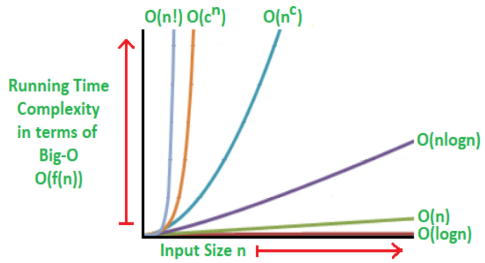
```
c[k] = (a[i] <= b[j]) ? a[i++] : b[j++];
```




O(n!),O(c^n),O(n^c) - Worst
O(nlogn) - Bad
O(n) - Fair
O(logn) - Good
O(1) - Best

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$
$T(1) = 1$
Where $a \geq 1$, $b > 1$ If $f(n) \in \Theta(n^c)$, then:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^c \\ \Theta(n^c \log_2 n) & \text{if } a = b^c \\ \Theta(n^c) & \text{if } a < b^c \end{cases}$$

Big-O: Sufficient (but not necessary) Condition

If $\left[\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = d < \infty\right]$ then $f(n)$ is $O(g(n))$

| | Unordered Array | Ordered (Sorted) Array | Binary Heap |
|---|---|---|---|
| create(range) | $\Theta(n)$ | $\Theta(n \log n)$ | $\Theta(n)$ |
| push() | $\Theta(1)$ | $\Theta(n)$ | $\Theta(\log n)$ |
| top() | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |
| pop() | $\Theta(n)$ or $\Theta(1)$ | $\Theta(1)$ | $\Theta(\log n)$ |

| Sort | Best | Average | Worst | Memory | Stable? | Adaptive? |
|---|---|---|---|---|---|---|
| Bubble | Ω(n) | Θ(n²) | O(n²) | O(1) | Yes | Yes |
| Selection | Ω(n²) | Θ(n²) | O(n²) | O(1) | No | No (not without extra memory) |
| Insertion | Ω(n) | Θ(n²) | O(n²) | O(1) | Yes | Yes |
| Heap | Ω(n log n) | Θ(n log n) | O(n log n) | O(1) | No | No |
| Merge | Ω(n log n) | Θ(n log n) | O(n log n) | O(n) | Yes (if merge is stable) | No |
| Quick | Ω(n log n) | Θ(n log n) | O(n²) | O(log n) | No (unless partition is stable) | No |

- `std::vector` – very compact in memory, very cache-local, amortized constant `push/pop_back`, random access
  - Random `insert/erase` requires linear copying!
  - Elements are stored contiguously in memory.
- `std::deque` – fairly compact in memory, fairly cache-local, amortized constant `push_back/front` & `pop_back/front`, random access
  - Good choice if you need to `push/pop` at the front and back
  - Also will never invalidate pointers through normal use
- `std::list` – very spread out in memory, two pointers per element of memory overhead, constant `insert/erase`, no random access
  - Elements are never moved/copied, good for large objects.
  - Random `insert/erase` (given an iterator) is constant time.

| Operation | Array | Linked List | Deque |
|---|---|---|---|
| add_value(val) (inserts value anywhere in container) | Θ(1) | Θ(1) | Θ(1) |
| remove(val) (remove value from container, but you must find it first) | Θ(n) | Θ(n) | Θ(n) |
| remove(iterator) (remove value from container, but you are given its iterator) | Θ(n) | Θ(n) if singly-linked Θ(1) if doubly-linked | Θ(n) |
| find(value) (find a value in the container) | Θ(n) | Θ(n) | Θ(n) |
| iterator::operator*() (dereference an iterator) | Θ(1) | Θ(1) | Θ(1) |
| operator[](index) (access element with a position at index) | Θ(1) | Θ(n) | Θ(1) |
| insert_after(iterator, val) (insert an element after a provided iterator) | Θ(n) | Θ(1) | Θ(n) |
| insert_before(iterator, val) (insert an element before a provided iterator) | Θ(n) | Θ(n) if singly-linked Θ(1) if doubly-linked | Θ(n) |

- `std::stack` – constant `push()` and `pop()` using back end of `std::deq` or `std::vector`, only access to `top()`
  - "Depth-first" searching, accessing most recent items
- `std::queue` – constant `push()` and `pop()` using front and back ends of `std::deque`, only access to `front()`
  - "Breadth-first" searching, any sort of line (first come → first served)
- `std::priority_queue` – log(n) `push()` and `pop()` using binary heap stored in `std::vector` or `std::deque`, constant access to `top()`
  - Takes a comparator (default `std::less`) to determine priority
  - Can be an efficient way to do things if comparisons are required

| Operation | Array (Sorted) | Linked List (Sorted) | Deque (Sorted) |
|---|---|---|---|
| add_value(val) (inserts value anywhere in container) | Θ(n) | Θ(n) | Θ(n) |
| remove(val) (remove value from container, but you must find it first) | Θ(n) | Θ(n) | Θ(n) |
| remove(iterator) (remove value from container, but you are given its iterator) | Θ(n) | Θ(n) if singly-linked Θ(1) if doubly-linked | Θ(n) |
| find(value) (find a value in the container) | Θ(log(n)) | Θ(n) | Θ(log(n)) |
| iterator::operator*() (dereference an iterator) | Θ(1) | Θ(1) | Θ(1) |
| operator[](index) (access element with a position at index) | Θ(1) | Θ(n) | Θ(1) |
| insert_after(iterator, val) (insert an element after a provided iterator) | N/A | N/A | N/A |
| insert_before(iterator, val) (insert an element before a provided iterator) | N/A | N/A | N/A |

- Pick a pivot from the list
  (you can assume the last element unless specified otherwise)
- Elements less than pivot swapped closer to front
- Elements greater than or equal to pivot swapped closer to back
- After partitioning, the pivot is swapped to the final position
- Recursively sort the sub-list of elements with smaller values and the sub-list of elements with greater values.

## Which sort is best?

- Array that is "almost" already sorted
  - Insertion, or maybe bubble
- Very small array
  - Insertion; Selection if large data (and thus expensive writes)
- Medium size array
  - Quicksort, maybe heapsort
- Large array (about as big as main memory)
  - Heapsort (Θ(1) memory, Θ(*n* log *n*) time), or maybe in-place unstable quicksort (which uses stack space)
- Very large tape drive
  - Merge sort

```
bool operator()(const Node *l1, const Node *l2) const {
  return l1->val > l2->val;
}
};
```

```
struct Interval {
  int start;
  int end;
};

vector<Interval> merge_intervals(vector<Interval> &vec) {
  if (vec.empty()) {
    return vector<Interval>{};
  }
  vector<Interval> result;
  sort(vec.begin(), vec.end(), [](Interval a, Interval b) {
    return a.start < b.start;
  });
  result.push_back(vec.front());
  for (size_t i = 1; i < vec.size(); ++i) {
    if (result.back().end < vec[i].start) {
      result.push_back(vec[i]);
    }
    else {
      result.back().end = max(result.back().end, vec[i].end);
    }
  }
  return result;
}
```

compare( )

```
bool matrix_search(vector<vector<int>> &matrix, int target) {
  int numRows = matrix.size();
  if (numRows == 0)
    return false;
  int numCols = matrix[0].size();
  int currRow = 0;
  int currCol = numCols - 1;
  while (currRow < numRows && currCol >= 0) {
    if (matrix[currRow][currCol] == target)
      return true;
    else if (matrix[currRow][currCol] > target)
      --currCol;
    else
      ++currRow;
  }
  return false;
}
```

2D matrix search

```
struct ListCompare {
  bool operator()(const Node *l1, const Node *l2) const {
    return l1->val > l2->val;
  }
};

Node * merge_lists(vector<Node *> &lists) {
  priority_queue<Node *, vector<Node *>, ListCompare> pq;
```

Compare( )

```
Node * reverse_list(Node *head) {
  Node *prev = nullptr;
  while (head != nullptr) {
    Node *next = head->next;
    head->next = prev;
    prev = head;
    head = next;
  }
  return prev;
};
```

```
int lower_bound(double a[], double val,
                int left, int right) {
  while (right > left) {
    int mid = left + (right - left) / 2;

    if (a[mid] < val)
      left = mid + 1;
    else // (a[mid] >= val)
      right = mid;
  }  // while

  return left;
}  // lower_bound()
```

best search

```
// ONE
double find_median(double a[], int left, int right) {
  int middle = (left + right - 1) / 2;
  if ((right - left) % 2 == 0) {
    double med1 = find_median_helper(a, left, right, middle);
    double med2 = find_median_helper(a, left, right, middle + 1);
    return (med1 + med2) / 2;
  }
  return find_median_helper(a, left, right, middle);
}
double find_median_helper(double a[], int left, int right, int middle) {
  int pivot = partition(a, left, right);
  if (pivot < middle)
    return find_median_helper(a, pivot + 1, right, middle);
  else if (pivot > middle)
    return find_median_helper(a, left, pivot, middle);
  return a[middle];
}
```

String length

```
char str1[] = "BING";
cout << strlen(str1);
cout << sizeof(str1);

string str2("BING");
cout << str2.length();
cout << str2.size();
```

What is the output?

A. 4444

B. 4455

C. 4544

**Big-Omega**

- an **asymptotic lower bound**
- $f \in \Omega(g)$ means $f$ grows **at lea**

**Big-Theta**

- an **asymptotic tight bound**,

```
void bubble(Item a[], size_t left, size_t right) {
  for (size_t i = left; i < right - 1; ++i) {
    bool swapped = false;
    for (size_t j = right - 1; j > i; --j) {
      if (a[j] < a[j - 1]) {
        swapped = true;
        swap(a[j - 1], a[j]);
      }  // if
    }  // for j
    if (!swapped)
      break;
  }  // for i
}  // bubble()
```

① bubble

```
void selection(Item a[], size_t left, size_t right) {
  for (size_t i = left; i < right - 1; ++i) {
    size_t minIndex = i;
    for (size_t j = i + 1; j < right; ++j)
      if (a[j] < a[minIndex])
        minIndex = j;
    if (minIndex != i)
      swap(a[i], a[minIndex]);
  }  // for
}  // selection()
```

② selection

- Extra comparison checks if item is already in position

```
size_t partition(Item a[], size_t left, size_t right) {
  size_t pivot = --right;
  while (true) {
    while (a[left] < a[pivot])
      ++left;
    while (left < right && a[right - 1] >= a[pivot])
      --right;
    if (left >= right)
      break;
    swap(a[left], a[right - 1]);
  }  // while
  swap(a[left], a[pivot]);
  return left;
}  // partition()
```

③ Lose partition

- Choose last item as pivot
- Scan…
  - from left for >= pivot
  - from right for < pivot
- Swap left & right pairs and continue scan until left & right cross
- Move pivot to 'middle'

```
void merge_sort(Item a[], size_t left, size_t right) {
  if (right < left + 2)  // base case: 0 or 1 item(s)
    return;
  size_t mid = left + (right - left) / 2;
  merge_sort(a, left, mid);  // [left, mid)
  merge_sort(a, mid, right);  // [mid, right)
  merge(a, left, mid, right);
}  // merge_sort()
```

④ Merge sort

```
void merge(Item a[], size_t left, size_t mid, size_t right) {
  size_t n = right - left;
  vector<Item> c(n);

  for (size_t i = left, j = mid, k = 0; k < n; ++k) {
    if (i == mid)
      c[k] = a[j++];
    else if (j == right)
      c[k] = a[i++];
    else
      c[k] = (a[i] <= a[j]) ? a[i++] : a[j++];
  }  // for

  copy(begin(c), end(c), &a[left]);
}  // merge()
```

```
void insertion(Item a[], size_t left, size_t right) {
  for (size_t i = right - 1; i > left; --i)  // find min item
    if (a[i] < a[i - 1])                     // put in a[left]
      swap(a[i - 1], a[i]);                  // as sentinel

  for (size_t i = left + 2; i < right; ++i) {
    Item v = a[i]; size_t j = i;           // v is new item
    while (v < a[j - 1]) {                 // v in wrong spot
      a[j] = a[j - 1];                     // copy/overwrite
      --j;
    }  // while
    a[j] = v;
  }  // for i
}  // insertion()
```

⑤ Insertion

a = { 4 2 5 1 3 }  input
a = { 1 4 2 5 3 }  sentinel
a = { 1 2 4 5 3 }  pass 1
a = { 1 2 4 5 3 }  pass 2
a = { 1 2 3 4 5 }  pass 3

```
void counting_sort(vector<Student> &students) {
  vector<Student> result(students.size());
  vector<size_t> counts(MAX);
  size_t temp, sum = 0;

  for (auto &s : students)                 // Pass 1
    ++counts[s.classStanding()];
  for (size_t i = 0; i < MAX; ++i) {       // Pass 2
    temp = counts[i];
    counts[i] = sum;
    sum += temp;
  }  // for i
  for (auto &s : students) {               // Pass 3
    result[counts[s.classStanding()]] = s;
    ++counts[s.classStanding()];
  }  // for
  swap(students, result);
}  // counting_sort()
```

⑥ Co

```
void quicksort(Item a[], size_t left, size_t right) {
  if (left + 1 >= right)
    return;
  size_t pivot = partition(a, left, right);
  if (pivot - left < right - pivot) {
    quicksort(a, left, pivot);
    quicksort(a, pivot + 1, right);
  } else {
    quicksort(a, pivot + 1, right);
    quicksort(a, left, pivot);
  }  // else
}  // quicksort()
```

- Worst memory requirement?
- Both sides equal: $O(\log n)$

```
size_t partition(Item a[], size_t left, size_t right) {
  size_t pivot = left + (right - left) / 2; // pivot is middle
  swap(a[pivot], a[--right]);               // swap with right
  pivot = right;                            // pivot is right

  while (true) {
    while (a[left] < a[pivot])
      ++left;
    while (left < right && a[right - 1] >= a[pivot])
      --right;
    if (left >= right)
      break;
    swap(a[left], a[right - 1]);
  }  // while
  swap(a[left], a[pivot]);
  return left;
}  // partition()
```

middle partition

- Choose middle item as pivot
- Swap it with the right end
- Repeat as before

```
void fixUp(Item heap[], int k) {
  while (k > 1 && heap[k / 2] < heap[k]) {
    swap(heap[k], heap[k / 2]);
    k /= 2;  // move up to parent
  }  // while
}  // fixUp()
```

Note: root is well-known (position 1)

1) Pass index *k* of array element with increased priority
2) Swap the node's key with the parent's key until:
   a. the node has no parent (it is the root), or
   b. the node's parent has a higher (or equal) priority key

```
void fixDown(Item heap[], int heapsize, int k) {
  while (2 * k <= heapsize) {
    int j = 2 * k;  // start with left child
    if (j < heapsize && heap[j] < heap[j + 1]) ++j;
    if (heap[k] >= heap[j]) break;  // heap restored
    swap(heap[k], heap[j]);
    k = j;  // move down
  }  // while
}  // fixDown()
```

1) Pass index *k* of array element with decreased priority
2) Exchange the key in the given node with the highest priority key among the node's children, moving down until
   a. the node has no children (leaf node), or
   b. the node has no children with a higher key

```
void heapsort(Item heap[], int n) {
  heapify(heap, n);
  for (int i = n; i >= 2; --i) {
    swap(heap[i], heap[1]);
    fixDown(heap, i - 1, 1);
  }  // heapsort()
}  // heapsort()
```

Root is index 1

Part 1: Transform unsorted array into heap (*heapify*)
Part 2: Remove the highest priority item from heap, add it to sorted sequence, and fix the heap, repeat…

```
size_t find(size_t x) {
  while (reps[x] != x) {
    x = reps[x];
  }
  return x;
}
```

```
void push(Item newItem) {
  heap[++heapsize] = newItem;
  fixUp(heap, heapsize);
}  // push()
```

1) Insert `newItem` into bottom of tree/heap, i.e., last element
2) `newItem` "bubbles up" tree swapping with parent while parent's key is less (use greater for min-heap)

```
void pop() {
  heap[1] = heap[heapsize--];
  fixDown(heap, heapsize, 1);
}  // pop()
```

1) Remove root element – results in disjoint heap
2) Move the last element into the root position
3) New root "sinks" down the tree swapping with highest priority child whose key is greater (less for min-heap)

## When Counting Sort Can Be Used

- ONLY usable when the number of keys is limited (and small)
  - For example:
    - Number of possible birthdays annually = 366
    - Number of possible class standings = 4
  - INT_MAX is not "limited"
  - Changes with architecture
  - For 64 bit computers, $2^{64}$ 8-byte integers = 144 quintillion bytes = 160 million 1TB hard drives

```
size_t find(size_t x) {
  size_t pathStart = x;
  // Pass 1 - find the ultimate rep
  while (reps[x] != x) {
    x = reps[x];
  }
  // x is now the ultimate rep
  // Pass 2 - path compression
  while (reps[pathStart] != x) {
    size_t tmp = reps[pathStart];
    reps[pathStart] = x; // Update path to ultimate rep
    pathStart = tmp;
  }
  return x;
}
```

```
class UnionFind {
private:
  vector<size_t> reps;
public:
  UnionFind(size_t size) {
    reps.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
      // at the beginning, every
      // node represents itself!
      reps.push_back(i);
    }
  }
  size_t find(x);
  void set_union(x, y);
```