Sorting

- a list of records \( (R_1, R_2, \ldots, R_n) \)

- each \( R_i \) has key value \( K_i \)

- assume an ordering relation \( (\prec) \) on the keys, so that for any 2 key values \( x \) and \( y \), \( x=y \) or \( x<y \) or \( x>y \). \( \prec \) is transitive.

The sorting problem is that of finding a permutation, \( \sigma \), such that \( K_{\sigma(i)} \leq K_{\sigma(i+1)} \), \( 1 \leq i \leq n-1 \). The desired ordering is \( (R_{\sigma(1)}, R_{\sigma(2)}, \ldots, R_{\sigma(n)}) \).
Stable Sorting

Let $\sigma_s$ be the permutation with the following properties:

(1) $K_{\sigma_s(i)} \leq K_{\sigma_s(i+1)}$, $1 \leq i \leq n-1$.

(2) If $i < j$ and $K_i = K_j$ in the input list, then $R_i$ precedes $R_j$ in the sorted list.

A sorting method that generates the permutation $\sigma_s$ is **stable**.
Two main operations

• Key Compare

• Data movement
Sorting Categories

• Data location
  – Internal Sorting
  – External Sorting

• Sorting principle
  – Insert
  – Exchange
  – Selection
  – Merge
  – Multiple keys
Assume that relational operators have been overloaded so that record comparison is done by comparing their keys.
• Basic step: Insert e into a sorted sequence of i records in such a way that the resulting sequence of size i+1 is also ordered.

• Uses a[0] to simplify the while loop test.
template<class T> void Insert(const T& e, T *a, int i)
{
    // a must have space for at least i+2 elements.
    a[0]=e;
    while (e < a[i])
    {
        a[i+1]=a[i];
        i--;
        // a[i+1] is always ready for storing element
    }
    a[i+1]=e;
}
• **Insertion sort:**

• Begin with the ordered sequence \( a[1] \), then successively insert \( a[2], a[3], \ldots, a[n] \) into the sequence.

• **template**

  ```cpp
template<class T>
void InsertionSort (Element *list, const int n)
{
    //sort \( a[1:n] \) into nonincreasing order.
    for (int j=2; j<=n; j++) {
        T temp = a[j];
        Insert (temp, a, j-1);
    }
}
```
Example

\[
\begin{align*}
  i=1 & \quad (49) \quad 38 \quad 65 \quad 97 \quad 76 \quad 13 \quad 27 \\
  i=2 & \quad 38 \quad (38 \quad 49) \quad 65 \quad 97 \quad 76 \quad 13 \quad 27 \\
  i=3 & \quad 65 \quad (38 \quad 49 \quad 65) \quad 97 \quad 76 \quad 13 \quad 27 \\
  i=4 & \quad 97 \quad (38 \quad 49 \quad 65 \quad 97) \quad 76 \quad 13 \quad 27 \\
  i=5 & \quad 76 \quad (38 \quad 49 \quad 65 \quad 76 \quad 97) \quad 13 \quad 27 \\
  i=6 & \quad 13 \quad (13 \quad 38 \quad 49 \quad 65 \quad 76 \quad 97) \quad 27 \\
  i=7 & \quad 27 \quad (13 \quad 27 \quad 38 \quad 49 \quad 65 \quad 76 \quad 97)
\end{align*}
\]

Result: \( (13 \quad 27 \quad 38 \quad 49 \quad 65 \quad 76 \quad 97) \)
Analysis of insert sort

The worst case

Insert(e, a, i) makes i+1 comparisons before making insertion --- \( O(i) \).

InsertSort invokes Insert for \( i=j-1=1, 2, \ldots, n-1 \), so the overall time is

\[
O(\sum_{i=1}^{n-1} (i + 1)) = O(n^2).
\]
Insert sort

Variations:

Linked Insert Sort
Binary Insert Sort
Shell Insert Sort
Binary Insert Sort

i=1  (30)  13  70  85  39  42  6  20
i=2  13  (13  30)  70  85  39  42  6  20
i=7  6  (6  13  30  39  42  70  85 )  20
i=8  20  (6  13  30  39  42  70  85 )  20
void binsort(JD r[], int n) {
    int i, j, x, s, m, k;
    for (i = 2; i <= n; i++) {
        r[0] = r[i];
        x = r[i].key;
        s = 1; j = i - 1;
        while (s <= j) {
            m = (s + j) / 2;
            if (x < r[m].key)
                j = m - 1;
            else
                s = m + 1;
        }
        for (k = i - 1; k >= s; k--)
            r[k + 1] = r[k];
        r[s] = r[0];
    }
}
Shell Insert Sort

- Donald Shell
- Diminishing increment sort
  - Select an integer $d_1 = h < n,$
    - every $h$th elements yields a group
    - Sort the groups via simple insert sort
    - Results to $h$-sorted file
  - Select an integer $d_2 < h$
    - Grouping & sorting
  - Until $d_i = 1$
d1=5
Grouping: 49 38 65 97 76 13 27 48 55 4

Sorting: 13 27 48 55 4 49 38 65 97 76

d2=3
Grouping: 13 27 48 55 4 49 38 65 97 76

Sorting: 13 4 48 38 27 49 55 65 97 76

d3=1
Grouping: 13 27 48 55 4 49 38 65 97 76

Sorting: 4 13 27 38 48 49 55 65 76 97
void shellsort(JD r[], int n, int d[T]) {
    int i, j, k;
    JD x; k = 0;
    while (k < T) {
        for (i = d[k] + 1; i <= n; i++) {
            x = r[i];
            j = i - d[k];
            while ((j > 0) && (x.key < r[j].key)) {
                r[j + d[k]] = r[j];
                j = j - d[k];
            }
            r[j + d[k]] = x;
        }
        k++;
    }
}
#define T 3
int d[]={5,3,1};

Round 1

Round 2:

Round 3:
Shell Insert Sort

- Selection of increments (di)

- ???
• Exercises: P401-1, 3
Exchange Sorting: Bubble Sorting

Round one:

Round two:

Round three:

Round four:

Round five:

Round six:
void bubble_sort(JD r[], int n) {
    int m, i, j, flag = 1;
    JD x;
    m = n - 1;
    while((m > 0) && (flag == 1)) {
        flag = 0;
        for(j = 1; j <= m; j++)
            if(r[j].key > r[j + 1].key) {
                flag = 1;
                x = r[j];
                r[j] = r[j + 1];
                r[j + 1] = x;
            }
        m--;
    }
}
function cocktail_sort(list, list_length) {
    bottom = 0; top = list_length - 1;
    swapped = true;
    while(swapped == true) {
        swapped = false;
        for(i = bottom; i < top; i = i + 1) {
            if(list[i] > list[i + 1]) {
                swap(list[i], list[i + 1]);
                swapped = true;
            }
        }
        top = top - 1;
        for(i = top; i > bottom; i = i - 1) {
            if(list[i] < list[i - 1]) {
                swap(list[i], list[i - 1]);
                swapped = true;
            }
        }
        bottom = bottom + 1;
    }
}
Quick Analysis

• M runs/rounds

• Each run : O(n)
  – One record selected
  – Traverse the whole remaining unsorted file

• How to improve?
  – Each run: traverse part of the file
  – How?

    \[ [a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6 \ a_7 \ a_8] \rightarrow \]
    
    \[ \text{Mid} = a_1 \rightarrow \]
    
    \[ [a_2 \ a_3 \ a_4] \ \text{a}_1 \ [a_5 \ a_6 \ a_7 \ a_8] \]
Quick Sort

- **Fastest** known sorting algorithm in practice
- Average case: $O(n \log n)$
- Worst case: $O(n^2)$
  - But, the worst case seldom happens.
- Divide-and-conquer recursive algorithm
Quicksort

• **Divide step:**
  – Pick any element (*pivot*) \( v \) in \( S \)
  – Partition \( S - \{v\} \) into two disjoint groups
    \[
    S_1 = \{ x \in S - \{v\} \mid x \leq v \} \\
    S_2 = \{ x \in S - \{v\} \mid x \geq v \}
    \]

• **Conquer step**
  – Recursively sort \( S_1 \) and \( S_2 \)

• **Combine step**
  – Combine the sorted \( S_1 \), followed by \( v \), followed by the sorted \( S_2 \)
Example: Quicksort
Example: Quicksort...
Input: an array $A[p, r]$

**Quicksort** $(A, p, r)$ {
    if $(p < r)$ {
        $q = \text{Partition} (A, p, r)$ //q is the position of the pivot element
        **Quicksort** $(A, p, q-1)$
        **Quicksort** $(A, q+1, r)$
    }
}
Partitioning

- Key step of quicksort algorithm
- Goal: given the picked pivot, partition the remaining elements into two smaller sets
- Many ways to implement
- Even the slightest deviations may cause surprisingly bad results.

- We will learn an easy and efficient partitioning strategy here.
- How to pick a pivot will be discussed later
Partitioning Strategy

- Want to partition an array A[left .. right]
- First, get the pivot element out of the way by swapping it with the last element. (Swap pivot and A[right])
- Let \( i \) start at the first element and \( j \) start at the next-to-last element (\( i = \text{left}, j = \text{right} - 1 \))

\[
\begin{array}{cccccccc}
5 & 6 & 4 & 6 & 3 & 12 & 19 \\
\end{array}
\]  \hspace{1cm} \rightarrow \hspace{1cm}
\begin{array}{cccccccc}
5 & 6 & 4 & 19 & 3 & 12 & 6 \\
\end{array}

\[\text{pivot}\]
\[\text{swap}\]
\[i\]
\[j\]
Partitioning Strategy

- Want to have
  - $A[p] \leq pivot$, for $p < i$
  - $A[p] \geq pivot$, for $p > j$

- When $i < j$
  - Move $i$ right, skipping over elements smaller than the pivot
  - Move $j$ left, skipping over elements greater than the pivot
  - When both $i$ and $j$ have stopped
    - $A[i] \geq pivot$
    - $A[j] \leq pivot$
Partitioning Strategy

• When i and j have stopped and i is to the left of j
  – Swap A[i] and A[j]
    • The large element is pushed to the right and the small element is pushed to the left
  – After swapping
    • A[i] <= pivot
    • A[j] >= pivot
  – Repeat the process until i and j cross
Partitioning Strategy

- When i and j have crossed
  - Swap A[i] and pivot
- Result:
  - A[p] \leq pivot, for p < i
  - A[p] \geq pivot, for p > i
Small arrays

• For very small arrays, quicksort does not perform as well as insertion sort
  – how small depends on many factors, such as the time spent making a recursive call, the compiler, etc

• Do not use quicksort recursively for small arrays
  – Instead, use a sorting algorithm that is efficient for small arrays, such as insertion sort
Picking the Pivot

- **Use the first element as pivot**
  - if the input is random, ok
  - if the input is presorted (or in reverse order)
    - all the elements go into S2 (or S1)
    - this happens consistently throughout the recursive calls
    - Results in $O(n^2)$ behavior (Analyze this case later)

- **Choose the pivot randomly**
  - generally safe
  - random number generation can be expensive
Picking the Pivot

• Use the median of the array
  – Partitioning always cuts the array into roughly half
  – An optimal quicksort (O(N log N))
  – However, hard to find the exact median
    • e.g., sort an array to pick the value in the middle
Pivot: median of three

- We will use **median of three**
  - Compare just three elements: the leftmost, rightmost and center
  - Swap these elements if necessary so that
    - \( A[\text{left}] \) = Smallest
    - \( A[\text{right}] \) = Largest
    - Pick \( A[\text{center}] \) as the pivot
    - Swap \( A[\text{center}] \) and \( A[\text{right} - 1] \) so that pivot is at second last position (why?)

```c
int center = (left + right) / 2;
if( a[ center ] < a[ left ] )
    swap( a[ left ], a[ center ] );
if( a[ right ] < a[ left ] )
    swap( a[ left ], a[ right ] );
if( a[ right ] < a[ center ] )
    swap( a[ center ], a[ right ] );

// Place pivot at position right - 1
swap( a[ center ], a[ right - 1 ] );
```
Pivot: median of three

Example:


Swap A[center] and A[right]

Choose A[center] as pivot

Swap pivot and A[right – 1]

Note we only need to partition A[left + 1, ..., right – 2]. Why?
Main Quicksort Routine

if( left + 10 <= right )
{
    Comparable pivot = median3( a, left, right );

    // Begin partitioning
    int i = left, j = right - 1;
    for( ; ; )
    {
        while( a[ ++i ] < pivot ) { }
        while( pivot < a[ --j ] ) { }
        if( i < j )
            swap( a[ i ], a[ j ] );
        else
            break;
    }
    swap( a[ i ], a[ right - 1 ] ); // Restore pivot

    quicksort( a, left, i - 1 ); // Sort small elements
    quicksort( a, i + 1, right ); // Sort large elements
}
else // Do an insertion sort on the subarray
    insertionSort( a, left, right );
Partitioning Part

- Works only if pivot is picked as median-of-three.
  - Thus, only need to partition A[left + 1, ..., right – 2]

- j will not run past the end
  - because a[left] <= pivot

- i will not run past the end
  - because a[right-1] = pivot

```c
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```
Analysis

• Assumptions:
  – A random pivot (no median-of-three partitioning)
  – No cutoff for small arrays
• Running time
  – pivot selection: constant time $O(1)$
  – partitioning: linear time $O(N)$
  – running time of the two recursive calls
• $T(N)=T(i)+T(N-i-1)+cN$ where $c$ is a constant
  – $i$: number of elements in $S_1$
Worst-Case Analysis

• What will be the worst case?
  – The pivot is the smallest element, all the time
  – Partition is always unbalanced

\[
\begin{align*}
T(N) &= T(N - 1) + cN \\
T(N - 1) &= T(N - 2) + c(N - 1) \\
T(N - 2) &= T(N - 3) + c(N - 2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c \sum_{i=2}^{N} i = O(N^2)
\end{align*}
\]
Best-case Analysis

• What will be the best case?
  – Partition is perfectly balanced.
  – Pivot is always in the middle (median of the array).

\[
\begin{align*}
\frac{T(N)}{N} & = \frac{2T(N/2) + cN}{N/2} + c \\
\frac{T(N/2)}{N/2} & = \frac{T(N/4) + c}{N/4} + c \\
\frac{T(N/4)}{N/4} & = \frac{T(N/8) + c}{N/8} + c \\
& \vdots \\
\frac{T(2)}{2} & = \frac{T(1) + c}{1} + c \\
\frac{T(N)}{N} & = \frac{T(1) + c \log N}{1} + c \log N \\
T(N) & = cN \log N + N = O(N \log N)
\end{align*}
\]
Average-Case Analysis

• Assume
  – Each of the sizes for S1 is equally likely
• This assumption is valid for our pivoting (median-of-three) and partitioning strategy
• On average, the running time is $O(N \log N)$

Exercises: P405-1, 2, 5
Mergesort

Based on divide-and-conquer strategy

- Divide the list into two smaller lists of about equal sizes
- Sort each smaller list *recursively*
- Merge the two sorted lists to get one sorted list

How do we divide the list? How much time needed?
How do we merge the two sorted lists? How much time needed?
Dividing

• If the input list is a linked list, dividing takes $\Theta(N)$ time
  – We scan the linked list, stop at the $\lceil N/2 \rceil$ th entry and cut the link

• If the input list is an array $A[0..N-1]$:
dividing takes $O(1)$ time
  – we can represent a sublist by two integers $\text{left}$ and $\text{right}$: to divide
    $A[\text{left}..\text{Right}]$, we compute $\text{center}=(\text{left}+\text{right})/2$ and obtain
    $A[\text{left}..\text{Center}] \text{ and } A[\text{center}+1..\text{Right}]$
void mergesort(vector<int> & A, int left, int right)
{
    if (left < right) {
        int center = (left + right)/2;
        mergesort(A,left,center);
        mergesort(A,center+1,right);
        merge(A,left,center+1,right);
    }
}

Mergesort
**How to merge?**

- **Input:** two sorted array A and B
- **Output:** an output sorted array C
- **Three counters:** Actr, Bctr, and Cctr
  - initially set to the beginning of their respective arrays

1. The smaller of A[Actr] and B[Bctr] is copied to the next entry in C, and the appropriate counters are advanced
2. When either input list is exhausted, the remainder of the other list is copied to C
Example: Merge
Example: Merge...

Running time analysis:
- Clearly, merge takes $O(m_1 + m_2)$ where $m_1$ and $m_2$ are the sizes of the two sublists.

Space requirement:
- merging two sorted lists requires linear extra memory
- additional work to copy to the temporary array and back
Algorithm merge($A,p,q,r$)
Input: Subarrays $A[p..l]$ and $A[q..r]$ s.t. $p \leq l = q - 1 < r$.
(* $T$ is a temporary array. *)
1. $k = p; \ i = 0; \ l = q - 1;
2. \textbf{while} \ p \leq l \ \textbf{and} \ q \leq r
3. \quad \textbf{do if} \ A[p] \leq A[q]
4. \quad \quad \textbf{then} \ T[i] = A[p]; \ i = i + 1; \ p = p + 1;
5. \quad \quad \textbf{else} \ T[i] = A[q]; \ i = i + 1; \ q = q + 1;
6. \quad \textbf{while} \ p \leq l
7. \quad \quad \textbf{do} \ T[i] = A[p]; \ i = i + 1; \ p = p + 1;
8. \quad \textbf{while} \ q \leq r
9. \quad \quad \textbf{do} \ T[i] = A[q]; \ i = i + 1; \ q = q + 1;
10. \textbf{for} \ i = k \ \textbf{to} \ r
11. \quad \textbf{do} \ A[i] = T[i - k];
Analysis of mergesort

Let $T(N)$ denote the worst-case running time of mergesort to sort $N$ numbers.

Assume that $N$ is a power of 2.

- Divide step: $O(1)$ time
- Conquer step: $2T(N/2)$ time
- Combine step: $O(N)$ time

Recurrence equation:

\[
T(1) = 1 \\
T(N) = 2T(N/2) + N
\]
Analysis: solving recurrence

\[ T(N) = 2T\left(\frac{N}{2}\right) + N \]

\[ = 2(2T\left(\frac{N}{4}\right) + \frac{N}{2}) + N \]

\[ = 4T\left(\frac{N}{4}\right) + 2N \]

\[ = 4(2T\left(\frac{N}{8}\right) + \frac{N}{4}) + 2N \]

\[ = 8T\left(\frac{N}{8}\right) + 3N = \cdots \]

\[ = 2^k T\left(\frac{N}{2^k}\right) + kN \]

Since \( N=2^k \), we have \( k=\log_2 n \)

\[ T(N) = 2^k T\left(\frac{N}{2^k}\right) + kN \]

\[ = N + N \log N \]

\[ = O(N \log N) \]
Comparing $n \log_{10} n$ and $n^2$

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n \log_{10} n$</th>
<th>$n^2$</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.2K</td>
<td>10K</td>
<td>50</td>
</tr>
<tr>
<td>1000</td>
<td>3K</td>
<td>1M</td>
<td>333.33</td>
</tr>
<tr>
<td>2000</td>
<td>6.6K</td>
<td>4M</td>
<td>606</td>
</tr>
<tr>
<td>3000</td>
<td>10.4K</td>
<td>9M</td>
<td>863</td>
</tr>
<tr>
<td>4000</td>
<td>14.4K</td>
<td>16M</td>
<td>1110</td>
</tr>
<tr>
<td>5000</td>
<td>18.5K</td>
<td>25M</td>
<td>1352</td>
</tr>
<tr>
<td>6000</td>
<td>22.7K</td>
<td>36M</td>
<td>1588</td>
</tr>
<tr>
<td>7000</td>
<td>26.9K</td>
<td>49M</td>
<td>1820</td>
</tr>
<tr>
<td>8000</td>
<td>31.2K</td>
<td>64M</td>
<td>2050</td>
</tr>
</tbody>
</table>
An experiment

- Code from textbook (using template)
- Unix `time` utility

<table>
<thead>
<tr>
<th>$n$</th>
<th>$I_{\text{sort}}$ (secs)</th>
<th>$M_{\text{sort}}$ (secs)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.01</td>
<td>0.01</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>0.18</td>
<td>0.01</td>
<td>18</td>
</tr>
<tr>
<td>2000</td>
<td>0.76</td>
<td>0.04</td>
<td>19</td>
</tr>
<tr>
<td>3000</td>
<td>1.67</td>
<td>0.05</td>
<td>33.4</td>
</tr>
<tr>
<td>4000</td>
<td>2.90</td>
<td>0.07</td>
<td>41</td>
</tr>
<tr>
<td>5000</td>
<td>4.66</td>
<td>0.09</td>
<td>52</td>
</tr>
<tr>
<td>6000</td>
<td>6.75</td>
<td>0.10</td>
<td>67.5</td>
</tr>
<tr>
<td>7000</td>
<td>9.39</td>
<td>0.14</td>
<td>67</td>
</tr>
<tr>
<td>8000</td>
<td>11.93</td>
<td>0.14</td>
<td>85</td>
</tr>
</tbody>
</table>
Iterative merge sort

• At the beginning, interpret the input as \( n \) sorted sublists, each of size 1

• These lists are merged by pairs to obtain \( n/2 \) lists, each of length 2 (if \( n \) is odd, then one list is of length 1)

• These \( n/2 \) lists are then merged by pairs

• Repeat until only one list is get.
Pass 1
[49] [38] [65] [97] [76] [13] [27]

Pass 2 [38 49] [65 97] [13 76] [27]

Pass 3 [38 49 65 97] [13 27 76]

[13 27 38 49 65 76 97]
✓ Multiple scans of input file
✓ Given a len-sorted input file
  • After a scan-merge, (2len)-sorted file obtained
Problems

• Two-way merge algorithm

• Grouping the input file to do one pass merge
  – How?
    • Length of sub-sequence: 1 \rightarrow n
    • If $2 \times \text{len}$
      merged
    • Or, do something else

• Determine passes
  – How?
    • Length of sub-sequence: 1 \rightarrow n
2-way merge

✓ Input: list, startpos, len, endpos
2-way merge

template <class KeyType>
void merge(Element<Type> *initList, Element<Type>* mergedList, const int startPos, const int len, const int endpos)
{
    for (int i1 = startPos, i2 = i1+len, iResult = startPos;
        i1 <= i1+len-1 && i2 <= endpos;
        iResult++)
        if (initList[i1].getKey() <= initList[i2].getKey()) {
            mergedList[iResult] = initList[i1];
            i1++;
        }
    else {
        mergedList[iResult] = initList[i2];
        i2++;
    }
}
if (i1 > i+len-1)
    for (int t = i2; t <= endpos; t++)
        mergedList[iResult+t-i2] = initList[t];
else
    for (int t = i1; t <= i+len-1; t++)
        mergedList[iResult+t-i1] = initList[t];
Group & Merge

template <class KeyType>
void MergePass(Element<KeyType> *initList,
Element<KeyType> *resultList, const int n, const int len) {

for (int i = 0;
    i <= n – 2 * len;
    i += 2 * len)
    merge(initList, resultList, i, len, i+2 * len–1);

if (i+len–1 < n–1)
    merge(initList, resultList, i, len, n–1);
else
    for (int t = i; t <= n–1; t++)
        resultList[t] = initList[t];
}
Passes determining & mergesort

template <class KeyType>
void MergeSort(Element<KeyType> *list, const int n) {
    Element<KeyType> *tempList = new Element<KeyType>[n];
    for (int l = 1; l < n; l *= 2) {
        MergePass(list, tempList, n, l);
        l *= 2;
        MergePass(tempList, list, n, l);
    }
    delete [ ] tempList;
}
Analysis:

✓ Scan of input file
  ➢ First pass: length of 1
  ➢ Second pass: length of 2
  ➢ i-th pass: length of $2^i$

✓ How many passes?
  ➢ $[\log_2 n]$  

✓ Each pass: $O(n)$

✓ Time complexity of mergesort
  $O(n \log n)$  

Exercises: P412-1
Select sort

- Naïve algorithm

- Basic idea
  - Select the smallest item by n-1 comparisons
    - Exchange it with the first item
  - Select the smallest item of remaining n-1 items by n-2 comparisons
    - Exchange it with the second item
  - ……
  - Repeat n-1 times
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
</tr>
</tbody>
</table>

\[
i = 1
\]

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>38</td>
<td>65</td>
<td>97</td>
<td>76</td>
<td>49</td>
<td>27</td>
</tr>
</tbody>
</table>

\[
i = 2
\]

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
</tr>
</tbody>
</table>

\[
i = 2
\]

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>27</td>
<td>65</td>
<td>97</td>
<td>76</td>
<td>49</td>
<td>38</td>
</tr>
</tbody>
</table>

\[
i = 2
\]

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>27</td>
<td>[65</td>
<td>97</td>
<td>76</td>
<td>49</td>
<td>38</td>
</tr>
</tbody>
</table>

\[
i = 2
\]

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>27</td>
<td>38</td>
<td>[97</td>
<td>76</td>
<td>49</td>
<td>65</td>
</tr>
</tbody>
</table>

\[
i = 2
\]

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>27</td>
<td>38</td>
<td>49</td>
<td>[76</td>
<td>97</td>
<td>65</td>
</tr>
</tbody>
</table>

\[
i = 2
\]

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>27</td>
<td>38</td>
<td>49</td>
<td>65</td>
<td>[97</td>
<td>76</td>
</tr>
</tbody>
</table>

\[
i = 2
\]

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>27</td>
<td>38</td>
<td>49</td>
<td>65</td>
<td>76</td>
<td>[97</td>
</tr>
</tbody>
</table>
```c
void smp_selesort(JD r[], int n) {
    int i, j, k;
    JD x;
    for (i = 1; i < n; i++) {
        k = i;
        for (j = i + 1; j <= n; j++)
            if (r[j].key < r[k].key)
                k = j;
        if (i != k) {
            x = r[i];
            r[i] = r[k];
            r[i] = x;
        }
    }
}
```

- **Analysis**
  - $O(n^2)$
Heap sort

- Initialize a heap
- Output min/max
- Adjust the heap
- Repeat output/adjust until …
Heap sort

- With no extra space
  - Swap and readjust
template <class T>
void HeapSort (T *list, const int n)
{
    // Sort a[1:n] into nondecreasing order.
    for (int i=n/2; i>=1; i--)   // convert list into a heap
        Adjust(a, i, n);
    for (i=n-1; i>=1; i--)         // sort
    {
        swap(a[1], a[i+1];   // swap first and last
        Adjust(a, 1, i);    // recreate heap
    }
}
(a) Input list

(a) Input list

(a) Input list
(b) Initial heap
(c) Heap size=9
Sorted=[77]
(d) Heap size=8
Sorted=[61, 77]
(e) Heap size=7
Sorted=[59, 61, 77]

(f) Heap size=6
Sorted=[48, 59, 61, 77]
(g) Heap size = 5
[26, 48, 59, 61, 77]

(h) Heap size = 4
[19, 26, 48, 59, 61, 77]
(i) Heap size=3
[15, 19, 26, 48, 59, 61, 77]

(j) Heap size=2
[11, 15, 19, 26, 48, 59, 61, 77]

(j) Heap size=1
[5, 11, 15, 19, 26, 48, 59, 61, 77]
Analysis of HeapSort:

• suppose $2^{k-1} \leq n < 2^k$, the tree has $k$ levels.

• the number of nodes on level $i \leq 2^{i-1}$.

• in the first loop, Adjust is called once for each node that has a child, hence the time is no more than

$$
\sum_{1 \leq i \leq k-1} 2^{i-1} (k - i) =
\sum_{1 \leq i \leq k-1} 2^{k-i-1} i \leq n \sum_{1 \leq i \leq k-1} i/2^i < 2n = O(n)
$$
in the next loop, n-1 applications of Adjust are made with maximum depth \( k = \lfloor \log_2(n+1) \rfloor \).
The total time: \( O(n \log n) \).
Additional space: \( O(1) \).

**Exercises:** P416-1, 2
Radix Sort

• Extra information: every integer can be represented by at most k digits
  – $d_1 d_2 \ldots d_k$ where $d_i$ are digits in base $r$
  – $d_1$: most significant digit
  – $d_k$: least significant digit
Radix sort

• **Origin**: Herman Hollerith’s card-sorting machine for the 1890 U.S. Census

• Digit-by-digit sort.

• Hollerith’s original (bad) idea: sort on most-significant digit first.

• Good idea: Sort on *least-significant digit first* with auxiliary *stable* sort.
Operation of radix sort

3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
4 5 7
3 5 5
8 3 9
7 2 0
3 5 5
4 3 6
4 5 7
6 5 7
3 2 9
8 3 9
3 5 5
6 5 7
8 3 9
Correctness of radix sort

*Induction on digit position*

- Assume that the numbers are sorted by their low-order $t-1$ digits.
- Sort on digit $t$
Correctness of radix sort

Induction on digit position

• Assume that the numbers are sorted by their low-order \( t - 1 \) digits.

• Sort on digit \( t \)
  - Two numbers that differ in digit \( t \) are correctly sorted.
Correctness of radix sort

**Induction on digit position**

- Assume that the numbers are sorted by their low-order \( t - 1 \) digits.

- Sort on digit \( t \)
  - Two numbers that differ in digit \( t \) are correctly sorted.
  - Two numbers equal in digit \( t \) are put in the same order as the input \( \Rightarrow \) correct order.
Radix Sort

• Algorithm
  – sort by the least significant digit first
    => Numbers with the same digit go to same bin
  – reorder all the numbers: the numbers in bin 0
    precede the numbers in bin 1, which precede the
    numbers in bin 2, and so on
  – sort by the next least significant digit
  – continue this process until the numbers have been
    sorted on all k digits
Algorithm RadixSort\((A, n, d)\)

1. \(\textbf{for } 0 \leq p \leq 9\quad// \text{ base 10}\)
2. \(\textbf{do } Q[p] := \text{ empty queue};\quad// \text{ FIFO}\)
3. \(D := 1;\)
4. \(\textbf{for } 1 \leq k \leq d\quad// \text{ d times of counting sort}\)
5. \(\textbf{do }\)
6. \(D := 10 \times D;\)
7. \(\textbf{for } 0 \leq i < n\quad// \text{ scan } A[i], \text{ put into correct slot}\)
8. \(\textbf{do } t := (A[i] \mod D) \div (D/10);\)
9. \(\textbf{do } \text{ enqueue}(A[i], Q[t]);\)
10. \(j := 0;\)
11. \(\textbf{for } 0 \leq p \leq 9\quad// \text{ re-order back to original array}\)
12. \(\textbf{do while } Q[p] \text{ is not empty}\)
13. \(\textbf{do } A[j] := \text{ dequeue}(Q[p]);\)
14. \(j := j + 1;\)
Starting:

```
278 → 109 → 063 → 930 → 589 → 184 → 505 → 269 → 008 → 083
```

```
```

```
930
```

```
```

```
930 → 063 → 083 → 184 → 505 → 278 → 008 → 109 → 589 → 269
```
template <class T>
int RadixSort (T *a, const int d, const int r, const int n) 
{
    int e[r], f[r]; // queue end and front pointers

    // create initial chain of records starting at first
    int first=1;
    for (int i=1; i<n; i++) link[i]=i+1; // linked into a chain
    link[n]=0;

    for (i=d-1; i>=0; i--)
    { // sort on digit i
        fill(f, f+r, 0); // initialize bins to empty queues
        for (int current=first; current; current=link[current])
            { // put records into queues
            
            }
```c
int k = digit(a[current], i, r);
if (f[k] == 0) f[k] = current;
else link[e[k]] = current;
e[k] = current;
}
for (int j = 0; !f[j]; j++) { // find first nonempty queue
    first = f[j]; int last = e[j];
    for (int k = j + 1; k < r; k++) { // concatenate remaining queues
        if (f[k]) {
            link[last] = f[k]; last = e[k];
        }
    }
link[last] = 0;
}
return first;
```
\[ f[0] = 4 \quad e[0] = 4 \]
\[ f[1] = 0 \quad e[1] = 0 \]
\[ f[2] = 0 \quad e[2] = 0 \]
\[ f[3] = 3 \quad e[3] = 10 \]
\[ f[4] = 6 \quad e[4] = 6 \]
\[ f[5] = 7 \quad e[5] = 7 \]
\[ f[6] = 0 \quad e[6] = 0 \]
\[ f[7] = 0 \quad e[7] = 0 \]
\[ f[8] = 1 \quad e[8] = 9 \]
\[ f[9] = 2 \quad e[9] = 8 \]
f[0] = 7  e[0] = 2
f[1] = 0  e[1] = 0
f[9] = 0  e[9] = 0
\[
\begin{align*}
\text{f}[0] &= 9, & \text{e}[0] &= 10 \\
\text{f}[1] &= 2, & \text{e}[1] &= 6 \\
\text{f}[2] &= 8, & \text{e}[2] &= 1 \\
\text{f}[3] &= 0, & \text{e}[3] &= 0 \\
\text{f}[4] &= 0, & \text{e}[4] &= 0 \\
\text{f}[5] &= 7, & \text{e}[5] &= 5 \\
\text{f}[6] &= 0, & \text{e}[6] &= 0 \\
\text{f}[7] &= 0, & \text{e}[7] &= 0 \\
\text{f}[8] &= 0, & \text{e}[8] &= 0 \\
\text{f}[9] &= 4, & \text{e}[9] &= 4
\end{align*}
\]
Radix Sort

• Increasing the base $r$ decreases the number of passes $k$ (e.g. 999)

• Running time
  – $k$ passes over the numbers
  – each pass takes $O(N+r)$
  – total: $O(Nk+rk)$
  – $r$ and $k$ are constants: $O(N)$

Exercises: P422-1, 3, 5
# Summary of Internal Sorting

<table>
<thead>
<tr>
<th>Method</th>
<th>Worst</th>
<th>Average</th>
<th>Working Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$n^2$</td>
<td>$n \log n$</td>
<td>$O(n)$ or $O(\log n)$</td>
</tr>
<tr>
<td>n</td>
<td>Insert</td>
<td>Heap</td>
<td>Merge</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>0</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>50</td>
<td>0.004</td>
<td>0.009</td>
<td>0.008</td>
</tr>
<tr>
<td>100</td>
<td>0.011</td>
<td>0.019</td>
<td>0.017</td>
</tr>
<tr>
<td>200</td>
<td>0.033</td>
<td>0.042</td>
<td>0.037</td>
</tr>
<tr>
<td>300</td>
<td>0.067</td>
<td>0.066</td>
<td>0.057</td>
</tr>
<tr>
<td>400</td>
<td>0.117</td>
<td>0.090</td>
<td>0.079</td>
</tr>
<tr>
<td>500</td>
<td>0.179</td>
<td>0.116</td>
<td>0.100</td>
</tr>
<tr>
<td>1000</td>
<td>0.662</td>
<td>0.245</td>
<td>0.213</td>
</tr>
<tr>
<td>2000</td>
<td>2.439</td>
<td>0.519</td>
<td>0.459</td>
</tr>
<tr>
<td>3000</td>
<td>5.390</td>
<td>0.809</td>
<td>0.721</td>
</tr>
<tr>
<td>4000</td>
<td>9.530</td>
<td>1.105</td>
<td>0.972</td>
</tr>
<tr>
<td>5000</td>
<td>15.935</td>
<td>1.410</td>
<td>1.271</td>
</tr>
</tbody>
</table>
• For average behavior, we can see:
  • Quick Sort outperforms the other sort methods for suitably large n.

• the break-even point between Insertion and Quick Sort is near 100, let it be $n_{\text{Break}}$.

• when $n < n_{\text{Break}}$, Insert Sort is the best, and when $n > n_{\text{Break}}$, Quick Sort is the best.

• improve Quick Sort by sorting sublists of less than $n_{\text{Break}}$ records using Insertion Sort.

Experiments: P435-4
External Sorting

• External-memory algorithms
  – When data do not fit in main-memory

• What does it mean?
  – Sort atomic-operations accomplished with part of data in memory
  – Controllable Disk I/Os

• Which internal sorting algorithm is applicable?
  – Insert sort
  – Exchange sort
  – Select sort
  – Merge sort
External Sorting

• Rough idea:
  • sort pieces that fit in main-memory  
    – known as runs
  • “merge” them
• In each level: merge *runs* (sorted sequences) of size $x$ into runs of size $2x$, decrease the number of runs twofold.

• **What would it mean to run this on a file in external memory?**
  – One pass, one disk scan!
External-Memory Merge Sort

• Input file $X$, empty file $Y$

• *Phase 1*: Repeat until end of file $X$:
  – Read the next $M$ elements from $X$
  – Sort them in main-memory
  – Write them at the end of file $Y$

• *Phase 2*: Repeat while there is more than one run in $Y$:
  – Empty $X$
  – $\text{MergeAllRuns}(Y, X)$
  – $X$ is now called $Y$, $Y$ is now called $X$
External-Memory Merging

- *MergeAllRuns*(Y, X): repeat until the end of Y:
  - Call *TwoWayMerge* to merge the next two runs from Y into one run, which is written at the end of X
- *TwoWayMerge*: uses three main-memory arrays of size B

![Diagram showing file merging process](image_url)
Analysis: Assumptions

• Assumptions and notation:
  – Disk page size:
    • $B$ data elements
  – Data file size:
    • $N$ elements, $n = N/B$ disk pages
  – Available main memory:
    • $M$ elements, $m = M/B$ pages
Analysis

• Phase 1:
  – Read file X, write file Y: $2n = O(n)$ I/Os

• Phase 2:
  – One iteration: Read file Y, write file X: $2n = O(n)$ I/Os
  – Number of iterations: $\log_2 \frac{N}{M} = \log_2 \frac{n}{m}$
Analysis: Conclusions

• Total running time of external-memory merge sort: $O(n \log_2 n/m)$
Can we do better?

- $\log_2 n/m$ I/Os
  - Decrease $n/m$?
  - Initial runs – the size of available main memory ($M$ data elements) ????

$log_2 n/m$
Can we do better?

- Idea 1: decrease number of passes
- Increase the size of initial runs!
Run Generation (self study)

27  16  60

15

30  25

45  49  50  35
We can do better!

- Merge tree: Binary-tree with \( n \) leaf elements
- I/O cost = \( \Sigma (R_i \cdot L_i) \)!
- Huffman Tree!
- How to Minimize I/O?
Analysis: Conclusions

• Total running time of external-memory merge sort: $O(n \log_2 n/m)$

• We can do better!

• Observation:
  – Phase 1 uses all available memory
  – Phase 2 uses just 3 pages out of $m$ available!!!
Two-Phase, Multiway Merge Sort

• Idea: merge all runs at once!
  – Phase 1: the same (do internal sorts)
  – Phase 2: perform $\text{MultiwayMerge}(Y,X)$
Multiway Merging

Read, when \( p_i = B \)

\[ \min(Bf_1[p_1], Bf_2[p_2], \ldots, Bf_k[p_k]) \]

Write, when \( Bfo \) full
Analysis of TPMMS

- **Phase 1**: $O(n)$, Phase 2: $O(n)$
- **Total**: $O(n)$ I/Os!
- The catch: files only of “limited” size can be sorted
  - Phase 2 can merge a maximum of $m-1$ runs.
  - Which means: $N/M < m-1$
General Multiway Merge Sort

• What if a file is very large or memory is small?

• General *multiway merge sort*:
  – Phase 1: the same (do internal sorts)
  – Phase 2: do as many iterations of merging as necessary until only one run remains
• **Phase 1**: $O(n)$, each iteration of phase 2: $O(n)$

• How many iterations are there in phase 2?
  - Number of iterations: $\log_{m-1}\frac{N}{M} = \log_m n$
  - Total running time: $O(n \log_m n)$ I/Os
Conclusions

- External sorting can be done in $O(n \log_m n)$ I/O operations for any $n$
  - This is asymptotically optimal

- In practice, we can usually sort in $O(n)$ I/Os
  - Use two-phase, multiway merge-sort
• Exercises: P457-2