5 Sorting

A collection of items satisfying a **weak order relation** "<" is sorted. Beware that "≤" isn't weak. For n items $\exists$ $n!$ orders, and k binary comparisons decide between $\geq 2^k$ of them, so to sort using only comparisons need $k = \Theta(n\lg(n))$. Sorting is **stable** if equal items keep their original relative order. Using item location as a **secondary key** ensures stability but needs more memory and is clumsy for the caller. Item sorting analyses usually assume O(1) comparisons. For expensive-to-copy items, for efficiency sort an array of pointers to them.

5.1 Insertion Sort

For small arrays insertion sort is stable and the fastest, so it makes a good helper function for some smarter algorithms. It mimics sorting a hand of cards. Given a sorted array, initially with the first item, iteratively insert the next one into the correct place.

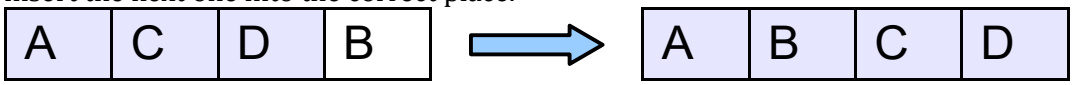

Figure 5.1: Next unsorted item logic of insertion sort

```
template<typename ITEM, typename COMPARATOR>
void insertionSort(ITEM* vector, int left, int right, COMPARATOR const& c)
{//allow more general left != 0
    for(int i = left + 1; i <= right; ++i)
    {
        ITEM e = vector[i];
        int j = i;
        for(;j > left && c(e, vector[j - 1]); --j)
            vector[j] = vector[j - 1];
        vector[j] = e;
    }
}
```

The runtime is O(n^2) with very low constant factors, and O(the number of reversed pairs called **inversions**) = O(n) for almost sorted input.

5.2 Quicksort

If don't need stability, quicksort is the fastest. The basic version:

1. **Pick a pivot item**
2. **Partition the array so that items ≤/≥ the pivot are on the left/right**
3. **Sort the two halves recursively**

The order of items within each subarray after a partition doesn't matter:

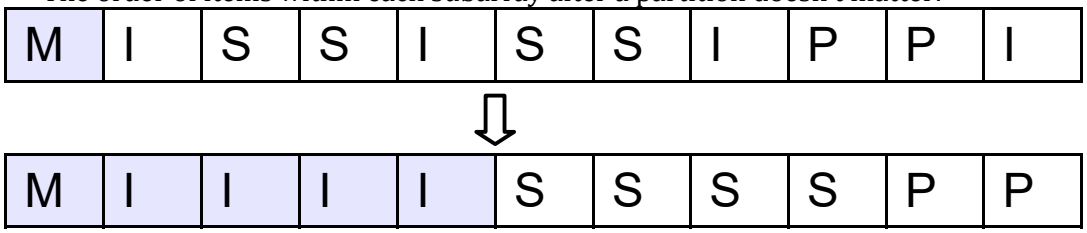

Figure 5.2: A possible result of quicksort pivoting with pivot 'M'

The most practical pivot is the **median of three** random items because:

- Deterministic picks may give O(n^2) runtime
- A single random pivot is slightly slower
- Using five or more is negligibly faster but more complex

```
template<typename ITEM, typename COMPARATOR>
```

```
int pickPivot(ITEM* vector, int left, int right, COMPARATOR c)
{
    int i = GlobalRNG().inRange(left, right), j =
        GlobalRNG().inRange(left, right), k = GlobalRNG().inRange(left, right);
    if(c(vector[j], vector[i])) swap(i, j);
    //i <= j, decide where k goes
    return c(vector[k], vector[i]) ? i : c(vector[k], vector[j]) ? k : j;
}
```

Partitioning actually should group items into < pivot, = pivot, and > pivot, in that order (Sedgewick 1999). Note that unlike for a complete sort, the items in "<" and ">" sections will be in an arbitrary order, which is to be taken care of by further partitioning. As an intermediate step, move equal items to the sides:

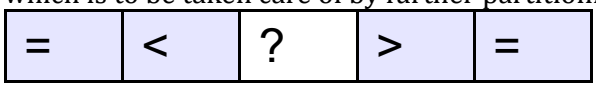

Figure 5.3: Quicksort partitioning work strategy

Use left and right pointers to scan the array from both directions at the same time. If a scanned item doesn't belong to corresponding "<" or ">" section, it's marked for swapping. At every iteration the pointers stop at such items. The process stops when the pointers cross. Then the side "=" section are swapped back to the middle.

```
template<typename ITEM, typename COMPARATOR> void partition3(ITEM* vector,
    int left, int right, int& i, int& j, COMPARATOR const& c)
{//i/j are the current left/right pointers
    ITEM p = vector[pickPivot(vector, left, right, c)];
    int lastLeftEqual = i = left - 1, firstRightEqual = j = right + 1;
    for(;;)//the pivot is the sentinel for the first pass
    {//after one swap swapped items act as sentinels
        while(c(vector[++i], p));
        while(c(p, vector[--j]));
        if(i >= j) break;//pointers crossed
        swap(vector[i], vector[j]);//both pointers found swappable items
        //swap equal items to the sides
        if(c.isEqual(vector[i], p))//i to the left
            swap(vector[++lastLeftEqual], vector[i]);
        if(c.isEqual(vector[j], p))//j to the right
            swap(vector[--firstRightEqual], vector[j]);
    }
    //invariant: i == j if they stop at an item = pivot
    //and this can happen at both left and right item
    //or they cross over and i = j + 1
    if(i == j){++i; --j;}//don't touch pivot in the middle
    //swap side items to the middle; left with "<" section and right with ">"
    for(int k = left; k <= lastLeftEqual; ++k) swap(vector[k], vector[j--]);
    for(int k = right; k >= firstRightEqual; --k)
        swap(vector[k], vector[i++]);
}
```

Either *i* or *j* may be out of bounds. The postcondition:

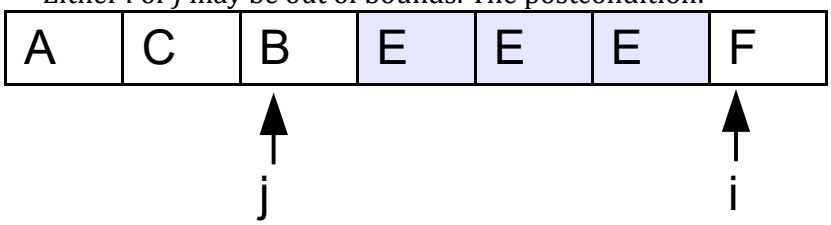

Figure 5.4: Quicksort pointers after 3-partition

The slightly less complicated basic "≤/≥" partitioning pays no special attention to equal items and is obtained by just removing the code for swapping to the sides and back. The *i-j* invariant remains the same, but it defines the postcondition. The three-partitioning also applies to vector sorting and is faster for array with many equal items. With few extra items its equality checks and swaps are also few.

Optimizations for the main algorithm:

- Sorting smaller subarrays first ensures O(lg(n)) extra memory, which practically guarantees that the recursion stack won't run out.
- Use insertion sort for small subarrays of size 16 (per studies 5–25 works). Due to caching, recursing to insertion sort is faster than a single insertion sort over the whole array in the end, despite using more instructions (Mehlhorn & Sanders 2008).
- Remove the tail recursion. Removing the other one complicates the algorithm with little gain.

```
template<typename ITEM, typename COMPARATOR>
void quickSort(ITEM* vector, int left, int right, COMPARATOR const& c)
{
    while(right - left > 16)
    {
        int i, j;
        partition3(vector, left, right, i, j, c);
        if(j - left < right - i)//pick smaller
        {
            quickSort(vector, left, j, c);
            left = i;
        }
        else
        {
            quickSort(vector, i, right, c);
            right = j;
        }
    }
    insertionSort(vector, left, right, c);
}
template<typename ITEM> void quickSort(ITEM* vector, int n)
    {quickSort(vector, 0, n - 1, DefaultComparator<ITEM>());}
```

With basic partitioning E[the runtime] = O($n$lg(n)). Proof: Suppose the pivot is random, and all items are unique. Let X_{ij} be the number of times that the item at i was compared to the item at j in the sorted array with $j > i$. E[X_{ij}] = Pr(i or j was a pivot) because i and j were compared at most once and only if one of them was a pivot in a subarray containing the other. Else, if an item at $> j$ or $< i$ was a pivot, i and j go into the same subarray, else into separate ones. Because $\exists\ j - i + 1$ separating pivots, Pr(i or j was a pivot) = $\frac{2}{j-i+1}$, and the E[the total number of comparisons] = $E\left(\sum_{0\le i<n}\sum_{i+1\le j<n}X_{ij}\right) < 2\sum_{0\le i<n}\sum_{1\le k<n}\frac{1}{k} < 2n\lg(n)$ (Cormen et al. 2009). □

The unlikely worst case is O(n^2) if bad pivots are picked every time. The same analysis extends to the three-partitioning because the only difference is in the equality comparisons, which are a non-dominant operation.

5.3 Mergesort

Mergesort is the most efficient stable sort:

1. **Split the array into equal halves**
2. **Mergesort each recursively**
3. **Merge the halves in O(n) time**

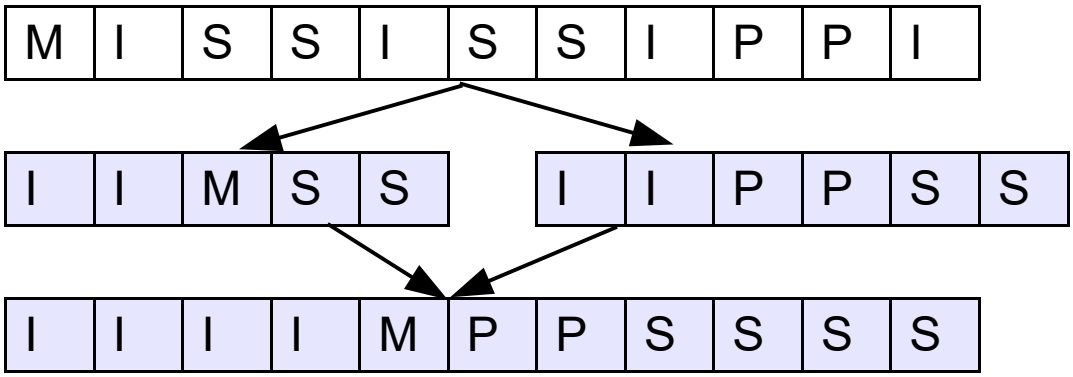

Figure 5.5: Mergesort splitting and merging

Optimizations for the main algorithm:

- Alternate the data and the temporary storage arrays to avoid unnecessary copies
- Use insertion sort for arrays of size ≤ as for quicksort

Merging is most of the work. It iteratively moves the smallest leftmost item of both arrays to the result array. The rightmost index of the left array is `middle`. The recursive call and the merge assume that for [*left*, *right*] the items are housed in the temporary storage array, so that the original [*left*, *right*] slice is overwritten with the sorted result.

```
template<typename ITEM, typename COMPARATOR> void merge(ITEM* vector,
    int left, int middle, int right, COMPARATOR const& c, ITEM* storage)
{//i for the left half, j for the right, merge until fill up vector
    for(int i = left, j = middle + 1; left <= right; ++left)
    {//either i or j can get out of bounds
        bool useRight = i > middle || (j <= right &&
            c(storage[j], storage[i]));
        vector[left] = storage[(useRight ? j : i)++];
    }
}
template<typename ITEM, typename COMPARATOR> void mergeSortHelper(
    ITEM* vector, int left, int right, COMPARATOR const& c, ITEM* storage)
{
    if(right - left > 16)
    {//sort storage using vector as storage, then merge into vector
        int middle = (right + left)/2;
        mergeSortHelper(storage, left, middle, c, vector);
        mergeSortHelper(storage, middle + 1, right, c, vector);
        merge(vector, left, middle, right, c, storage);
    }
    else insertionSort(vector, left, right, c);
}
template<typename ITEM, typename COMPARATOR>
void mergeSort(ITEM* vector, int n, COMPARATOR const& c)
{//copy vector to storage first
    if(n <= 1) return;
    Vector<ITEM> storage(n, vector[0]);//reserve space for n with 1st item
    for(int i = 1; i < n; ++i) storage[i] = vector[i];
    mergeSortHelper(vector, 0, n - 1, c, storage.getArray());
}
```

The runtime satisties $R(n) = O(n) + 2R(n/2)$, so by the master theorem $R(n) = O(n\lg(n))$.

5.4 Integer Sorting

Can sort integers in $O(n)$ time by not using "<". For integers mod N, **counting sort** counts how many times each occurs and creates a sorted array from the counts in $O(n + N)$ time. It's stable.

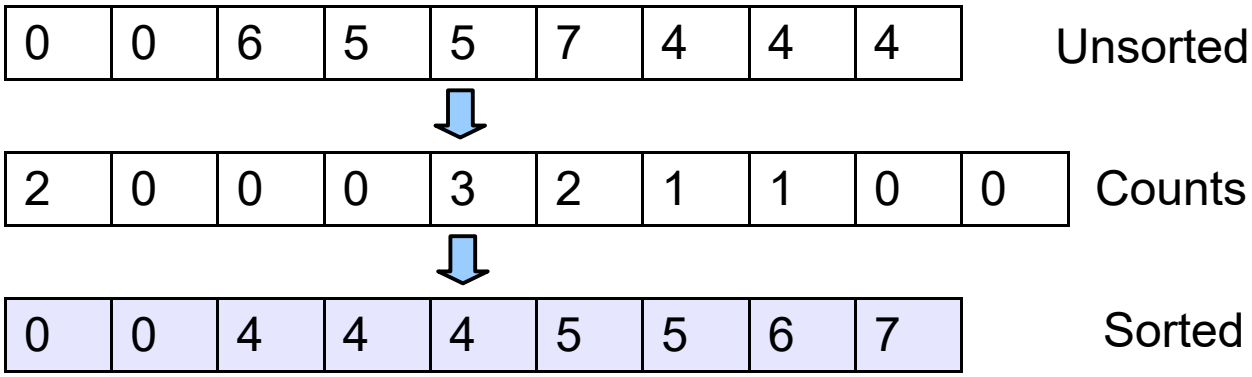

Figure 5.6: Counting sort counting and sorted array creation for N = 10

```
void countingSort(int* vector, int n, int N)
{
    Vector<int> counter(N, 0);
    for(int i = 0; i < n; ++i) ++counter[vector[i]];
    for(int i = 0, index = 0; i < N; ++i)
        while(counter[i]-- > 0) vector[index++] = i;
}
```

For items with integer-mod-*N* keys, **key-indexed counting sort** (**KSort**) works similarity but needs temporary storage because can't create items from counts. Also work with cumulative counts because they tell how many smaller-value items precede one with the particular key. Nothing preceeds the smallest item, so create a sentinel for 0. As items are placed, precedence counts are incremented to move the next location. So KSort:

1. **Counts how many have a particular key**
2. **Uses the cumulative counts to create a temporary sorted array**
3. **Copies it into the original**

The implementation needs a functor `ORDERED_HASH` that extracts items' keys—e.g., it can get byte 0 from an integer.

```
template<typename ITEM, typename ORDERED_HASH> void KSort(ITEM* a, int n,
    int N, ORDERED_HASH const& h)
{
    ITEM* temp = rawMemory<ITEM>(n);
    Vector<int> prec(N + 1, 0);
    for(int i = 0; i < n; ++i) ++prec[h(a[i]) + 1];
    for(int i = 0; i < N; ++i) prec[i + 1] += prec[i];//accumulate counts
    //rearrange items
    for(int i = 0; i < n; ++i) new(&temp[prec[h(a[i])]++])ITEM(a[i]);
    for(int i = 0; i < n; ++i) a[i] = temp[i];
    rawDelete(temp);
}
```

It's stable and takes O(*n* + *N*) time.

5.5 Vector Sorting

Sorting *n* vectors of size *k* as items takes O(*kn*lg(*n*)) time. For quicksort on *n* vectors of length ∞, E[the runtime] = O(*n*lg(*n*)^2) (Vallee et al. 2009). This is intuitive because for two random sequences in a collection of *n* with items from an alphabet of size *A*, E[**lcp** (**least common prefix**)] = log$_A$(*n*), requiring that time for a comparison.

Multikey quicksort three-partitions on the first letter and recurses on each subarray, going to the next letter for the equal part:

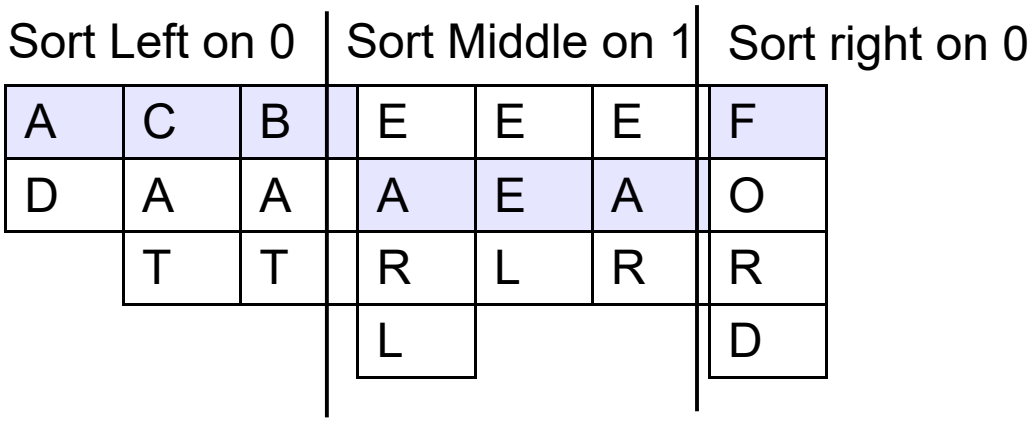

Figure 5.7: Multikey quicksort recursive 3-partitioning on consecutive characters of words

The user-provided comparator, such as the one for vectors below, keeps track of current depth, starting with 0, which allows sorting arbitrary tuples.

```
template<typename VECTOR> struct VectorComparator
{
    mutable int depth;
    VectorComparator(): depth(0){}
    bool operator()(VECTOR const& lhs, VECTOR const& rhs)const
    {
        return depth < lhs.length() ?
            depth < rhs.length() && lhs[depth] < rhs[depth] :
            depth < rhs.length();
    }
    bool isEqual(VECTOR const& lhs, VECTOR const& rhs)const
    {
        return depth < lhs.length() ?
            depth < rhs.length() && lhs[depth] == rhs[depth] :
            depth >= rhs.length();
    }
};
```

Remove the recursion to not run out of stack for long vectors with high lcp. The depth parameter allows computing suffix arrays (see the "String Algorithms" chapter). The algorithm is driven from a stack that contains a set of intervals to process along with their depth. Start it with (*left*, *right*, 0). Note that no longer:

- Use insertion sort for small subarrays due to its inability to work efficient with vectors
- Process smaller subarray first—the equal is the longest in the worst case, and no longer worry about the memory use of recursion—so just process the middle piece last

```
template<typename VECTOR, typename COMPARATOR> void multikeyQuicksortNR(
    VECTOR* vector, int left, int right, COMPARATOR c,
    int maxDepth = numeric_limits<int>::max())
{
    Stack<int> stack;
    stack.push(left);
    stack.push(right);
    stack.push(0);
    while(!stack.isEmpty())
    {
        c.depth = stack.pop();
        right = stack.pop();
        left = stack.pop();
        if(right - left > 0 && c.depth < maxDepth)
        {
            int i, j;
            partition3(vector, left, right, i, j, comparator);
            //left
            stack.push(left);
```

```
                stack.push(j);
                stack.push(c.depth);
                //right
                stack.push(i);
                stack.push(right);
                stack.push(c.depth);
                //middle
                stack.push(j + 1);
                stack.push(i - 1);
                stack.push(c.depth + 1);
            }
        }
}
```

E[the runtime] = O($n$lg(n)), and the expected runtime with respect to the length is the optimal O(n(the length + lg(n))) (Sedgewick 1999). The unlikely worst case is O(n(length + n)). For arrays of such items must sort pointers or permutations (not implemented here for the vector comparator for simplicity) to avoid copying because otherwise vector sorting gains nothing.

If items are vectors of small integers of fixed length k, **LSD sort** is stable and the most efficient. It sorts k times using vector[$k - i$] as the key to KSort in pass i, with the overall runtime O(nk). This works because KSort is stable. Because the use case is so specific, no general code is provided, and it's up to the caller to setup the KSort calls—e.g., see suffix array construction in the "String Algorithms" chapter.

5.6 Permutation Sort

To sort array a according to a permutation p defined by an array of sorted indices, can copy the items to a temporary array and populate the original from it according to p.

But can avoid the temporary because any permutation is composed of independent subpermutations (i.e., a **product of disjoint cycles** in abstract algebra terminology). Think of $f = p[i]$ as specifying from where to take the item for position i. A loop over the array gets all cycles in O(n) time. Have start of unprocessed cycle if $p[i] \neq i$.

1. **Remember the first item in the cycle, and set *to* = its index *i***
2. **While *p*[*to*] ≠ *to***
3. ** *from* = *p*[*to*]**
4. ** *a*[*to*] = *a*[*from*]**
5. ** Mark to processed with *p*[*to*] = *to***
6. ** *to* = *from***
7. **End cycle with *a*[*to*] = the first item**

E.g., consider the permutation 3210 applied to *abcd*. Start with 0. Remember $v[0] = a$, from position $p[0] = 3$ take d, and put it into position 0. Check $p[3] = 0$ to discover the end of cycle due to $p[0] = 0$, and put stored a into position 3. Move to position 1. The same logic swaps b and c. After this, all positions are marked identity, and moving to 2 and 3 changes nothing.

```
template<typename ITEM> void permutationSort(ITEM* a, int* permutation, int n)
{
    for(int i = 0; i < n; ++i) if(permutation[i] != i)
        {//start cycle
            ITEM temp = a[i];
            int to = i;
            while(permutation[to] != to)
            {
                a[to] = a[permutation[to]];
                swap(permutation[to], to);//mark to processed and advance cycle
            }
            a[to] = temp;//complete cycle
        }
}
```

5.7 Selection

Want to arrange array items so that the specified item is in the correct place, e.g., to find the median. **Quickselect** is like quicksort but doesn't sort the subarray that can't contain the item. So it avoids one recursive call, and can be implemented iteratively.

```
template<typename ITEM, typename COMPARATOR> ITEM quickSelect(ITEM* vector,
    int left, int right, int k, COMPARATOR c)
{
    assert(k >= left && k <= right);
    for(int i, j; left < right;)
    {
        partition3(vector, left, right, i, j, c);
        if(k >= i) left = i;
        else if(k <= j) right = j;
        else break;
    }
    return vector[k];
}
```

E[the runtime] = O(n). The unlikely worst case is O(n^2). For vectors somewhat unintuitively E[the runtime] = O(n) (Vallee et al. 2009), but can extend multikey quicksort to **multikey quickselect**. Same for multiple select (covered later in the chapter).

```
template<typename VECTOR, typename COMPARATOR> void multikeyQuickselect(
    VECTOR* vector, int left, int right, int k, COMPARATOR c)
{
    assert(k >= left && k <= right);
    for(int d = 0, i, j; right - left >= 1;)
    {
        partition3(vector, left, right, i, j, c);
        if(k <= j) right = j;
        else if (k < i)//equal case j < k < i
        {
            left = j + 1;
            right = i - 1;
            ++c.depth;
        }
        else left = i;
    }
}
```

5.8 Multiple Selection

To sort only the first k items, an optimal O(n + $k$lg(k)) solution is to run quicksort(0, k − 1) on the result of quickselect(k).

A more general problem is to output an array with only the specified items in correct places. Specify them with a Boolean array, and have quicksort not recurse into subarrays without any selected items. E.g., can compute statistical quantiles this way.

```
template<typename ITEM, typename COMPARATOR> void multipleQuickSelect(ITEM*
    vector, bool* selected, int left, int right, COMPARATOR c)
{
    while(right - left > 16)
    {
        int i, j;
        for(i = left; i <= right && !selected[i]; ++i);
        if(i == right + 1) return;//none are selected
        partition3(vector, left, right, i, j, c);
        if(j - left < right - i)//smaller first
        {
            multipleQuickSelect(vector, selected, left, j, c);
```

```
            left = i;
        }
        else
        {
            multipleQuickSelect(vector, selected, i, right, c);
            right = j;
        }
    }
    insertionSort(vector, left, right, c);
}
```

∀ selection E[the runtime] is optimal, but depends on the number and the positions of the specified items (Kaligosi et al. 2005). The unlikely worst case is $O(n^2)$.

5.9 Searching

Sequential search is the fastest for few items despite the $O(n)$ runtime and the only choice if items aren't sorted. For sorted data **binary search** is worst-case optimal, taking $O(\lg(n))$ time. It starts in the middle, and if query ≠ item, goes left if query < item and right otherwise.

```
template<typename ITEM, typename COMPARATOR> int binarySearch(ITEM const*
    vector, int left, int right, ITEM const& key, COMPARATOR c)
{
    while(left <= right)
    {
        int middle = (left + right)/2;
        if(c.isEqual(key, vector[middle])) return middle;
        c(key, vector[middle]) ? right = middle - 1 : left = middle + 1;
    }
    return -1;//not found
}
```

Before using binary search need prior knowledge of sorting. It's easy to check in $O(n)$ time:

```
template<typename ITEM, typename COMPARATOR> bool isSorted(ITEM const*
    vector, int left, int right, COMPARATOR const& c)
{
    for(int i = left + 1; i <= right; ++i)
        if(c(vector[i], vector[i - 1])) return false;
    return true;
}
```

Exponential search is useful when the "array" upper bound is unknown. Assume that it's 1, then 2, 4, 8, etc., and, after it's found, do binary search between bound/2 and bound. Arrays know their bounds, but it's very useful if a function implicitly represents the search range. E.g., can guess a positive number that someone thinks of but discloses only comparison results of it to other numbers. The runtime is O(lg(the upper bound)).

5.10 Comments

Some slower $O(n^2)$ sorts aren't useful:

- **Selection sort**—swap the minimum item with the first item, then repeat this for the rest of the array—makes the minimal possible number of item moves
- **Bubble sort**—exchange adjacent items until every item is in correct order—how people in a group sort themselves by height

Three-partitioning is a solution to the Dutch national flag problem, which is implicitly defined by it. The classic algorithm for it by Dijkstra uses fewer instructions and is a bit simpler but has higher constant factors with few equal items, which is often the case for sorting (Sedgewick 1999). Must be very careful with partitioning code because it's easy to get it wrong, particularly with sentinels for the `while` loops.

An interesting idea is using several pivots, resulting in particular in **dual-pivot quicksort**. It's slightly faster than regular quicksort but needs twice more code and is more complicated. The analysis is still ongoing, but currently the conclusion is that have fewer cache misses, which more than compensates for more

instructions (Kushagra et al. 2013).

The STL uses quicksort with deterministic median-of-three pivot but switches to a slower, safer in-place heapsort (see the "Priority Queues" chapter) on reaching a high enough depth. Though this strategy ensures O($n$lg(n)) runtime, it too benefits from using random pivots. Due to E[the runtime] guarantees, the switch seems unnecessary and doesn't generalize to other situations such as vector sorting. **Shellsort** is suboptimal but empirically slightly faster than heapsort (Sedgewick 1999); despite that it has no use case.

An interesting search algorithm for sorted numeric items is **interpolation search** (Wikipedia 2015). It's like binary search but, instead of using the average index, uses the index based on the item values. E[the runtime for uniformly distributed items] = O(ln(ln(n)), but such use case is very limited, and any gain over binary search is negligible.

To verify that an array is sorted can also do several binary searches for random elements. I haven't been able to find a reference for this with a probabilistic correctness guarantee.

An interesting problem is sorting a linked list. Because a list doesn't support random access, the only goal is traversing in sorted order. Can adapt mergesort to sort without using extra memory (Roura 1999).

5.11 References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

Kaligosi, K., Mehlhorn, K., Munro, J. I., & Sanders, P. (2005). Towards optimal multiple selection. In *Automata, Languages, and Programming* (pp. 103–114). Springer.

Kushagra, S., López-Ortiz, A., Munro, J. I., & Qiao, A. (2013). Multi-pivot Quicksort: theory and experiments. In *Proc. 16th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM.

Mehlhorn, K., & Sanders, P. (2008). *Algorithms and Data Structures: The Basic Toolbox*. Springer.

Roura, S. (1999). Improving mergesort for linked lists. In *Algorithms-ESA'99* (pp. 267–276). Springer.

Sedgewick, R. (1999). *Algorithms in C++, Parts 1–4*. Addison-Wesley.

Vallée, B., Clément, J., Fill, J. A., & Flajolet, P. (2009). The number of symbol comparisons in Quicksort and Quickselect. In *Automata, Languages, and Programming* (pp. 750–763). Springer.

Wikipedia (2015). Interpolation search. https://en.wikipedia.org/wiki/Interpolation_search. Accessed November 3, 2015.