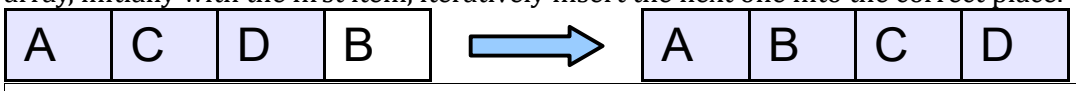
# 5 Sorting

A collection of items satisfying a **weak order relation** "<" is sorted. Don't use "≤" because it's not weak. For $n$ items, ∃ $n!$ orders, and $k$ binary comparisons decide between at most $2^k$ of them, so to sort $k = \Theta(n\lg(n))$. Sorting is **stable** if equal items keep their original relative order. Using item location as **secondary key** ensures stability but is slow. Item sorting analysis assumes O(1) time comparisons. For expensive-to-copy items, sort an array of pointers to them.

## 5.1 Insertion Sort

For small arrays, insertion sort is stable and the fastest. 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.




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

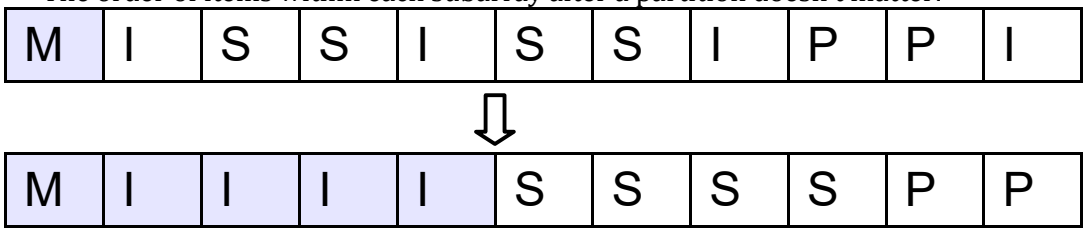
The runtime is O($n^2$), with very low constant factors, and O(the number of reversed pairs called **inversions**) = O($n$) for almost sorted input.

## 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:



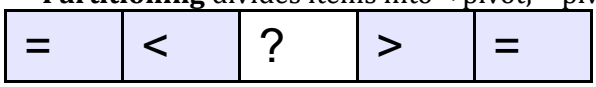

The most practical pivot is the **median of three** random items:

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

```
template<typename ITEM, typename COMPARATOR>
int pickPivot(ITEM* vector, int left, int right, COMPARATOR comparator)
{
    int i = GlobalRNG.inRange(left, right), j =
        GlobalRNG.inRange(left, right), k = GlobalRNG.inRange(left, right);
```

```
    if(comparator.isLess(vector[j], vector[i])) swap(i, j);
    //i <= j, decide where k goes
    return comparator.isLess(vector[k], vector[i]) ?
        i : comparator.isLess(vector[k], vector[j]) ? k : j;
}
```

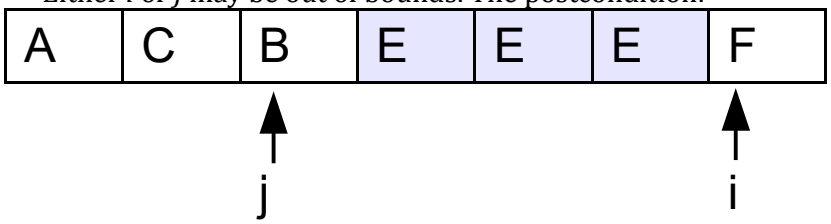
**Partitioning** divides items into < pivot, = pivot, and > pivot, moving equal items to the sides:




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. The process stops when the pointers cross.

```
template<typename ITEM, typename COMPARATOR> void partition3(ITEM* vector,
    int left, int right, int& i, int& j, COMPARATOR comparator)
{
    ITEM p = vector[pickPivot(vector, left, right, comparator)];
    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(comparator.isLess(vector[++i], p));
        while(comparator.isLess(p, vector[--j]));
        if(i >= j) break;
        swap(vector[i], vector[j]);
        //swap equal items to the sides
        if(comparator.isEqual(vector[i], p))
            swap(vector[++lastLeftEqual], vector[i]);
        if(comparator.isEqual(vector[j], p))
            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;}
    //swap side items to the middle
    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:




The extra work and complexity are small relative to the basic partitioning (that pays no special attention to equal items). Also, it's faster for many equal items and used for vector sorting. Optimizations:

- 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 5–25. 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.
- Remove the tail recursion. Removing the other one complicates the algorithm.

```
template<typename ITEM> void quickSort(ITEM* vector, int left, int right)
    {quickSort(vector, left, right, DefaultComparator<ITEM>());}
template<typename ITEM, typename COMPARATOR>
void quickSort(ITEM* vector, int left, int right, COMPARATOR comparator)
{
```

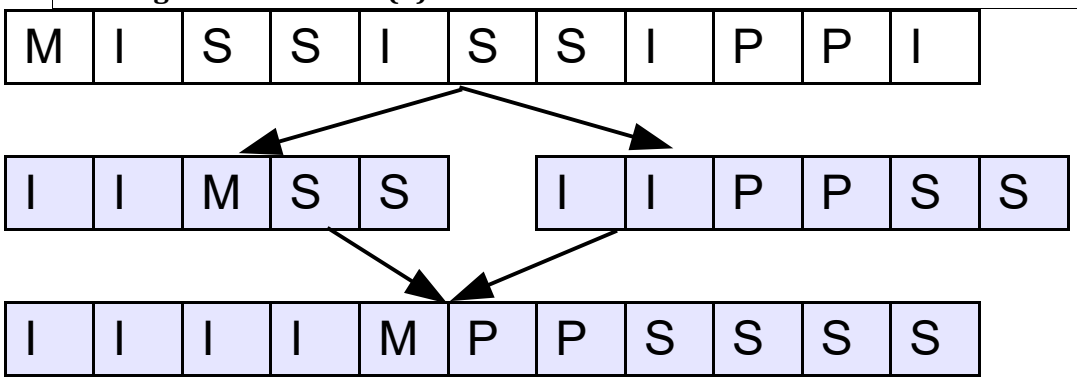
```
    while(right - left > 16)
    {
        int i, j;
        partition3(vector, left, right, i, j, comparator);
        if(j - left < right - i)//smaller first
        {
            quickSort(vector, left, j, comparator);
            left = i;
        }
        else
        {
            quickSort(vector, i, right, comparator);
            right = j;
        }
    }
    insertionSort(vector, left, right, comparator);
}
```

E[the runtime] = O($n$lg($n$)). 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) = $2/(j - i + 1)$, and the E[the total number of comparisons] = $E(\sum_{0 \le i < n}\sum_{i+1 \le j < n} X_{ij}) < 2\sum_{0 \le i < n}\sum_{1 \le k < n} 1/k < 2n\lg(n)$. The unlikely worst case is O($n^2$).

## 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**




Optimizations:

- Alternate the data and the temporary storage arrays to avoid unnecessary copies
- Use insertion sort for small arrays

Merging iteratively moves the smallest leftmost item of both arrays to the result array. The rightmost index of the left array is `middle`.

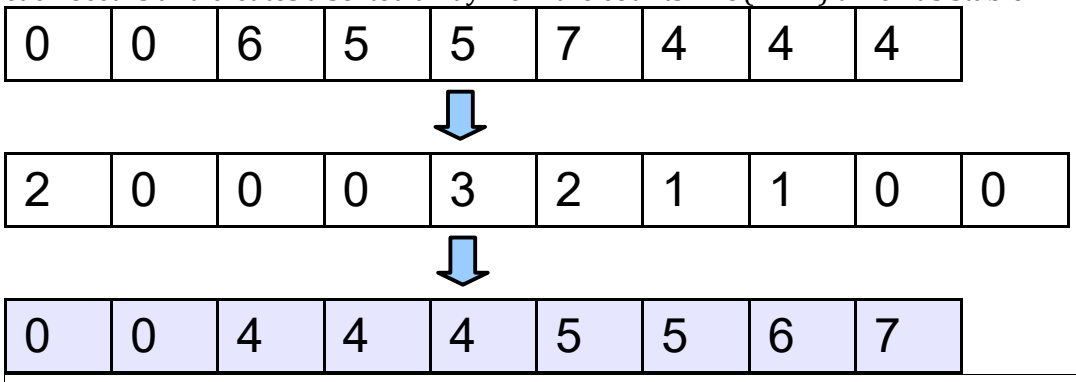
```
template<typename ITEM, typename COMPARATOR> void merge(ITEM* vector,
    int left, int middle, int right, COMPARATOR const& c, ITEM* storage)
{
    for(int i = left, j = middle + 1; left <= right; ++left)
    {
        bool useRight = i > middle || (j <= right &&
            c.isLess(storage[j], 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
        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)
{
    if(n <= 1) return;
    Vector<ITEM> storage(vector, n);
    mergeSortHelper(vector, 0, n - 1, c, storage.getArray());
}
template<typename ITEM> void mergeSort(ITEM* vector, int n)
    {mergeSort(vector, n, DefaultComparator<ITEM>());}
```

The runtime $R(n) = O(n) + 2R(n/2)$. 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.




```
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) counts how many have a particular key, uses the cumulative counts to create a temporary sorted array, and copies it into the original. The implementation needs a functor `ORDERED_HASH` that extracts items' keys.

```
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> count(N + 1, 0);
    for(int i = 0; i < n; ++i) ++count[h(a[i]) + 1];
    for(int i = 0; i < N; ++i) count[i + 1] += count[i];//accumulate counts
    //rearrange items
```

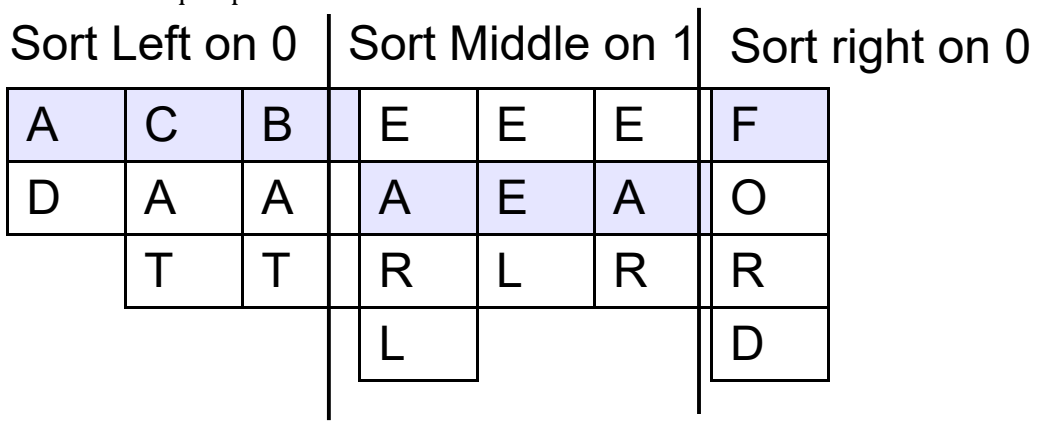
```
    for(int i = 0; i < n; ++i) new(&temp[count[h(a[i])]++])ITEM(a[i]);
    for(int i = 0; i < n; ++i) a[i] = temp[i];
    rawDelete(temp);
}
```

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 $\infty$, 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:




The comparator keeps track of current depth, starting with 0, which allows sorting arbitrary tuples.

```
template<typename VECTOR, typename COMPARATOR> void multikeyQuicksort(VECTOR*
    vector, int left, int right, COMPARATOR comparator)
{
    if(right - left < 1) return;
    int i, j;
    partition3(vector, left, right, i, j, comparator);
    ++comparator.depth;
    multikeyQuicksort(vector, j + 1, i - 1, comparator);
    --comparator.depth;
    multikeyQuicksort(vector, left, j, comparator);
    multikeyQuicksort(vector, i, right, comparator);
}
```

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).

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

```
            partition3(vector, left, right, i, j, comparator);
            //left
            stack.push(left);
            stack.push(j);
            stack.push(comparator.depth);
            //right
            stack.push(i);
            stack.push(right);
            stack.push(comparator.depth);
            //middle
            stack.push(j + 1);
            stack.push(i - 1);
            stack.push(comparator.depth + 1);
        }
    }
}
```

E[the runtime] = O($n$lg($n$)), and the 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 large items, use pointers 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.

## 5.6 Permutation Sort

To sort according to a permutation defined by an array of sorted indices, can copy the items to a temporary array and populate the original from it according to the permutation. But, can avoid the temporary because a permutation is a product of disjoint cycles. Think of $f$ = permutation[$i$] as specifying from where to take the item for position $i$. Mark permutation[$i$] as processed by setting it to $i$, and process permutation[$f$] from which took the item. Need to remember only the first item in the cycle, which is immediately replaced, and a loop over the array gets all cycles in O($n$) time.

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

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.

## 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. It iteratively shrinks the interval containing the item.

```
template<typename ITEM, typename COMPARATOR> ITEM quickSelect(ITEM* vector,
    int left, int right, int k, COMPARATOR comparator)
{
    assert(k >= left && k <= right);
    for(int i, j; left < right;)
    {
        partition3(vector, left, right, i, j, comparator);
        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 also E[the runtime] = O($n$) (Vallee et al. 2009), but can extend multikey quicksort to **multikey quickselect**. Same for partial sort and multiple select (covered later in the chapter).

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

## 5.8 Partial Sort

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($k$).

To sort incrementally, sorting another item only if needed, select it but reuse the right bound values from the previous calls, stored on a stack. This uses the optimal expected O($n$ + $k$lg($k$)) time after selecting $k$ items (Paredes & Navarro 2006) and works because selection is from left to right, so quickselect always executes `right = j`. Before the first call, the stack needs to contain the array's rightmost index, and after each call pop it to ensure `left` + 1 ≤ `right` for the next call. The array is sorted when the stack is empty, and `left` ≥ `right`. Can't modify the stack and the vector between calls.

```
template<typename ITEM, typename COMPARATOR> ITEM incrementalQuickSelect(
    ITEM* vector, int left, Stack<int>& s, COMPARATOR comparator)
{
    for(int right, i, j; left < (right = s.getTop()); s.push(j))
        partition3(vector, left, right, i, j, comparator);
    s.pop();
    return vector[left];
}
```

The unlikely worst case is O($n^2$). To sort completely using partial sort:

```
template<typename ITEM, typename COMPARATOR> void incrementalSort(
    ITEM* vector, int n, COMPARATOR const& c)
{
    Stack<int> s;
```

```
    s.push(n - 1);
    for(int i = 0; i < n; ++i) incrementalQuickSelect(vector, i, s, c);
}
```

## 5.9 Multiple Selection

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 quartiles or quantiles for statistics this way.

```
template<typename ITEM, typename COMPARATOR> void multipleQuickSelect(ITEM*
    vector, bool* selected, int left, int right, COMPARATOR comparator)
{
    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, comparator);
        if(j - left < right - i)//smaller first
        {
            multipleQuickSelect(vector, selected, left, j, comparator);
            left = i;
        }
        else
        {
            multipleQuickSelect(vector, selected, i, right, comparator);
            right = j;
        }
    }
    insertionSort(vector, left, right, comparator);
}
```

∀ 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 $O(n^2)$.

## 5.10 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 comparator)
{
    while(left <= right)
    {
        int middle = (left + right)/2;
        if(comparator.isEqual(key, vector[middle])) return middle;
        comparator.isLess(key, vector[middle]) ?
            right = middle - 1 : left = middle + 1;
    }
    return -1;
}
```

**Exponential search** is useful when the "array" upper bound is unknown. It assumes that it's 1, then 2, 4, 8, etc., and, after it's found, does binary search between bound/2 and bound. It's not useful for arrays, which know their bounds, but 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)).

## 5.11 Comments

Some slower $O(n^2)$ sorts:

- **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**. 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 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.

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).

## 5.12 References

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.

Paredes, R., & Navarro, G. (2006). Optimal incremental sorting. In *Proc. 8th Workshop on Algorithm Engineering and Experiments and 3rd Workshop on Analytic Algorithmics and Combinatorics (ALENEX-ANALCO'06)* (pp. 171–182). 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* (Vol. 1). 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.