initially sorted array $O(n^2)$
Insertion Sort

- Input all elements, one after the other, in a sorted array

1. 25 47 48 37 12 x[0] sorted
2. 25 47 48 37 12 x[0..1] sorted
3. 25 47 48 37 12 x[0..2] sorted
4. 25 37 47 48 12 47,48 right 1 pos
5. 12 25 37 47 48 25,47,48 right 1 pos
Complexity

- **Best case**: sorted input $O(n)$
  - one comparison per element
- **Worst case**: input sorted in reverse order $O(n^2)$
  - $i$ comparisons and $i$ moves for the $i$-th element
- **Average case**: random input $O(n^2)$
Shell Sort

- Sort parts of the input separately
  - part: every $k^{th}$ element ($k = 7, 5, 3, 1$)
  - $k$ parts, sort the $k$ parts
  - decrease $k$: get larger part and sort again
  - repeat until $k = 1$: the entire array is sorted!
- Each part is more or less sorted
  - sort parts using algorithm which performs well for already sorted input (e.g., insertion sort)
k = 5, 3, 1

input  25  57  48  37  12  92  86  33

k = 5

1st step  12  57  48  33  25  92  86  37
k = 3

2nd step  25  12  33  37  48  92  86  57
k = 1

3rd step  12  25  33  37  48  57  86  92 sorted output
Observation

- The smaller the $k$ is, the larger the parts are
  - $1^{st}$ part: $x[0], x[0+k], x[0+2k], \ldots$
  - $2^{nd}$ part: $x[1], x[1+k], x[1+2k], \ldots$
  - $j^{th}$ part: $x[j-1], x[j-1+k], x[j-1+2k], \ldots$
  - $k^{th}$ part: $x[k-1], x[2k-1], x[3k-1], \ldots$
  - $i^{th}$ element of $j^{th}$ part: $x[(i-1)k+j]$
- Average case $O(n^{1.5})$ difficult to analyze
Mergesort

- **Mergesort**: split input into small parts, sort and merge the parts
  - parts of 1 element are sorted!
  - sorting of parts not necessary!
  - split input in parts of size $n/2$, $n/4$, ... 1
  - $m = \log n$ parts
- **Complexity**: always $O(n \log n)$
  - needs $O(n)$ additional space
Radix Sort

- **Radix sort**: sorting based on digit values
  - put elements in 10 queues based on less significant digit
  - output elements from 1st up to 10th queue
  - reinsert in queues based on second most significant digit
  - repeat until all digits are exhausted
  - $m$: maximum number of digit must be known

- **Complexity**: always $O(mn)$
- **additional space**: $O(n)$
input: 25 57 48 37 12 92 86 33

front       rear
queue[0]:
queue[1]:

queue[2]: 12 92
queue[3]: 33
queue[4]:
queue[5]: 25
queue[6]: 86
queue[7]: 57 37
queue[8]: 48
queue[9]:

Sorted output: 12 25 33 37 48 57 86 92
External sorting

- Sorting of large files
  - data don’t fit in main memory
  - minimize number of disk accesses
- Standard method: mergesort
  - split input file into small files
  - sort the small files (e.g., using quicksort)
  - these files must fit in the main memory
  - merge the sorted files (on disk)
  - the merged file doesn't fit in memory
Example from main memory, Complexity: $O(n \log n)$
Observations

- Initially \( m \) parts of size > 1
  - \( \log m + 1 \) levels of merging
  - complexity: \( O(n \log_2 m) \)
  - \( k \)-way merging: the parts are merged in groups of \( k \)

\[
\begin{align*}
[25] & \quad [57] & \quad [48] & \quad [37] & \quad [12] & \quad [92] & \quad [86] & \quad [33] \\
\downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow \\
[25 37 48 57 ] & \quad [12 33 86 92 ] \\
\text{k = 4}
\end{align*}
\]
External K-Way Merging

- **Disk accesses:**
  - Initially $m$ data pages (parts), merged in groups of $k$
  - $\log_k m$ levels of merging (passes over the file)
  - $m$ accesses at every level
  - $m \log_k m$ accesses total

- **Comparisons:**
  - 2-way merge: $n$ comparisons at each level
  - $k$-way merge: $kn$ comparisons at each level
  - $kn \log_k m = (k-1)/\log_2 k \ n \log_2 n$ comparisons total
## Sorting algorithms

### Complexity

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Shell Sort</td>
<td>$O(n^{1.5})$</td>
<td>?</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
Observation

- *Selection sort*: only $O(n)$ transpositions
- *Mergesort*: $O(n)$ additional space
- *Quicksort*: recursive, needs space, eliminate recursion
Theorem

- Sorting *n* elements using key comparisons only cannot take time less than $\Omega(n \log n)$ in the worst case
  - *n*: number of elements

- **Proof**: the process of sorting is represented by a decision tree
  - _nodes_ correspond to _keys_
  - _arcs_ correspond to key comparisons
Decision tree: example input \((k_1,k_2,k_3)\)

\[
\begin{align*}
\text{k}_1 & \leq \text{k}_2 \quad (1,2,3) \\
\text{k}_2 & \leq \text{k}_3 \quad (1,2,3) \\
\text{k}_1 & \leq \text{k}_3 \quad (2,1,3) \\
\end{align*}
\]
Complexity

- The decision tree has at most $n!$ leafs
  - $d$: min. depth of decision tree (full tree)
  - $d$: number of comparisons
  - at most $2^d$ leafs in full binary tree
  - $2^d \geq n! \Rightarrow d \geq \log n!$
  - given: $n! \geq (n/2)^{n/2} \Rightarrow \log_2 n! \leq n/2 \log_2 n/2$
    - $\Rightarrow n/2\log n/2 \leq d \Rightarrow d \in \Omega(n\log n)$