Elementary Sorting Methods

- Simple sorting methods
  - Useful if the sorting program is to be used a few times, or if the sorting program is used over a small amount of data to be sorted at any time
  - Useful for small files, because of low overhead
  - Useful for well-structured files – one with almost sorted elements, or with multiple records with the same key
  - Most of them are $O(N^2)$

Rules of the game

- Sorting algorithms will be applied to *files* of *items* (*records*) containing *keys*
  - Keys are part of the items that are used to control the sort
- Objective of sorting algorithm is to rearrange the items such that their keys are ordered according to some well-defined ordering rule (numerical or alphabetic)
- We will not be concerned about specific characteristics of keys but only about some criterion that can be used to put the records in order based on the key
  - Check out the `man` page for `qsort(3)` and the usage of the comparison function
- Internal sorting – File fits into memory, and does not require an extra array for keeping temporary values
- External sorting – File requires an extra array for keeping temporary values, and possibly the entire file
  - External sort may require items to be accessed in a sequential manner, and was more prevalent in the days past when memory was very expensive
- Program 6.1, reading assignment. Compare it with a program based on `qsort(3)`
- Nonadaptive sort
  - The operation to compare and exchange is fixed
  - You may not be able to specify your own criterion for comparison, and may not be able to specify the order of sort (ascending or descending)
  - Exemplified by Program 6.1
- Adaptive sort
  - Allows you to perform different sequences of operations
  - Well-suited for hardware operations, and general-purpose sorting implementations
  - Exemplified by `qsort(3)`

  Definition 1 *Stable sorting. A sorting method is said to be* ***stable*** *if it preserves the relative order of items with duplicated keys in file.*

- Indirect sort
  - Useful technique when the items (records) to be sorted are large relative to the keys
  - Worthwhile to sort keys and associate a pointer with the keys to point to actual records

- Exemplified by indices in the databases

Selection sort

- One of the simplest sorting algorithms
- At every step, find the smallest element in the array and exchange it with the first element, then, remove the first element from consideration

```
void selection ( item * a, const int l, const int r, bool * compare )
{
    for ( int i (l ); i < r; i++ )
    {
        for ( int min ( i ), int j ( i + 1 ); j <= r; j++ )
            if ( compare ( a[j], a[min] ) )    // a[j] < a[min]
                min = j;
        swap ( a[i], a[min] );
    }
}
```

- Run time of selection sort is almost independent of the amount of order in the file
  - Selection in each iteration does not use any information from selection in previous iterations
  - Same performance in files with almost sorted data, or all keys almost equal, or randomly ordered file
- Good for sorting files with large items and small keys
  - Requires very little data movement compared to other algorithms
  - Only one swap in every iteration

Insertion sort

- Insert an element in its proper place by moving the already sorted elements to make space for the new element
- Elements to the left of current index are in sorted order during the sort though they may not be in their final position yet
- Array is fully sorted when the index reaches the right end
- Implementation is straightforward but inefficient

Bubble sort

Performance characteristics of elementary sorts

- Each of the above sorting algorithms is quadratic-time and in-place
  - *in-place sorting* – Only a constant number of elements of the input array are ever stored outside the array

Property 5 *Selection sort uses about* $\frac{N^2}{2}$ *comparisons and* N *exchanges.*

Property 6 *Insertion sort uses about* $\frac{N^2}{4}$ *comparisons and* $\frac{N^2}{4}$ *half-exchanges (moves) on the average and twice as many in the worst case.*

Property 7 *Bubble sort uses about* $\frac{N^2}{2}$ *comparisons and* $\frac{N^2}{2}$ *exchanges on the average and in the worst case.*

Shellsort

- Simple extension of insertion sort that gains speed by allowing exchanges of elements that are far apart
- If every hth element in an array is sorted with respect to other elements that are in a location which is a multiple of h, the array is said to be h-sorted
  - An h-sorted file is h independent sorted files, interleaved together
- By h-sorting for some large value of h, we can move elements in the array by large distances and make it easier to sort for smaller values of h
- 5-sorting a string

```
A s o r t I n g e x A m p l e
A s o r t A n g e x I m p l e

a S o r t a N g e x i M p l e
a N o r t a S g e x i M p l e
a M o r t a N g e x i S p l e

a m O r t a n G e x i s P l e
a m G r t a n O e x i s P l e

a m g R t a n o E x i s p L e
a m g E t a n o R x i s p L e
a m g E t a n o L x i s p R e

a m g e T a n o l X i s p r E
a m g e E a n o l T i s p r X
```

  Now the elements can be sorted by insertion sort much faster as they are already closer to their final resting place compared to when we started
- For large arrays, we can come up with a sequence $h_0, h_1, h_2, \ldots$ such that each h_i is greater than h_{i-1} to allow for fast final placement of elements
- The most important thing in shellsort is to come up with the sequence that will optimize the number of comparisons and movements
  - There is no provably optimal sequence
  - General idea is to use geometrically decreasing sequences so that the number of increments is logarithmic to the size of file
- Program for shellsort with a sequence of $1, 4, 13, 40, 121, \ldots$

```
void shellsort ( item * a, const int l, const int r )
{
    int h ( 1 );
    for ( ; h <= ( r - l ) / 3; h = 3 * h + 1 );
    for ( ; h > 0; h /= 3 )
        for ( int i ( l + h ); i <= r; i++ )
        {
            for ( int j(i), item v(a[i]); j >= l + h && v < a[j-h]; j -= h )
```

```
                a[j] = a[j-h];
            a[j] = v;
        }
}
```

The above sequence is good because the h_is are relatively prime to each other leading to comparisons between different elements in successive passes

- A very good sequence for shellsort is given by $4^{i+1} + 3.2^i + 1$ for $i > 0$
- The sequence 2^i for $i \geq 0$ is bad as the odd numbered elements are not compared to the even-number elements until the final pass
- Nobody has been able to analyze the algorithm (you get an A if you analyze it correctly) ;-) and there is no functional form of the running time for shellsort

Property 8 *The result of h-sorting a file that is k-ordered is a file that is both h- and k-ordered.*

Property 9 *Shellsort does less than $N(h-1)(k-1)/g$ comparisons to g-sort a file that is h- and k-ordered, provided that h and k are relatively prime.*

Property 10 *Shellsort does less than $O(N^{3/2})$ comparisons for the increments given by the sequence $h_i = 3h_{i-1} + 1$ for $i > 0$, with $h_0 = 1$.*

Property 11 *Shellsort does less than $O(N^{4/3})$ comparisons for the increments given by the sequence $4^{i+1} + 3.2^i + 1$.*

Sorting of other types of data

- Reading assignment

Index and pointer sorting

- Reading assignment

Sorting of linked lists

- Reading assignment

Key-indexed counting

- This algorithm takes advantage of special properties of keys to perform sorting
- Assumption: Each of the n input elements is an integer in the range 1 to k
- For each input element x, determine the number of elements less than x
- You can count the keys in an array by

```
for ( int i(0); i < n; b[key(a[i])] = a[i++] );
```

This creates the index of keys

- Not an in-place sorting algorithm
- Algorithm code:

```
void counting_sort ( item * A, item * B, const int num_keys )
{
    int * c = new int[num_keys];
    bzero ( c, num_keys * sizeof ( int ) );
    for ( int i ( 0 ); i < A.num_elements(); c[A[i++]]++ );
    for ( int i ( 1 ); i < num_keys; i++ )
        c[i] += c[i-1];
    for ( int i ( A.num_elements() - 1 ); i >= 0; i-- )
        B[--c[A[i]]] = A[i];
}
```

- Counting sort is stable and runs in $O(n)$ time
- In case of duplicate keys, you can count the number of instances of each key in the first pass and distribute the elements in the proper place in the second pass

Property 12 *Key-indexed counting is a linear-time sort, provided that the range of distinct key values is within a constant factor of the file size.*

Quicksort

- The most widely used sorting algorithm (standard sorting library function in Unix is called `qsort(3)`)
- Developed in 1960 by C.A.R. Hoare
- Performs in-place sorting using a small auxiliary stack
- Extremely short inner loop
- Worst case running time – $O(n^2)$
- Average case running time – $O(n \lg n)$
- Not stable
- Based on divide-and-conquer approach

  Divide. The array `A[l..r]` is partitioned into two nonempty subarrays `A[l..m]` and `A[m+1..r]` such that each element of `A[l..m]` is less than or equal to each element of `A[m+1..r]`
  - The index `m` is computed as a part of this partitioning procedure
  - As a result of partition, the element `A[m]` is in its final place
  - Achieved by the function `partition`

  Conquer. The two subarrays `A[l,m-1]` and `A[m+1,r]` are sorted by recursive calls to quicksort.

  Combine. Since the subarrays are sorted in place, no work is needed to combine them.

- The implementation

```
void quicksort ( item * A, const int l, const int r )
{
    if ( l < r )
    {
        int m ( partition ( A, l, r) ):
```

```
        quicksort ( A, l, m-1 );
        quicksort ( A, m+1, r );
    }
}
```

To sort the entire array, the initial call is `quicksort ( A, 1, A.length() - 1 )`

- The partitioning process always puts at least one element in its proper place

```
int partition ( item * A, const int l, const int r )
{
    int pivot ( A[l] );      // Left element in the array is the pivot
    int i ( l - 1), j ( r + 1 );
    while ( i < j )
    {
        do
            j--
        while A[j] > pivot;
        do
            i++
        while A[i] <= pivot;
        if ( i < j )
            swap ( A[i], A[j] );
    }
    swap ( A[l], A[j] );
    return ( j );
```

Performance of Quicksort

- Worst-case partitioning
  - Sorting an already sorted array
  - Recurrence

    $$T_n = T_{n-1} + \Theta(n)$$

    Observe that $T_1 = \Theta(1)$

    $$\begin{aligned} T_n &= T_{n-1} + \Theta(n) \\ &= \sum_{k=1}^{n} \Theta(k) \\ &= \Theta\left(\sum_{k=1}^{n} k\right) \\ &= \Theta(n^2) \end{aligned}$$

- Best-case partitioning
  - Partitioning procedure produces two regions of size n/2
  - Recurrence

    $$T_n = 2T_{\frac{n}{2}} + \Theta(n)$$

  - Solution from Case 2 of master theorem – $O(n \lg n)$
- Balanced partitioning

  - $T_n = T_{\frac{9n}{10}} + T_{\frac{n}{10}} + \Theta(n)$
- Average case partitioning

Randomized version of quicksort

- Selecting pivot as a random element
- Selecting pivot as median-of-three elements

Merging and Mergesort

Priority Queues and Heapsort

Priority Queues

- A data structure to maintain a set S of elements, each with an associated value called a *key*

  Definition 2 *A **priority queue** is a data structure of items with keys that support two basic operations: insert a new item, and remove the term with the largest key.*

- Forms the basis for the data structure called *heap*

Elementary implementation of priority queue

- Unordered array
  - Definition

```
class priority_queue
{
    int    size;        // Number of elements in the queue
    item  *data;        // The actual records
public:
    item get_maximum ( void );
    void insert_element ( const item );
};
```

  - `get_maximum()` has to be implemented by linear scan of the array
  - `insert_element()` is simple
  - We can find implementations where one of the above two operations take constant time but finding an efficient method to perform *both* is a challenge
- Ordered array

Heap

- Formed like a complete binary tree

  Definition 3 *A tree is **heap-ordered** if the key in each node is greater than or equal to the keys in all of the node's children (if any). Equivalently, the key in each node of a heap-ordered tree is less than or equal to the key in that node's parent.*

Property 13 ***Heap property****. No node in a heap has a key larger than the key at the root.*

Definition 4 *A* ***heap*** *is a set of nodes with keys arranged in a complete heap-ordered binary tree, represented as an array.*

- Data

16	14	10	8	7	9	3	2	4	1

- Properties of complete binary tree represented as array
  - parent(i) = $\lfloor i/2 \rfloor$
  - left_child(i) = $2 \times i$
  - right_child(i) = $2 \times i + 1$
- Heap property

$$\texttt{A[parent(i)]} \geq \texttt{A[i]}$$

- Height of a node in a tree – Number of edges on the longest simple downward path from the node to a leaf
- Height of the tree – Height of the root
- Height of a heap with n elements – $\Theta(\lg n)$

Algorithms on heap

- Heapify, or fix the heap
  - This process works by first violating the heap property and then, fixing it
  - The algorithm works only on a heap where there is a single violation of heap property
  - Two possible cases for heapify
    1. The priority of some node is increased, or a new node is added to the bottom of the heap
    2. The priority of some node is decreased, or the root node is removed from the heap
  - Code for heapify (top-down)

```
void heapify ( heap *h, int node, const int nelements )
{
    // Ensure that heap property is satisfied in the subtree with root index
    // given by node

    if ( node > nelements ) return;

    int left ( node * 2 ), right ( node * 2 + 1 );
    int max ( node );

    if ( left <= nelements && h[left] > h[node] )
        max = left;

    if ( right <= nelements && h[right] > h[max] )
        max = right;

    if ( max != node )
```

```
    {
        swap ( h[node], h[max] );
        heapify ( heap, max, nelements );    // max is index and not maximum element
    }
}
```

  - This code is used to fix the heap after removing the largest element, or after increasing the priority of an element

- Building a heap

```
void build_heap ( heap *h, const int nelements )
{
    for ( int i ( nelements / 2 ); i; heapify ( h, i--, nelements ) );
}
```

  - The first element of heap is always considered to be 1 (to account for left and right child)

- Algorithm analysis

Cost of each call to heapify	$O(\lg n)$
Number of calls to `heapify()`	n
Cost of `build_heap()`	$O(n \lg n)$

Property 14 *The* `insert_element()` *and* `remove_max()` *operations for the priority queue ADT can be implemented with heap-ordered trees such that* `insert_element()` *requires no more than* $\lg N$ *comparisons and* `remove_max()` *no more than* $2 \lg N$ *comparisons, when performed on an* N*-item queue.*

Property 15 *The* `change_priority()`*,* `remove_max()`*, and* `replace_max()` *operations for the priority queue ADT can be implemented with heap-ordered trees such that no more than* $2 \lg N$ *comparisons are required for any operation on an* N*-item queue.*

Heapsort

- Use `build_heap()` and `heapify()` to perform sorting
  - Notice that `heapify()` maintains the heap property from the given node downwards, and leaves the nodes above it undisturbed
- Algorithm `heapsort`

```
void heapsort ( item * A, const int nelements )
{
    build_heap ( A, nelements );
    int heap_elements ( nelements );
    for ( int i ( nelements ); i > 2; )
    {
        swap ( A[1], A[i--] );
        heapify ( A, 1, i );
    }
}
```

Property 16 `build-heap()` *executes in linear time.*

Property 17 *Heapsort uses fewer than* $O(N \lg N)$ *comparisons to sort* N *elements.*

Property 18 *Heap-based selection allows the kth largest of N items to be found in time proportional to N when k is small or close to N, and in time proportional to $N \lg N$ otherwise.*

Priority-queue ADT

- Inserting an element into the queue

```
void insert ( priority_queue S, int& nelements, const item x )
{
    int i ( ++nelements );
    while ( i > 1 && S[i/2] < x )
    {
      S[i] = S[i/2];     // Move parent's data into current node
      i /= 2;
    }
    S[i] = item;
}
```

- Getting the maximum element

```
item get_max ( priority_queue S )
{
    return ( S[1] );
}
```

- Extracting the maximum element

```
item extract_max ( priority_queue S, int& nelements )
{
    item max ( S[1] );
    S[1] = S[nelements--];
    heapify ( S, nelements, 1 );
    return ( max );
}
```

Radix Sorting

- All methods studied so far can be classified as *comparison sorts*
  - Given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$ to determine their relative order.
  - May be unnecessary – you look at the first few letters of a person's last name to search for him in the phone book
- The radix sort algorithms are based on properties of keys; instead of comparing keys, they process and compare pieces of keys, or bytes
- Virtually any key that can be represented in a computer can be treated as a string of bits, or radix 2 integer
- Easy to implement in languages such as C/C++ because of availability of low-level bit manipulation operations
- Two types of radix sort
  1. MSD radix sort compares key from left (most significant bit) to right

2. LSD radix sort compares keys from right (least significant bit) to left

Bits, bytes, and words

- Most processing done in terms of *words* – smallest unit of data to be transferred between memory and CPU at any one time
- Words generally made up of bytes – smallest addressable unit of memory
- Bytes are made up of bits – smallest unit of information in a computer

Binary quicksort

- Also known as *radix exchange sort*

```
void radix_exchange ( item * A, const int left, const int right, const int bit )
{
    if ( right > left ) and ( bit >= 0 )
    {
        int i ( left ), j ( right );
        do
        {
            while ( bits ( A[i], bit, 1 ) == 0 && i < j ) i++;
            while ( bits ( A[j], bit, 1 ) == 1 && i < j ) j--;
            swap ( A[i], A[j] );
        } while ( j != i );
        if ( bits ( A[right], bit, 1 ) == 0 ) j++;
        radix_exchange ( A, left, j-1, bit-1 );
        radix_exchange ( A, j, right, bit-1 );
    }
}
```

Lower bounds for sorting

- Assume that all of the input elements are distinct
- No need for comparisons of type $a_i = a_j$
- Assume that all comparisons have the form $a_i \leq a_j$

The Decision-Tree Model

- Decision tree to represent the comparison sorts
- Each leaf a permutation of the array to be sorted, plus the elements to be compared $(a_i : a_j)$
- Leaves contain the sorted array

Lower bound for the worst case

- Length of the longest path from the root of a decision tree to any of its leaves.

- Height of the decision tree

Theorem 1 *Any decision tree that sorts n elements has height $\Omega(n \lg n)$.*

Proof Consider a decision tree of height h that sorts n elements
Possible number of distint sorted orders = Number of permutations = $n!$
The tree has at least $n!$ leaves
But a binary tree of height h cannot have more than 2^h leaves
Therefore, $n! \leq 2^h$
or, $h \geq \lg(n!)$
From Stirling's approximation, we have $n! > (\frac{n}{e})^n$
Therefore,

$$\begin{aligned} h &\geq \lg\left(\frac{n}{e}\right)^n \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) \end{aligned}$$

Corollary 1 *Heapsort and mergesort are asymptotically optimal comparison sorts.*

Radix Sort

Straight Radix Sort

- Originally used by card sorting machines
- Used to sort records of information keyed by multiple fields

Bucket Sort

Assumption

- Elements distributed uniformly over the range $[0, 1)$

Divide the interval $[1, 0)$ into n equal sized intervals, called *buckets*
Distribute the input numbers into the buckets
Sort the numbers in each bucket and then, go through the buckets in order, listing the elements in each