Sorting Algorithms

- Sorting for a short list
  - Simple sort algorithms: $O(n^2)$ time
  - Bubble sort, insertion sort, selection sort

- Popular sorting algorithm
  - Quicksort: $O(n \log n)$ time on average, $O(n^2)$ time in the worst case

  - Heapsort, mergesort: $O(n \log n)$ in the worst case, but average case behavior may not be quite as good as that of quicksort
Sorting Algorithms

- Sorting for special kinds of data
  - Bin (or bucket) sort: $O(n)$ time in the worst case
  - Works on integers from a limited range

- Internal vs. external sorting
  - Internal sorting
    - Uses the random access capability of the main memory
  - External sorting:
    - Necessary to sort too large number of objects to fit in main memory
    - Uses the main memory and secondary storage (hard disk)
Sorting

- Objects to be sorted
  - Multiple fields, one of the fields is *key*
  - Sorting is to arrange a sequence of records so that their key fields form a nondecreasing sequence
    - Sorted records: $r_1, r_2, ..., r_n$
    - Keys of sorted records: $k_1 \leq k_2 \leq ... \leq k_n$

- Evaluating running time of internal sort
  - Algorithm step
  - Number of comparisons
  - Number of data movement
Simple Sorting Schemes

- **Bubble sort**
  - The records with low key values are “light” and bubble up to the top
  - Repeated passes over the array

```plaintext
(1) for i := 1 to n-1 do
(2)   for j := n downto i+1 do
(3)     if A[j].key < A[j-1].key then
(4)       swap(A[j], A[j-1])
```

\[\text{swap}(x, y)\]
\[
\begin{align*}
\text{temp} &:= x; \\
x &:= y; \\
y &:= \text{temp}
\end{align*}
\]
Simple Sorting Schemes

- **Insertion sort**
  - In the $i^{th}$ pass, insert $A[i]$ into right place among $A[1], A[2], ..., A[i-1]$, which is previously sorted
  - Make $A[0]$ has $-\infty$, which makes the process easier

```plaintext
(1) A[0].key := -\infty;
(2) for i := 2 to n do begin
(3)     j := i;
(5)         swap(A[j], A[j-1]);
(6)     end
(7)     j := j-1
end
end
```
Simple Sorting Schemes

- **Selection sort**
  - In the $i^{th}$ pass, select the record with the lowest key among $A[i], A[i+1], \ldots, A[n]$ and swap it with $A[i]$
  - $A[1], \ldots, A[i]$ is sorted

```plaintext
begin
(1) for $i := 1$ to $n-1$ do begin
  \{ select the lowest among $A[i], \ldots, A[n]$ and swap it with $A[i]$ \}
  lowindex := $i$;
  lowkey := $A[i].key$;
(3) for $j := i+1$ to $n$ do
  \{ compare each key with current lowkey \}
  if $A[j].key < lowkey$ then begin
    lowkey := $A[j].key$;
    lowindex := $j$
  end;
(8) swap($A[i], A[lowindex]$)
end;
```
Counting Swaps

- If size of record is large, \textit{swap} takes much time
  - Bubble sort, insertion sort
    - Number of swap for bubble sort
      \[ n(n-1)/2 \approx n^2/2 \] in the worst case
    - \( n^2/4 \) on average, which is the same for insertion sort
  - Selection sort
    - In the \( i^{th} \) pass, swap places \( A[i] \) in the final place
    - Number of swap for selection sort is \( O(n) \)

- For long records, swap pointers rather than records
### Simple Sort Comparison

- **Counting operations**
  - Numbers of comparisons, swaps
  - Numbers on average, worst case

<table>
<thead>
<tr>
<th></th>
<th>#Comparisons</th>
<th></th>
<th>#Swaps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>worst</td>
<td>average</td>
<td>worst</td>
</tr>
<tr>
<td><strong>Bubble Sort</strong></td>
<td>$\frac{n^2}{2}$</td>
<td>$\frac{n^2}{2}$</td>
<td>$\frac{n^2}{2}$</td>
</tr>
<tr>
<td><strong>Insertion Sort</strong></td>
<td>$\frac{n^2}{2}$</td>
<td>$\frac{n^2}{4}$</td>
<td>$\frac{n^2}{2}$</td>
</tr>
<tr>
<td><strong>Selection Sort</strong></td>
<td>$\frac{n^2}{2}$</td>
<td>$\frac{n^2}{2}$</td>
<td>$n$</td>
</tr>
</tbody>
</table>
Quicksort

- Most efficient for internal sorting: $O(n \log n)$

- Selecting a *pivot*, key value $v$, among $A[1], \ldots, A[n]$
  - Hope the pivot is near the median key value

- Permute all the records around the pivot
  - $A[1], \ldots, A[k-1]$ : all the records with keys less than $v$
  - $A[k], \ldots, A[n]$ : all the records with keys greater than or equal to $v$

- Recursively, apply quicksort to both groups
Quicksort

- Divide-and-Conquer
  - \texttt{findpivot}(i, j) : return pivot’s index, 0 if all keys are same
  - \texttt{partition}(i, j, pivot) : permute around the pivot

\begin{verbatim}
procedure quicksort ( i, j: integer );
   begin
   (1)   pivotindex := findpivot(i, j);
   (2)   if pivotindex <> 0 then begin { do nothing if all keys are equal }
   (3)       pivot := A[pivotindex].key;
   (4)       k := partition(i, j, pivot);
   (5)       quicksort(i, k - 1);
   (6)       quicksort(k, j)
   end; { quicksort }
end
\end{verbatim}
**Quicksort**

- \( \text{partition}(i, j, \text{pivot}) \) : partition \( A[i], ..., A[j] \)
  
  *s.t. keys < \text{pivot} at the left and keys \geq \text{pivot} at the right*

```plaintext
function partition( i, j: integer; pivot: keytype ): integer;
begin
  (1)  \( l := i \);
  (2)  \( r := j \);
  repeat
    (3)  \( \text{swap}(A[l], A[r]) \);
    \{ now the scan phase begins \}
    (4)  \( \text{while } A[l].\text{key} < \text{pivot do} \)
    (5)  \( l := l + 1 \);
    (6)  \( \text{while } A[r].\text{key} \geq \text{pivot do} \)
    (7)  \( r := r - 1 \)
    until \( l > r \);
  (8)  \( l > r \);
  (9)  return (l)
end; \{ partition \}
```
Quicksort

Example
Quicksort – running time

- **Time complexity**
  - \( \text{partition}(i, j, \text{pivot}) \) will take \( O(i-j+1) \)
  - Each level of recursion, all elements are processed in one of partitioned groups, \( O(n) \) for each level
  - Recursive level is \( \log n \) on average, thus \( O(n \log n) \)

- **Worst case**
  - One partition with a single element
  - Repeating this pattern of partition
  - Worst \( \text{findpivot}() \) selection example
  - \( O(n^2) \)
Quicksort - Improvement

- Close-to-equal parts
  - Carefully select pivots that divide subarrays equally
  - Depth will be exactly $\log n$
- Pick $k$ elements at random and select a median
  - For example, if $k = 3$, two comparisons are enough
  - Fast median selection algorithm is needed
Quicksortk - improvement

- For small subarrays, use simple $O(n^2)$ algorithms
  - Knuth suggests if #elements $\leq 9$, call a simple sort

- Reduce swap count by using more space
  - Pointer swap – works for almost any sorts
  - At the end, rearrange the real records according to sorted pointers

- Negative side
  - Extra space for an array of pointers
  - Extra time to access key values via pointer references
Heapsort

- Use heap and apply DELETEMIN repeatedly
  - Heap: partially ordered tree in an array representation
  - Worst case as well as average case, time is $O(n \log n)$
  - INSERT, DELETE, MIN, DELETEMIN - $O(\log n)$

(1) \textbf{for} $x$ on list $L$ \textbf{do}
(2) \hspace{1em} INSERT($x$, $S$);
(3) \hspace{1em} \textbf{while not} EMPTY($S$) \textbf{do begin}
(4) \hspace{2em} $y :=$ MIN($S$);
(5) \hspace{2em} writeln($y$);
(6) \hspace{2em} DELETE($y$, $S$)
\textbf{end}

(4) and (6) can be combined into DELETEMIN
Heapsort

- Instead of `writeln(y)` in (5), store the deleted element \(A[1]\) in the back of array
- Repeat this process \(n\) times
Heapsort

- Initially make array $A$ into a heap
  - $\text{pushdown}(\text{first}, \text{last})$
    - Assume $A[\text{first}], \ldots A[\text{last}]$ obeys partially ordered tree property, except possibly for $A[\text{first}]$ with its children
    - The procedure pushes $A[\text{first}]$ down until partially ordered tree property is restored

- For $A[1:n], A[n/2+1], \ldots, A[n]$ are leaves, which automatically obey partially ordered tree property
- Need to call $\text{pushdown}(i, n)$ for $1 \leq i \leq n/2$
Heapsort

- Sorts array $A[1], \ldots, A[n]$ into decreasing order

```
procedure heapsort;
    begin
        { establish the partially ordered tree property initially }
        (1) for $i := n \ div \ 2$ downto 1 do
        (2)    pushdown($i, n$);
        (3) for $i := n$ downto 2 do begin
            (4)    swap($A[1], A[i]$);
                   { remove minimum from front of heap }
            (5)    pushdown(1, $i - 1$)
                   { re-establish partially ordered tree property }
        end
        end; { heapsort }
```
Bin Sorting

- $O(n)$ sort, if
  - Key values are integer,
  - Key values are in the range of 1, ..., n, and
  - No duplicated key values exist

- Two simple binsort algorithms

```plaintext
for i := 1 to n do

for i := 1 to n do
    while A[i].key <> i do
        swap(A[i], A[A[i].key]);
```
Binsort (Bucket Sort)

- In general, need to store multiple records with the same key value into a bin
  - Use an array $B$ with size $n$ for bins
  - Each bin is a list (all elements in a list have the same key value)
- Insert a record into an appropriate bin (list)
  - Assume key values are of enumerated type
- Concatenate all lists to make a sorted list
Binsort (Bucket Sort)

- **Time complexity**
  - $n$ elements to sort and $m$ different key values ($m$ lists)
  - $O(n + m)$  $O(n)$, if $n \geq m$

```plaintext
procedure binsort;
  { binsort array A, leaving the sorted list in B[lowkey] }
begin
  { place the records into bins }
  (1) for $i := 1$ to $n$ do
      { push $A[i]$ onto the front of the bin for its key }
  (2) INSERT($A[i]$, FIRST($B[A[i].key]$), $B[A[i].key]$);
  (3) for $v := \text{succ}(lowkey)$ to highkey do
      { concatenate all the bins onto the end of $B[lowkey]$ }
  (4) CONCATENATE ($B[lowkey]$, $B[v]$)
end; { binsort }
```
Sorting Large Key Sets

- Sort \( n \) elements ranging from 0, 1, 2, ..., \((n-1)^2\)
  - Example, \( n = 10 \) – 0, 1, 81, 64, 4, 25, 36, 16, 9, 49
  - Two pass bin sort – one for the first digit and the other for second digit, bin is selected with the current digit

<table>
<thead>
<tr>
<th>Bin</th>
<th>Contents</th>
<th>Bin</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0, 1, 4, 9</td>
</tr>
<tr>
<td>1</td>
<td>1, 81</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>64, 4</td>
<td>4</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>36, 16</td>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>8</td>
<td>81</td>
</tr>
<tr>
<td>9</td>
<td>9, 49</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Radix Sort

- Generalized version of sort in the previous slide
  - Keytype consists of \( k \) components, \( f_1, f_2, \ldots, f_k \)
  
  - Binsort first on \( f_k \), the least significant digit
  - Then, concatenate the bins, lowest value first
  - Binsort on \( f_{k-1} \), and so on
  - (NOTE) when inserting into bins, append each record at the end of the list
Radix Sort

- Assume $f_i$ has $s_i$ number of different values
  - $\sum_{i=1..k} O(s_i + n) = O(kn + \sum_{i=1..k} s_i) = O(n + \sum_{i=1..k} s_i)$

```plaintext
procedure radixsort;
{ sorts list A of n records with keys consisting of fields $f_1, \ldots, f_k$ }
begin
  (1) for $i := k \text{ downto } 1$ do begin
  (2)   for each value $v$ of type $t_i$ do { clear bins }
  (3)     make $B_i[v]$ empty;
  (4)   for each record $r$ on list $A$ do
  (5)     move $r$ from $A$ onto the end of bin $B_i[v]$, where $v$ is the value of field $f_i$ of the key of $r$;
  (6)   for each value $v$ of type $t_i$, from lowest to highest do
  (7)     concatenate $B_i[v]$ onto the end of $A$
  end
end; { radixsort }
```
Radix Sort

- Integer keys are in the range 0 to $d^k - 1$, for some constant $k$
  - Keys are base-$d$ integers $k$ digits long, i.e. $s_i$ is $d$
  - $O(n + \sum_{i=1..k} s_i) = O(n + kd) = O(n)$, if $d = O(n)$

```
329  720  720  329
457  355  329  355
657  436  436  436
839  457  839  457
436  657  355  657
720  329  457  720
355  839  657  839
```

\[ \uparrow \]