Elementary Sorts

- always consider that complex sorts often only become useful (or faster) when considering large datasets
- generally N^2 for basic datasets (mostly worst case)
- think of sorting as the sorting of files of items containing keys
- if the file fits in memory it is called internal sorting, otherwise it is called external sorting
- consider both arrays and linked lists
- NOTE: pass upper and lower array bounds in C/C++ to make the interfaces more flexible
- adaptive sort - a sort that performs different sequences of operations depending on the output of comparisons (or the initial state of the data)
- non-adaptive sort - a sort that performs the same sequence of operations on all input data
- performance:

```
N^2                 NlogN           N^3/2
bubble sort                         shellsort
selection sort
insertion sort
```

- basics:

```
template<class Item>
void exch(Item a[], size_t i, size_t j) {
    Item t = a[i];
    a[i] = a[j];
    a[j] = t;
}

template<class Item>
void compexch(Item a[], size_t i, size_t j) { if (a[j] < a[i]) { exch(a, i, j); }}

template<class Item>
void sort(Item a[], int l, int r) {
    for (int i = l + 1; i <= r; i++) {
        for (int j = i; j > l; j--) {
            compexch(a, a[j - 1], a[j]);
        }
    }
}
```

```
template<class Item>
void exch(Item a[], size_t i, size_t j) {
    Item t = a[i];
    a[i] = a[j];
    a[j] = t;
}

template<class Item>
void compexch(Item a[], size_t i, size_t j) { if (a[j] < a[i]) { exch(a, i, j); }}

template<class Item>
void sort(Item a[], int l, int r) {
    for (int i = l + 1; i <= r; i++) {
        for (int j = i; j > l; j--) {
            compexch(a, j - 1, j);
        }
    }
}
```

- two best approaches to improving performance:
    - find a better algorithm
    - tighten inner loops
- consider memory cost also
    - in place sorts (0 extra memory)
    - linked list sorts that need N extra memory for pointers
    - sorts that require a temporary second copy of the data
- stable sorts preserve the relative order of items with duplicate keys
    - stable: insertion, merge, bubble
    - not stable: heap, quick, shell
- two types of sorting, by keys or items
- indirect sort -> sort an array of pointers, references, or indices rather than the items themselves

Selection sort

- general properties:
    - not stable
    - O(1) extra space
    - Θ(n ^ 2) comparisons
    - Θ(n) swaps
    - not adaptive

- find smallest element, exchange with element in first position find smallest element in remaining and exchange with element in 2nd position etc.

```
template<typename Comparable>
void sort(Comparable a[], size_t size) {  // Sort a[] into increasing order.
    for (int i = 0; i < size; i++) {  // Exchange a[i] with smallest entry in a[i+
        int min = i;                 // index of minimal entr.
        for (int j = i + 1; j < size; j++) {
            if (a[j] < a[min])) {
                min = j;
            }
        }
        exch(a, i, min);
    }
}
```

Insertion sort

- general properties:
    - stable
    - O(1) extra space
    - O(n ^ 2) comparisons and swaps
    - adaptive: O(n) time when nearly sorted
    - very low overhead
- the C++ version provides performance improvements over standard versions

```
template<typename Comparable>
void sort(Comparable a[], size_t size) {  // Sort a[] into increasing order.
    for (int i = 1; i < N; i++) {  // Insert a[i] among a[i-1], a[i-2], a[i-3]...
        for (int j = i; j > 0 && a[j] < a[j - 1]; j--) {
            exch(a, j, j - 1);
        }
    }
}
```

Bubble sort

- general properties:
    - stable
    - O(1) extra space
    - O(n ^ 2) comparisons and swaps
    - adaptive: O(n) when nearly sorted

```
template<typename Comparable>
void sort(Comparable a, size_t size) {
    for (int i = 0; i < size; i++) {
        for (int j = r; j > i; j--) {
            compexch(a, j - 1, j);
        }
    }
}
```

- very similar to insertion sort, review Java code to determine exact differences
- the difference between the two is that in insertion sort the inner loop moves through the left sorted part of the array and in bubble sort it moves through the right, unsorted part of the array
- the inner loop can also be optimized as above
- can remove the non-adaptive compexch and and test for no exchanges in an inner loop and break out of outer loop

Performance

- selection, insertion, and bubble are all quadratic time (N^2)
- generally proportional to the number of compares, the number of exchanges, or both
- one exchange for each i from 1 to N - 1 and N - i comparisons, totals to N - 1 exchanges and ((N - 1) + (N - 2) + ... + 2 + 1) = N * (N - 1) / 2 comparisons
- insertion sort uses N ^ 2 / 4 comparisons and N ^ 2 / 4 exchanges on average and twice that at worst
- insertion sort uses N ^ 2 / 2 comparisons and N ^ 2 / 2 exchanges on average and in the worst case
- insertion sort and selection sort a 2 X bubble sort for smaller input
- all three are bad for large randomly ordered inputs
- when comparisons are expensive, insertion sort is faster
- when exchanges are expensive, selection sort is best
- an inversion is a pair of keys out of order in an input set
- the inversion count for an input set indicates how far (or close) the array is from/to being sorted
- if an input set is already sorted then the inversion count is 0
- if an input set is sorted in reverse order then the inversion count is at a max

- two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j
- insertion uses linear number of comparisons and exchanges when the input is constant having more than a constant number of inversions
- selection sort runs in linear time for files with large items and small keys

Shellsort

- general info:
  - not stable
  - O(1) extra space
  - O(n ^ 3/2) time as shown *
  - Adaptive: O(n·lg(n)) time when nearly sorted
- insertions sort moves items one place at a time, shellsort is like insertion sort but allows spaced exchanges
- shellsort takes every nth element starting from any position, sorts a segment and then interleaves segments

```
template<typename Comparable>
void sort(Comparable a[], size_t size) {
    int h = 1;
    while (h < size / 3) {
        h = 3 * h + 1;
    }
    for (int i = h; i < size; i++) {
        while (h >= 1) {
            for (int j = i; j >= h && less(a[j], a[j - h]); j -= h) {
                exch(a, j, j - h);
            }
            h = h / 3;
        }
    }
}
```

- general process -> for a modularity of h = 3 * h + 1, insertion sort each modular indexed element starting from 0 reduce h to h / 3, repeat
- different increment sequences can improve worst case performance for large files
- current was suggested by Knuth in 1969
- there are bad sequences including the original proposed by Shell
- the precise formula for running times is not known

- properties
    - each pass brings the file closer to sorted order
    - the result of h-sorting an input sequence that is k-ordered (sorted) results in a sequence that is h and k-ordered
    - does less than N(h - 1)(k - 1) / g comparisons to g-sort a sequence that is h and k-sorted
    - does less than O(N ^ 3/2) for h = 1,4,13,40,121,364,1093
    - does less than O(N ^ 4/3) for h = 1,8,23,281,1073,4193
    - does less than O(N(log N) ^ 2) for 1,2,3,4,6,9,12,18,27,16,24,36
- performant increment sequences:

```
- 1 2 4 8 32 64 128 256 512 1024 2048 // not great
- 1 4 13 40 121 364 1093 3280 9841
- 1 2 4 10 23 51 113 249 548 1207 2655 5843 *
- 1 8 23 77 281 1073 4193 16577 *
- 1 7 8 49 56 64 343 392 448 512 2401 2744
- 1 5 19 41 109 209 505 929 2161 3905 * BEST
```

Sorting of Other Types of Data

- need to move to a more general type than arrays of numbers or characters
- generic array driver components:
    - generic ADT to encapsulate many data types
    - an exchange and compare/exchange functions (swap and compare/swap)
    - initialization of randomized data and the print functions
    - (optionally a timer etc)
- array ADT:
    - random (initialization)
    - scan (initialization from standard in)
    - print
    - sort
- item ADT:
    - key
    - value
    - operator < (or compareTo)
    - scan (init from std in)
    - random (init)
    - print

Index and Pointer Sorting

- for string sorting the item ADT just contains a pointer to the str

```
struct Item {
    char *str;
}
```

- use a struct to facilitate operator overloading
- many options for custom string interfaces (dynamic memory, etc)
- always faced with memory management questions when working with pointers -> who is the owner? the client, the ADT, or the system?

Sorting tactics

- When sorting arrays of large items it can be more performant to maintain a separate array of indices into the original array
- using string ADT above:

```
bool operator<(const Item &a, const Item &b) {
    return strcomp(a.str, b.str) < 0;
}
```

- using a separate array (this has some confusing issues about where data is)

```
bool operator<(const Index &i, const Index &j) {
    return data[i] < data[j];
}
```

qsort

- pointer sort which takes a comparator (comparison function) as an argument

```
void qsort(array, array size, size of object, comparator)
int predicate(const void *a, const void *b) -> returns -1, 0, 1

std::qsort(a, size, sizeof *a, [](const void* a, const void* b) {
    int arg1 = *static_cast<const int*>(a);
    int arg2 = *static_cast<const int*>(b);

    if(arg1 < arg2) return -1;
    if(arg1 > arg2) return 1;
    return 0;
```

```
//  return (arg1 > arg2) - (arg1 < arg2); // possible shortcut
//  return arg1 - arg2; // erroneous shortcut (fails if INT_MIN is present)
});
```

Separate array

- the container class can have two arrays

```
data[]
sorted_data[]
```

- access:

```
data[sorted_data[k]]
```

- in place sort (in situ permutation) NOTE: this might have problems

```
template<typename Item>
void in_situ(Item data[], Index a[], int N) {
    for (int i = 0; i < N; i++) {
        Item v = data[i];
        int j;
        int k;
        for (k = i; a[k] != i; k = a[j], a[j] = j) {
            j = k;
            data[k] = data[a[k]];
        }
        data[k] = v;
        a[k] = k;
    }
}
```

- also see heapsort
- Dijkstra smoothsort (not stable)
    - https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Smoothsort

Sorting of Linked Lists

- uses same interface
    - random initialization
    - initialization from standard in
    - printing
    - sorting

- memory is managed by the list itself
- one implication that is not always apparent is that only links in nodes should be changed, leaving keys/data untouched
- insertion, selection, and bubble can be adapted but present certain challenges
- NOTE: these assume the list head is not a nullptr
- linked list selection sort:

```
node *list_selection_sort(node *list_head) {
    node dummy{0, list_head};
    node *dummy_head = &dummy;
    node *out{nullptr};

    while (dummy_head->next != nullptr) {
        node *before_max = dummy_head;
        node *tmp = before_max;
        while (tmp->next != nullptr) {
            if (tmp->next->item > before_max->next->item) {
                before_max = tmp;
            }
            tmp = tmp->next;
        }
        tmp = before_max->next; // actual node containing max item
        before_max->next = tmp->next; // remove max node from input list
        tmp->next = out; // current max points to previous max
        out = tmp; // output list head is current max
    }

    return out;
}
```

- linked list insertion sort:

```
node *list_insertion_sort(node *list_head) {
    node dummy{0, nullptr};
    node *dummy_head = &dummy;
    node *i, *j, *tmp;

    for (tmp = list_head->next; tmp != nullptr; tmp = i) {
        i = tmp->next;
        for (j = dummy_head; j->next != nullptr; j = j->next) {
            if (j->next->item > tmp->item) {
                break;
            }
        }
        tmp->next = j->next;
        j->next = tmp;
    }
```

```
    return dummy.next;
}
```

- linked list bubble sort:

```
node *list_bubble_sort(node *list_head) {
    node dummy{0, list_head};
    node *dummy_head = &dummy;
    node *i, *j, *t1, *t2;

    for (i = list_head->next; i != nullptr; i = i->next) {
        for (j = dummy_head; j->next != nullptr; j = j->next) {
            t1 = j->next;
            t2 = t1->next;
            if (t2 != nullptr && t2->item < t1->item) {
                t1->next = t2->next;
                t2->next = t1;
                j->next = t2;
            }
        }
    }

    return dummy.next;
}
```

- may sometimes use algorithms with sorted linked lists (for sets, if insertions are relatively rare, or if the list is small)

Key-Indexed Counting

- some keys have special properties such that they enable special sort techniques
- for integer keys

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

- can also use an array of counts for input arrays with duplicates

```
for (i = 0; i < n; i++) {
    b[cnt[a[i]]++] = a[i];
}
```

- key indexed counting:
    i. initialize counts to 0
    ii. get counts by incrementing `a[i] + l`
    iii. adds numbers to produce the count of the keys less than or equal to the current count

iv. use these numbers as indices into a temporary array representing the actual location within the array for l/r

v. move values from b back to a

```
void distcount(int a[], int l, int r) {
    int i, j, cnt[M];
    static int b[maxN];
    for (j = 0; j < M; j++) { cnt[j] = 0; }
    for (i = l; i <= r; i++) { cnt[a[i] + 1]++; }
    for (j = 1; j < M; j++) { cnt[j] += cnt[j - 1]; }
    for (i = l; i <= r; i++) { b[cnt[a[i]]++] = a[i]; }
    for (i = l; i <= r; i++) { a[i] = b[i - l]; }
}
```

Quicksort

- general properties
    - not stable
    - O(lg(n)) extra space
    - O(n ^ 2) time (typically O(n * lg(n)) time)
    - not adaptive
- divides into 2 partitions and sorts each independently
- the array partitions must make the following conditions hold
    - the element `a[i]` is in its final place in the array for some i
    - no element `a[l] ... a[i - l]` is greater than `a[i]`
    - no element `a[i + l] ... a[r]` is less than `a[i]`
- quicksort:

```
template<typename Item>
void quicksort(Item a[], int l, int r) {
    if (r <= l) {
        return;
    }
    int i = partition(a, l, r);
    quicksort(a, l, i - l);
    quicksort(a, i + l, r);
}
```

- partitioning:
    - begin with (arbitrary) element
    - scan from left and right exchanging elements that stop the scan until the pointers meet

```
template<typename Item>
int partition(Item a[], int l, int r) {
    int i = l - 1;
    int j = r;
    Item v = a[r];

    for (;;) {
        while (a[++i] < v) {}
        while (v < a[--j]) {
            if (j == l) {
                break;
            }
        }
        if (i >= j) {
            break;
        }
        exch(a[i], a[j]);
    }
    exch(a[i], a[r]);
    return i;
}
```

Performance of quicksort

- uses n ^ 2 / 2 comparisons in the worst case
- uses about 2 * n * ln(n)

Stack size for quicksort

- can use explicit stack and push partitions to be processed on stack
- the recursive implementation uses the system stack
- generally log(n) stack size but worst case of n
- can add safeguards to avoid worst case
- see program 7.3 for the nonrecursive implementation
- if the smaller of the two partitions is always sorted first then the stack size is never more than lg(n) for n elements
- the nonrecursive implementation processes the same partitions but in a different order than the recursive implementation

Small subfiles

- can prefer insertion sort for some M in the range 5 - 25

```
if (r - l <= M) {
    insertion(a, l, r);
}
```

- the partitioning scheme corresponds to a binary tree, with the recursive implementation traversing in preorder, nonrecusrive visits the smaller subtree first
- slightly better performance with

```
if (r - l <= M) {
    return;
}
```

- choosing an optimal size for small subfile cutoff results in roughly 10% improvement

Median-of-Three Partitioning

- one partitioning approach is to choose a random element which has pretty good odds of avoiding the worst case
- another option is to take a sample of three elements from the file and then use the median element
- using this method and the three-exchange method results in an improvement called median-of-three
- can improve the running time of quicksort by 20 to 25 percent
- further improvements are possible (median-of-five, hand coded assembly)

```
static const int M = 10;

template<class Item>
void quicksort(Item a[], int l, int r) {
    if (r - l <= M) { return; }
    exch(a[(l + r) / 2], a[r - 1]);
    compexch(a[l], a[r - 1]);
    compexch(a[l], a[r]);
    compexch(a[r - 1], a[r]);
    int i = partition(a, l + 1, r - 1);
    quicksort(a, l, i - 1);
    quicksort(a, i + 1, r);
}

template<class Item>
void hybridsort(Item a[], int l, int r) {
    quicksort(a, l, r);
    insertion(a, l, r);
}
```

Duplicate Keys

- large numbers of duplicate keys does not make quicksort perform poorly but exposes a number of improvements that can be made
- one method is to partition the keys such that the partitions are split into segments of keys less than, equal to, and greater than the partitioning element
- NOTE: this is a Dijkstra problem known as the Dutch National Flag problem
- Bentley and McIlroy's method:
  - keep keys equal to the partitioning element in the left subfile at the left end
  - keep keys equal to the partitioning element in the right subfile at the right end
  - l -> equal <- p -> less <- i ... j -> greater <- q -> equal <- v/r
  - when the pointers i & j cross it exposes the location to move the equal keys
  - NOTE: does not meet the requirement to partition the file in one pass

Strings and Vectors

- when working with string keys, can override comparator (operator <) to use strcmp < 0
- this method is fast but strcmp adds a cost -> as keys become more similar (later partitioning) the comparison takes longer
- to improve this, the compare function can modify itself in the stages of Bentley and McIlroy's partitioning method:
  - assuming characters are equal in positions 0 to d - 1, can just examine one character and perform a three way partition using keys whose d-th character is smaller, equal, and larger
- this approach generalizes to handle multidimensional sorts, i.e. sorts where keys are vectors

Selection

- one related application is finding a median
- sorting and finding the middle can be improved upon by using the quicksort partitioning
- finding the median is a special case of the generalization of finding the k-th smallest (largest) of a set of numbers
- select:

```
template <class Item>
void select(Item a[], int l, int r, int k) {
    if (r <= l) return;
    int i = partition(a, l, r);
    if (i > k) select(a, l, i-1, k);
```

```
    if (i < k) select(a, i+1, r, k);
}
```

- nonrecursive select:

```
template <class Item>
void select(Item a[], int l, int r, int k) {
    while (r > l) { i
        nt i = partition(a, l, r);
        if (i >= k) r = i-1;
        if (i <= k) l = i+1;
    }
}
```

- quicksort-based selection is linear time on average

Merging and Meergesort

- properties:
    - stable
    - O(n) extra space for arrays
    - O(lg(n)) extra space for linked lists
    - O(n * lg(n)) time
    - not adaptive
    - does not require random access (uses sequential access)
- quicksort uses selection - finding the k smallest elements/ n - k largest
- merging combines ordered files to make one larger ordered file
- selection and merging are complementary
- mergesort sorts in O(n log n), requires additional space proportional to n
- the running time is constant, i.e. non-adaptive which is not attractive in some cases
- mergesort is stable
- mergesort is the method of choice for sorting linked lists

Two-Way Merging

- many possible implementations
- two-way merging:
    - c is the target

- each iteration puts an element into c from a or b
- if a is exhausted, the element comes from b and vice versa
- assumes a and b are sorted and that c's storage does not overlap a or b

```
template <class Item>
void mergeAB(Item c[], Item a[], int N, Item b[], int M ) {
    for (int i = 0, j = 0, k = 0; k < N + M; k++) {
        if (i == N) { c[k] = b[j++]; continue; }
        if (j == M) { c[k] = a[i++]; continue; }
        c[k] = (a[i] < b[j]) ? a[i++] : b[j++];
    }
}
```

- other applications of merging:
  - data processing (database etc) batch append and resort (it is more efficient to sort the smaller batch and then merge)

Abstract In-Place Merge

- to merge two ascending files:
  - copy into auxiliary array with the 2nd in reverse
  - move left or right item with smaller key to the output
  - the larger key is the sentinel

```
template <class Item>
void merge(Item a[], int l, int m, int r) {
    int i, j;
    static Item aux[maxN];
    for (i = m+1; i > l; i--) aux[i-1] = a[i-1];
    for (j = m; j < r; j++) aux[r+m-j] = a[j+1];
    for (int k = l; k <= r; k++) {
        if (aux[j] < aux[i]) {
            a[k] = aux[j--]; else a[k] = aux[i++];
        }
    }
}
```

Top-Down Mergesort

- top-down mergesort is like a tree and no work is done until it is passed to a leaf node
- prototypical divide and conquer

```
template <class Item>
void mergesort(Item a[], int l, int r) {
    if (r <= l) return;
    int m = (r+l)/2;
```

```
    mergesort(a, l, m);
    mergesort(a, m+1, r);
    merge(a, l, m, r);
}
```

- the tree created by mergesort is only dependent on the intial input (and not the key values like quicksort)
- properties:
  - mergesort requires O(n*lg(n)) comparisons
  - extra space proportional to n
  - stable if the merge is stable
  - the additional resource requirements of mergesort are not dependent on the initial order of the input

Improvements to the Basic Algorithm

- handling small cases differently, i.e. switching to insertion for small files, provides 10 to 15 percent performance improvement
- another is to reduce the time taken to copy to the auxiliary array
  - one option is to have two versions of the array, one taking aux and a as input, the other is a to aux and have the two version call each other
  - another makes one auxiliary array to start and switches arguments between recursive calls this removes the array copy at the expense of adding tests back into the inner loop
- mergesort with no copying:

```
template <class Item>
void mergesortABr(Item a[], Item b[], int l, int r) {
    if (r-l <= 10) { insertion(a, l, r); return; }
    int m = (l+r)/2;
    mergesortABr(b, a, l, m);
    mergesortABr(b, a, m+1, r);
    mergeAB(a+l, b+l, m-l+1, b+m+1, r-m);
}

template <class Item>
void mergesortAB(Item a[], int l, int r) {
    static Item aux[maxN];
    for (int i = l; i <= r; i++) aux[i] = a[i];
    mergesortABr(a, aux, l, r);
}
```

Bottom-Up Mergesort

- all recursive algorithms have a nonrecursive corollary
- for mergesort, the recursive algorithm pre-determines the order of the merges

- bottom up mergesort:
  - start with sequence of passes doing m-by-m merges, doubling m on each pass
  - final subfile is m in m is an even multiple of m, so final merge is m-by-x where x is less than m

```
inline int min(int A, int B) { return (A < B) ? A : B; }

template <class Item>
void mergesortBU(Item a[], int l, int r) {
    for (int m = 1; m <= r-l; m = m+m) {
        for (int i = l; i <= r-m; i += m+m) {
            merge(a, i, i+m-1, min(i+m+m-1, r));
        }
    }
}
```

- for bottom-up algorithms, it is often useful to think of the algorithm as combine and conquer
- properties:
  - all merges in each pass of a bottom-up megesort involve file sizes that are a power of 2 except the final file size
  - the number of passes in a bottom-up mergesort of n elements is equal to the number of bits in the representation of n (ignoring leading 0s)
- can take each as a generalized divide-and-conquer and combine-and-conquer algorithm design
- TODO: review this to create a generalized outline of these

Performance Characteristics of Mergesort

- TODO: fill this in

Linked-list Implementations of Mergesort

- works well with mergesort
- merge:
  - uses c as an auxiliary pointer
  - merge is stable if b follows a
  - use end node of 0
  - do not use head nodes directly

```
link merge(link a, link b) {
    node dummy(0);
    link head = &dummy, c = head;
    while ((a != 0) && (b != 0)) {
        if (a->item < b->item) {
```

```
            c->next = a;
            c = a;
            a = a->next;
        } else {
            c->next = b;
            c = b;
            b = b->next;
        }
    }
    c->next = (a == 0) ? b : a;
    return head->next;
}
```

- top-down list mergesort

```
link mergesort(link c) {
    if (c == 0 || c->next == 0) return c;
    link a = c, b = c->next;
    while ((b != 0) && (b->next != 0)) {
        c = c->next; b = b->next->next;
    }
    b = c->next; c->next = 0;
    return merge(mergesort(a), mergesort(b));
}
```

- bottom-up list mergesort

```
link mergesort(link t) {
    QUEUE<link> Q(max);
    if (t == 0 || t->next == 0) return t;
    for (link u = 0; t != 0; t = u) {
        u = t->next; t->next = 0; Q.put(t);
    }
    t = Q.get();
    while (!Q.empty()) {
        Q.put(t); t = merge(Q.get(), Q.get());
    }
    return t;
}
```

Recursion Revisited

- general characteristics of divide-and-conquer
  - work is usually done before the recursive call
  - work on the large components first
- quicksort is more like conquer-and-divide
  - the non-recursive version maintains a stack because the the order is data-dependent
  - array based is not easily made stable
  - linked-list based is stable

- mergesort is standard divide-and-conquer

Radix Sorting

- properties:
    - LSD - stable, MSD - can be stable with extra work, trie - not necessarily stable
    - O(wn) performance for integers of word size w -> this changes in general where w becomes log(n) for random-access machines so time complexity for radix sort becomes O(n * log(n)) // - O(1) extra space (see discussion) // - O(n * lg(n)) time // - not really adaptive
- begins with assumption that sort can be improved by just comparing the first few elements then decomposing keys into a byte representation and comparing elements there
- radix sorting treats keys as numbers represented in a base-R number system
- for integers we work with R = 2
- for strings of characters we work with R = 128 or 256
- C++ facilitates this, but not all languages allow this and so radix sort is not very widely used
- certain architectures make use of smaller chunks of data difficult and thus slower
- two general types (MSD - most significant digit and LSD - least significant digit)

Bits, Bytes, and Words

- definitions:
    - byte: fixed-length sequence of bits
    - string: a variable-length sequence of bytes
    - word: fixed length sequence of bytes
- typical machines have 8-bit bytes and 32 or 64-bit words, but it is important to know this can change

```
const int bits_per_word = 32;
const int bits_per_byte = 8;
const int bytes_pre_word = bits_per_word / bits_per_byte; // assumes bpw is a mult
const int radix = 1 << bits_per_byte;

// extract b-th byte of binary word a
inline int digit(long a, int b) {
    return (a >> bits_per_byte * (bits_per_word - b - 1)) & (radix - 1);
}

// when using strings with chars
```

```
inline int digit(char *a, int b) {
    return a[b];
}
```

- can think of keys as digits
  - b-th digit of the radix representation of a: (a / radix ^ b) % radix
- can also think of keys as numbers between 0 and 1
  - b-th digit of the radix representation of a: (a * radix ^ b) % radix
- definition:
  - key: a radix number with digits numbered from left starting at 0

Binary Quicksort

- radix-exchange sort and binary quicksort are the same
- uses quicksort to arrange keys in a type of lexicographical ordering (start with 0 before start with 1, etc.)
- partitions keys on leading bits and sorts recursively

```
template <class Item>
void quicksort_b(Item a[], int l, int r, int d) {
    int i = l, j = r;
    if (r <= l || d > bitsword) {
        return;
    }
    while (j != i) {
        while (digit(a[i], d) == 0 && (i < j)) i++;
        while (digit(a[j], d) == 1 && (j > i)) j--;
        exch(a[i], a[j]);
    }
    if (digit(a[r], d) == 0) {
        j++;
    }
    quicksortB(a, l, j-1, d+1);
    quicksortB(a, j, r, d+1);
}

template <class Item>
void sort(Item a[], int l, int r) { quicksortB(a, l, r, 0); }
```

MSD Radix Sort

- using just 1 bit in radix quicksort amounts to treating keys as radix-2 (binary) numbers
- generalizing radix means partitioning a container (array) in R bins/buckets
- most significant digit -> start with first element in key, sort, move to next element, sort, etc

```
#define bin(A) l+count[A]
template <class Item>
void radix_msd(Item a[], int l, int r, int d) {
    int i, j, count[R+1];
    static Item aux[maxN];
    if (d > bytesword) {
        return;
    }
    if (r-l <= M) {
        insertion(a, l, r); return;
    }
    for (j = 0; j < R; j++) {
        count[j] = 0;
    }
    for (i = l; i <= r; i++) {
        count[digit(a[i], d) + 1]++;
    }
    for (j = 1; j < R; j++) {
        count[j] += count[j-1];
    }
    for (i = l; i <= r; i++) {
        aux[count[digit(a[i], d)]++] = a[i];
    }
    for (i = l; i <= r; i++) {
        a[i] = aux[i-l];
    }
    radixMSD(a, l, bin(0)-1, d+1);
    for (j = 0; j < R-1; j++) {
        radixMSD(a, bin(j), bin(j+1)-1, d+1);
    }
}
```

- can also use linked lists and keep one linked list for each bin
- for radix sort to be useful, must balance between radix size and cutoff for small files
- using R = 256 and removing the call for bin 0 is good for C-style strings
- must manage empty bins properly for applications
- the recursive call structure for msd radix sort corresponds to a multiway trie, which is a direct generalization of the trie structure for binary quicksort
- bin-span heuristic - keep track of the high and low ends of the range of nonempty bins

Three-way Radix Quicksort

- use three way partitioning on the leading byte of a key and move to next byte of only the middle subfile (keys with the leading byte equal to the partitioning element)
- amounts to using quicksort on the leading characters of the keys
- similar to a hybrid of MSD radix sort and normal quicksort

```
#define ch(A) digit(A, d)
template<typename Item>
```

```
void quicksort_x(Item a[], int l, int r, int d) {
    int i, j, k, p, q; int v;
    if (r-l <= M) { insertion(a, l, r); return; }
    v = ch(a[r]); i = l-1; j = r; p = l-1; q = r;
    while (i < j) {
        while (ch(a[++i]) < v) ;
        while (v < ch(a[--j])) if (j == l) break;
        if (i > j) break;
        exch(a[i], a[j]);
        if (ch(a[i])==v) { p++; exch(a[p], a[i]); }
        if (v==ch(a[j])) { q--; exch(a[j], a[q]); }
    }
    if (p == q) {
        if (v != '\0') quicksortX(a, l, r, d+1);
        return;
    }
    if (ch(a[i]) < v) {
        i++;
    }
    for (k = l; k <= p; k++, j--) {
        exch(a[k], a[j]);
    }
    for (k = r; k >= q; k--, i++) {
        exch(a[k], a[i]);
    }
    quicksort_x(a, l, j, d);
    if ((i == r) && (ch(a[i]) == v)) {
        i++;
    }
    if (v != '\0') {
        quicksortX(a, j+1, i-1, d+1);
    }
    quicksortX(a, i, r, d);
}
```

- can gain performance by using MSD radix sort for large files and three-way radix quicksort with a smaller radix for smaller files
- three-way radix quicksort is also good for vectors (mathematical and C++)

LSD Radix Quicksort

- examining bytes from right to left, start with final letter using key-indexed counting
- requires a stable underlying sort

```
template <class Item>
void radix_lsd(Item a[], int l, int r) {
    static Item aux[maxN];
    for (int d = bytesword-1; d >= 0; d--) {
        int i, j, count[R+1];
        for (j = 0; j < R; j++) {
            count[j] = 0;
        }
```

```
        for (i = l; i <= r; i++) {
            count[digit(a[i], d) + 1]++;
        }
        for (j = 1; j < R; j++) {
            count[j] += count[j-1];
        }
        for (i = l; i <= r; i++) {
            aux[count[digit(a[i], d)]++] = a[i];
        }
        for (i = l; i <= r; i++) {
            a[i] = aux[i-l];
        }
    }
}
```

- LSD radix sorting is the method used by old computer card sorting machines

Performance Characteristics of Radix Sorts

- sort of N records with w bytes is proportional to nw
- for long keys and short bytes the running time is n * lg(n)
- for shorter keys and longer bytes the running time is comparable to N
- properties:
  - worst case for radix sorting is to examine all of the bytes
  - binary quicksort examines about n * lg(n) bits on average when sorting keys composed of random bits
  - MSD radix sort with radix R on a file of size N requires at least 2 * n + 2 * R steps
  - if the radix is always less than the file size, the number of steps taken by MSD radix sort is within a small constant factor of N * log sub R (N)
  - three-way radix quicksort uses 2*N ln(N) byte comparisons
  - LSD radix sort can sort N records with w-bit keys in w/lg(R) passes, using extra space for R counters (and a buffer for rearranging the file)

Sublinear-Time Sorts

- radix sort can be sublinear in certain applications
- using two passes, can apply insertion sort afterwards
- LSD radix sort is more widely used

Special-Purpose Sorting Methods

Batcher's Odd-Even Mergesort

- nonadaptive sort - only dependent on the number of elements and not their input order or values
- perfect shuffle:
  - split an element in half and then interleave elements from each (alternate)
- perfect unshuffle:
  - odd elements go in the second half and even numbered elements go in the first half

```
template <class Item>
void shuffle(Item a[], int l, int r) {
    int i, j, m = (l+r)/2;
    static Item aux[maxN];
    for (i = l, j = 0; i <= r; i+=2, j++) {
        aux[i] = a[l+j]; aux[i+1] = a[m+1+j];
    }
    for (i = l; i <= r; i++) a[i] = aux[i];
}

template <class Item>
void unshuffle(Item a[], int l, int r) {
    int i, j, m = (l+r)/2;
    static Item aux[maxN];
    for (i = l, j = 0; i <= r; i+=2, j++) {
        aux[l+j] = a[i]; aux[m+1+j] = a[i+1];
    }
    for (i = l; i <= r; i++) a[i] = aux[i];
}
```

- Batcher's sort is mergesort but uses Batcher's odd-even merge
- assumes the number of elements to sort is a power of 2
- 1-by-1 merge does a simple compare/exchange, n-by-n does an unshuffle and then sorts recursively
- recursive Batcher's merge

```
template <class Item>
void merge(Item a[], int l, int m, int r) {
    if (r == l+1) compexch(a[l], a[r]);
    if (r < l+2) return;
    unshuffle(a, l, r);
    merge(a, l, (l+m)/2, m);
    merge(a, m+1, (m+1+r)/2, r);
    shuffle(a, l, r);
    for (int i = l+1; i < r; i+=2) {
```

```
        compexch(a[i], a[i+1]);
    }
}
```

- properties:
  - if a nonadaptive program produces sorted output when the inputs are all either 0 or 1, than it does so when the inputs are arbitrary keys (0 - 1 principle)
  - it is a valid merging method

Sorting Networks

- a sorting network is a machine that can access data only through compare/exchange operations
  - atomic compare-exchange modules or comparators wired together to create a generic sorting module
  - move from left to right and keys are exchanged by the comparator at each node
- useful for minimizing comparators, direct circuit realizations, and as a model for parallel computation
- given a network, it is possible to classify the comparators as a sequence of parallel stages Batcher's odd-even merge (nonrecursive)

```
template<typename Item>
void merge(Item a[], int l, int m, int r) {
    int N = r-l+1;  // assuming N/2 is m-l+1
    for (int k = N/2; k > 0; k /= 2) {
        for (int j = k % (N/2); j+k < N; j += k+k) {
            for (int i = 0; i < k; i++) {
                compexch(a[l+j+i], a[l+j+i+k]);
            }
        }
    }
}
```

- Batcher's odd-even sort (nonrecursive)

```
template <class Item>
void batchersort(Item a[], int l, int r) {
    int N = r-l+1;
    for (int p = 1; p < N; p += p) {
        for (int k = p; k > 0; k /= 2) {
            for (int j = k%p; j+k < N; j += (k+k)) {
                for (int i = 0; i < N-j-k; i++) {
                    if ((j+i)/(p+p) == (j+i+k)/(p+p)) {
                        compexch(a[l+j+i], a[l+j+i+k]);
                    }
                }
            }
        }
```

```
    }
}
```

- Bose-Nelson problem -> finding networks with as few comparators as possible
- AKS networks have better comparator values but not practical
- split-interleave merge
  - split and interleave, performing compare exchanges on each adjacent element
- applications in transposing 2^n X 2^n square matrix and FFT (solved with cycling perfect shuffling machines)

External Sorting

- high read/write cost for accessing sort input from external devices
- often improved by block sequential access
- assumption is each file is read from a different device
- can solve in steps:
  - first create a program to efficiently copy a file
  - then, reverse the file
  - apply lessons to sorting
- removing a single pass can improve performance
- assumptions:
  - n records on an external device
  - space in main memory to hold m records
  - 2 * P external devices to use during sort
- sort-merge strategies
  - three-way balanced merge
    - 2 * P external devices and memory for M records, a sort-merge based on a P-way balanced merge takes about 1 + log sub P (N / M)

Sort-Merge Implementations

- improvements:
  - replacement selection - write to priority queue, if new element is smaller than last, mark it as the start of the next block
  - polyphase merging - distribute blocks generated using replacement selection unevenly
- properties:
  - for random keys, the runs produced by replacement selection are about twice the size of the heap used
  - with 3 external devices and internal memory sufficient to hold M records, a sort-merge that is based on replacement selection followed by two-way polyphase merge takes about 1 + log@(N / 2M)@

Parallel Sort-Merge

- uses parallel processing for sort-merge operations
- assumptions:
  - N records
  - P processors with a max capacity of N / P records
- merging comparator:
  - input: two sorted files of size M
  - output: two sorted files, one containing the M smallest of the 2M inputs and the other containing the M largest of the 2M inputs
- properties:
  - sort a file of size N by dividing it into N / M blocks of size M, sorting each, then sort with a sorting network with merging comparators
  - block sorting on P processors, using Batcher's sort with merging comparators, can sort N records in about (lg(P))^2/2 parallel steps
- most of these extra sorts come from Knuth