# Recursion and Algorithms

PIC 10B, UCLA

# Recursion and Algorithms

Recursion is a process where a procedure/function calls itself. It arises naturally when an operation is “similar to itself”. Implementing recursion is important for data structures such as binary search trees that organize their data in a self-similar fashion, in implementing sorting algorithms, managing order of operations, etc. It requires a slightly different mindset to traditional coding, but when used correctly and appropriately, it can be very powerful.

At this stage, it is also important to consider how different functions/algorithms are implemented and consider the practical question of: “how long does it take to run?”

# Computing the Sum of Consecutive Integers

Suppose that we wanted a function to sum consecutive integers from **a** to **b** inclusive.

We first consider a very natural implementation of this sum:

```
/**
Function will sum integers from a to b inclusive.
@param a the starting integer
@param b the ending integer
@return the sum
*/
constexpr int sum_a_to_b(int a, int b) {
    int sum = 0; // start sum at 0
    for (int i=a; i <= b; ++i) { // and add each integer in range to sum
        sum += i;
    }
    return sum;
}
```

# Computing the Sum of Consecutive Integers

Now we consider a recursive means of doing the same thing.

Let's consider a concrete case in evaluating **sum_a_to_b(4,8)**. We note:

*sum_a_to_b(4,8)*
*= 4 + sum_a_to_b(5,8)*
*= 4 + ( 5 + sum_a_to_b (6,8) )*
*= 4 + ( 5 + ( 6 + sum_a_to_b(7,8) ) )*
*= 4 + ( 5 + ( 6 + (7 + sum_a_to_b(8,8) ) ) )*
*= 4 + (5 + (6 + (7 +* **8***) ) )*

# Computing the Sum of Consecutive Integers

With each step, we pull out a single term and add it to a “simpler” function call (one that ultimately involves adding fewer arguments).

At the end of the sequence, we know that **sum_a_to_b(8,8) = 8** so no more function calls are necessary.

This result is fed back to give **7 + 8**, the value of **sum_a_to_b(7,8)**, which can be evaluated to **15**, which was then used to evaluate **6 + sum_a_to_b(7,8)**, the value of **sum_a_to_b(6,8)**, etc.

# Computing the Sum of Consecutive Integers

Let's consider what we did in C++ code:

```
constexpr int sum_a_to_b(int a, int b) { // accept range of values to sum
    if (a < b) { // if more to sum, break off a and recurse
        return a + sum_a_to_b(a+1,b);
    }
    else if (a==b) { // if lower and upper bounds equal, just return value
        return a;
    }
    else { // means there's nothing to sum over
        return 0;
    }
}
```

# Computing the Sum of Consecutive Integers

**Note:** the function calls itself! This is okay, provided we can guarantee we will eventually reach one of the **base cases** whereby the function does not call itself.

We can otherwise obtain infinite recursion...

*This example was for illustration of the concept: in practice writing the loop is a lot easier to read!*

# Factorial

Recall the factorial function $n! = n \times (n-1) \times (n-2) \times ... \times 3 \times 2 \times 1$ with 0! and 1! defined to be 1.

Note this means that $n! = n(n-1)!$ unless $n = 0$. Here's an implementation:

```
constexpr int factorial(int n) {
    return (n > 0) ? n*factorial(n-1) : 1;
}
```

# Infinite Recursion

As a silly example without a base case, consider

```
int f() {
    return f();
    return 42;
}
```

This is sure to cause a stack overflow! To run **f**, **f** must call itself, which in turn calls itself, and so on. But it never happens that **f** gets a chance to do anything other than call itself again.

# Infinite Recursion

Recursion can be very costly for the stack. In the simple sum, **sum_a_to_b(4,8)**, 4 was saved for the first return and two parameters, 5 and 8, were passed to **sum_a_to_b**; this call in turn set aside 5 and passed two new parameters 6 and 8, to **sum_a_to_b**, etc.

Recursion isn't a tool for all problems: some problems are made significantly easier with recursion; others become more complicated by using recursion.

Generally any problem that can be solved with recursion can also be solved **iteratively** (i.e., without recursion), but it may be very hard to write.

# Recursion Tips

Some tips on recursion:

- **Be lazy:** do very little in one step and defer all the other work to another function call.
- **Be optimistic:** assuming the base case is managed, trust that the rest of the job will be done when recursion is used.
- **Be sure to have a base case:** determine the "last step(s)" and write a special piece of logic to manage them.
- **Don’t think too hard:** thought is involved, no doubt, but sometimes thinking too much about what a recursive function does makes it less intuitive and more confusing.

# Checking for Palindromes

Suppose we want a function that will determine if a given string is a palindrome, i.e. if it is exactly the same both forwards and backwards. This can be done both normally and recursively.

Note that the empty string or a string with only a single **char** is a palindrome. We can implement this straightforwardly as:

# Checking for Palindromes

```
bool isPalindrome(const std::string& s) {
    if ( s.size() <= 1 ) { // true if empty or a single char
        return true;
    }

    /* compare start and end, advance lower position, decrease upper
       position. Compare until mismatch or they meet. */
    for ( std::size_t low = 0, up = s.size()-1; low <= up; - - up, ++low ) {

        if ( s[low] != s[up] ) { // not palindrome if mismatch
            return false;
        }
    }

    return true; // if we got here, same both ways
}
```

# Checking for Palindromes

Recall that in a **for** loop, the syntax is:
**for( initialization; condition; execute)**

The **initialization** statement can initialize more than one variable, provided the types match.

The **condition** statement should evaluate to **true**/**false** (or be expliclty convertable to **bool**).

The statement to **execute** can involve the comma operator!

Any of the three statements can be empty, too!

# Checking for Palindromes

This could also be done rather nicely with recursion:

```
constexpr bool isPalindrome(const std::string_view& s) {
    if ( s.size() <= 1 ) { // true if empty or a single char
        return true;
    }
    else { // then multiple chars
        // it is false if the first and last chars do not match
        if ( s[0] != s[s.size()-1] ) {
            return false;
        }
    }

    // if here, nip off ends and check again from 2nd to 2nd last char
    return isPalindrome( s.substr(1,s.size()-2) );
}
```

# Checking for Palindromes

A **std::string_view** object is part of C++17. It behaves like a more efficient version of **std::string** when passed around. It only tracks the range of indices to manage in an array.

A **std::string_view** has various constructors; one accepts a **char** array terminated by a null character like a string literal or the output of the **c_str** function of **std::string**.

Unlike **std::string**, a **std::string_view** can be used in a **constexpr** context!

# Checking for Palindromes

```
std::string message("hello");

// acts like message but is not a copy of if: we can view message
std::string_view first(message.c_str());
constexpr std::string_view second(" world");

std::cout << first << second; // prints "hello world"
```

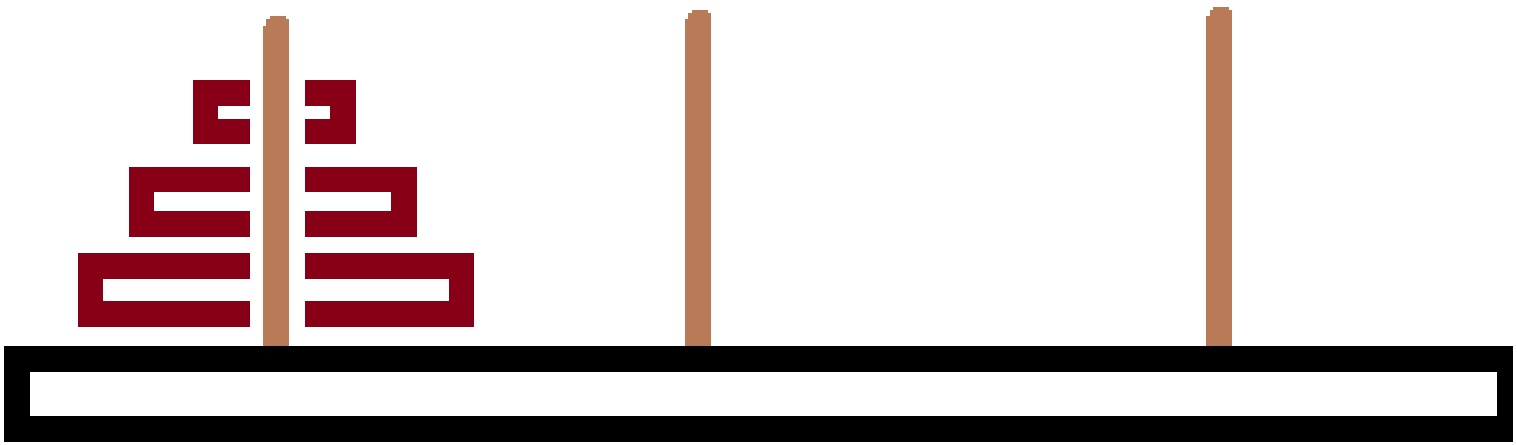
Recall that the **string::substr** function with two specified arguments begins at the first index and extracts a substring having the size of the second argument (as opposed to ending at the second argument).

The **c_str** function of **std::string** returns a pointer to the array of **char**s owned by the **std::string**.

# Tower of Hanoi

The Tower of Hanoi is a famous puzzle with 3 pegs at the left, middle, and right, and *n* disks of distinct sizes. The puzzle begins with the *n* disks on the leftmost peg, say, with the disks sorted in decreasing order of size from bottom to top.

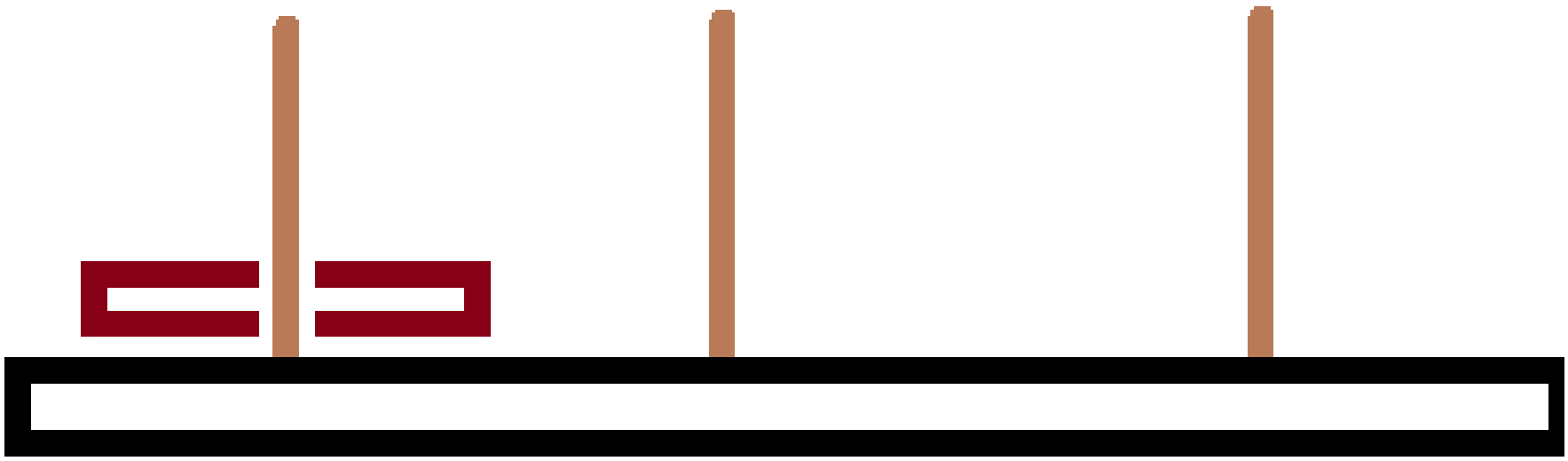
The problem is to move all of the disks to the rightmost peg. Disks can be moved one at a time from one peg to another peg but a bigger disk cannot be placed on top of a smaller disk.



# Tower of Hanoi

If there are no disks then the problem doesn't even make sense. So suppose we have one disk.

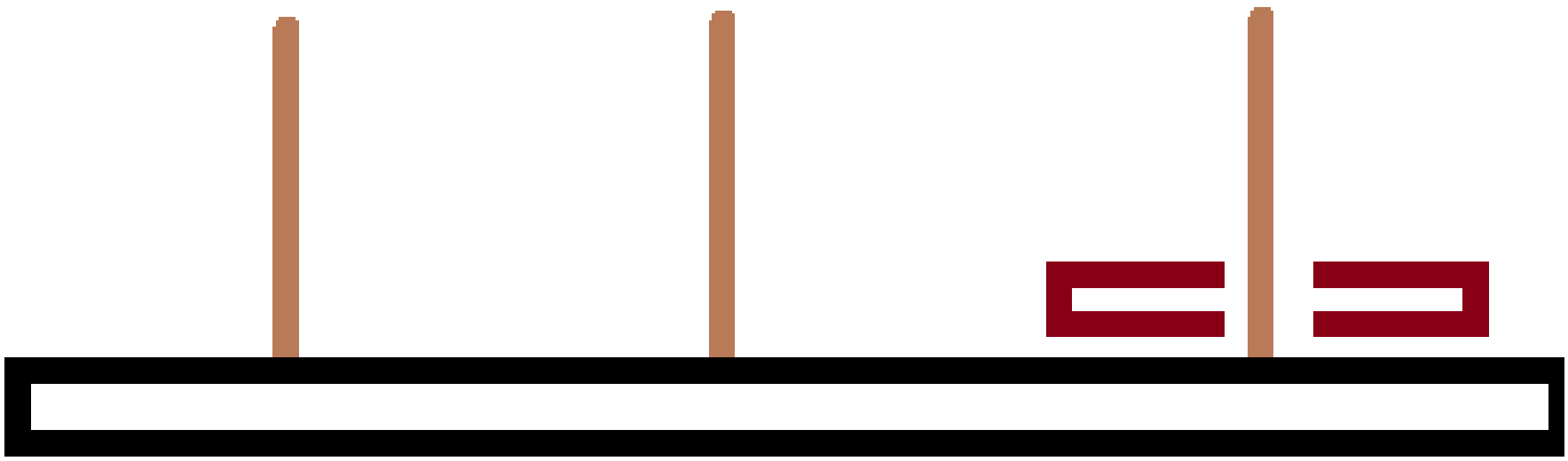
With one disk, we can simply move it from the left peg to the right.



# Tower of Hanoi

If there are no disks then the problem doesn't even make sense. So suppose we have one disk.

With one disk, we can simply move it from the left peg to the right.



# Tower of Hanoi

We denote the solution with one disk by **move(1, left, right, mid)** signifying: move one disk from the left peg to the right peg.

Note that in the **move** notation, the last parameter describes the peg that is neither the beginning nor ending destination.

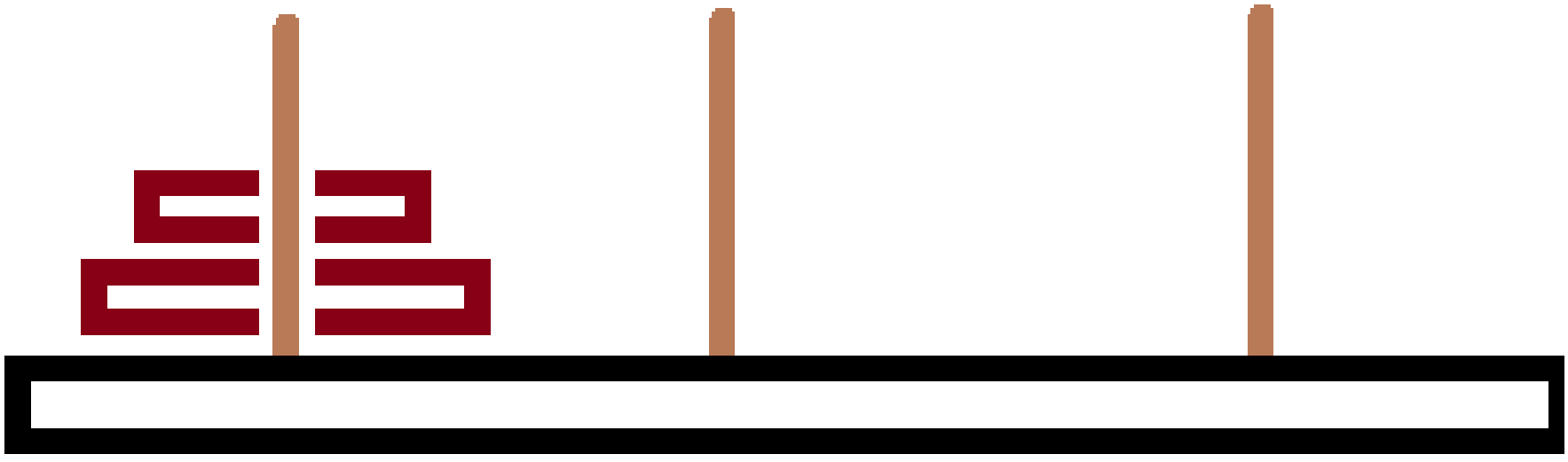
Also note that because this maneuver is so simple, we can **move(1,mid,right,left)** just as simply, i.e., we can move a single disk from the middle position to the right position.

# Tower of Hanoi

With two disks, we can move the small disk to the middle, the large disk to the right, and then the small disk to the right.

We can describe this by:

- **move(1,left,mid,right)**
- **move(1,left,right,mid)**
- **move(1,mid,right,left)**

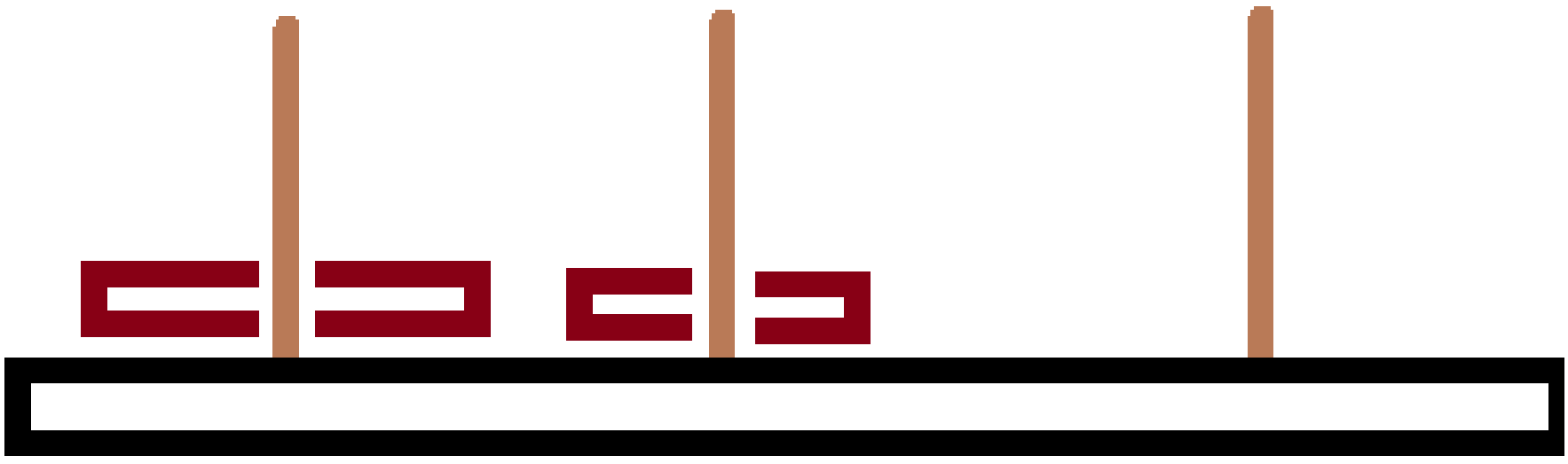


# Tower of Hanoi

With two disks, we can move the small disk to the middle, the large disk to the right, and then the small disk to the right.

We can describe this by:

- **move(1,left,mid,right)**
- **move(1,left,right,mid)**
- **move(1,mid,right,left)**

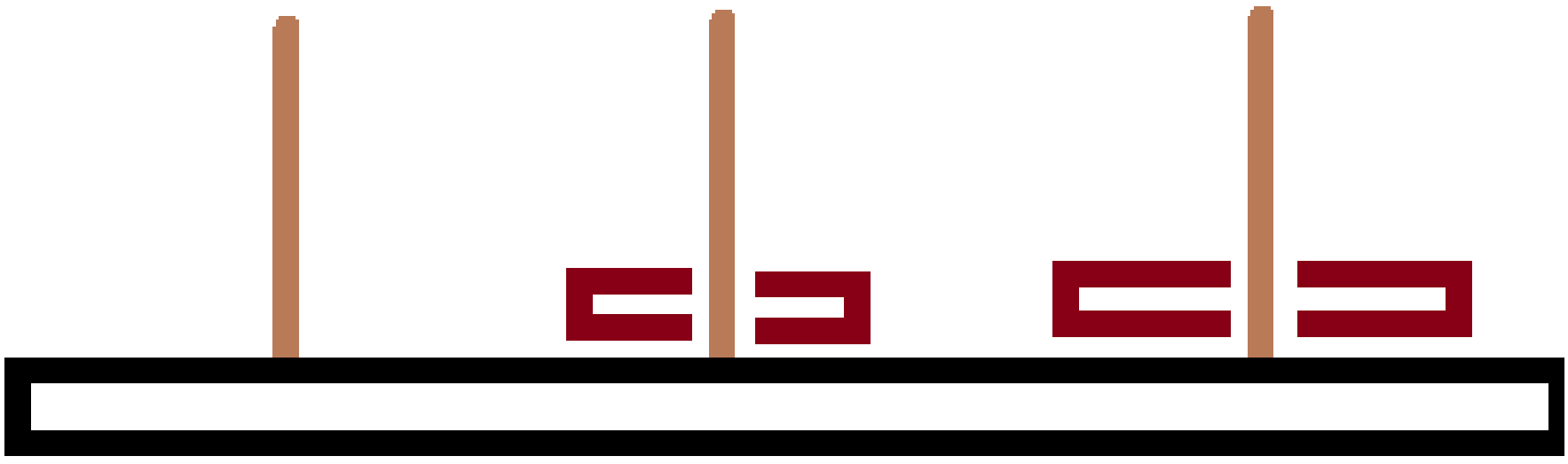


# Tower of Hanoi

With two disks, we can move the small disk to the middle, the large disk to the right, and then the small disk to the right.

We can describe this by:

- **move(1,left,mid,right)**
- **move(1,left,right,mid)**
- **move(1,mid,right,left)**

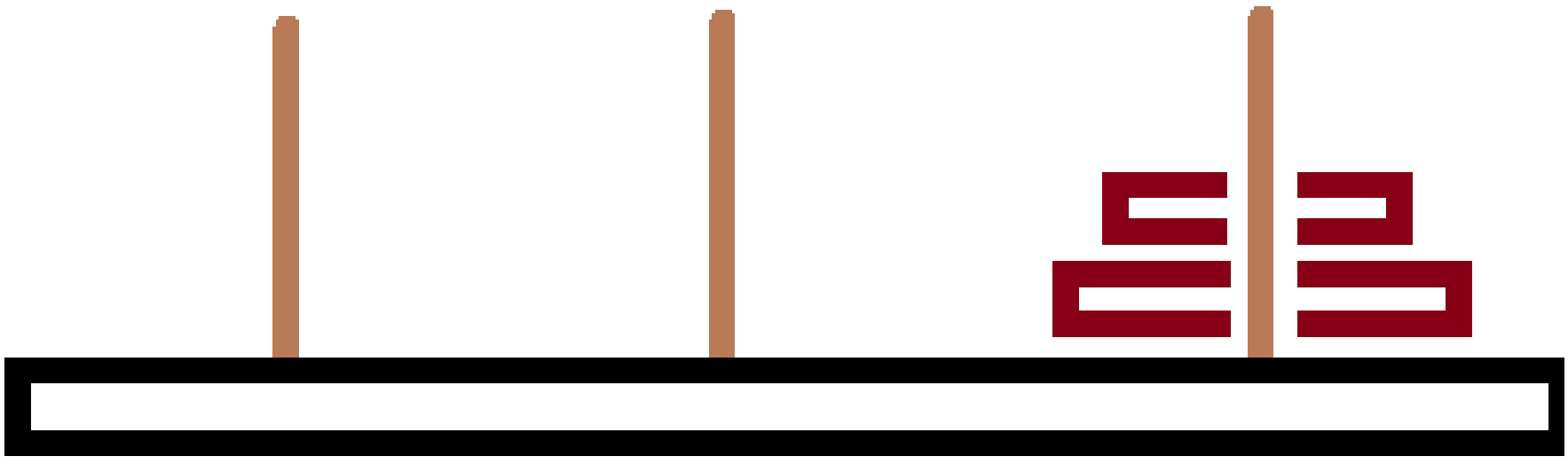


# Tower of Hanoi

With two disks, we can move the small disk to the middle, the large disk to the right, and then the small disk to the right.

We can describe this by:

- **move(1,left,mid,right)**
- **move(1,left,right,mid)**
- **move(1,mid,right,left)**



# Tower of Hanoi

Anticipating needing to move more disks around, we come up with a general notation:

**move(n,x,y,z)**

to denote moving the **top n** disks on peg **x** to peg **y**.

Thus in order to **move(2,left,right,mid)**, we

- **move(1,left,mid,right)**
- **move(1,left,right,mid)**
- **move(1,mid,right,left)**

# Tower of Hanoi

With $n$ disks, we can (somehow) move the top $n-1$ disks to the middle peg, move the large disk to the right, and (somehow) move the $n-1$ smallest disks from the middle to the right.

We can describe this by:

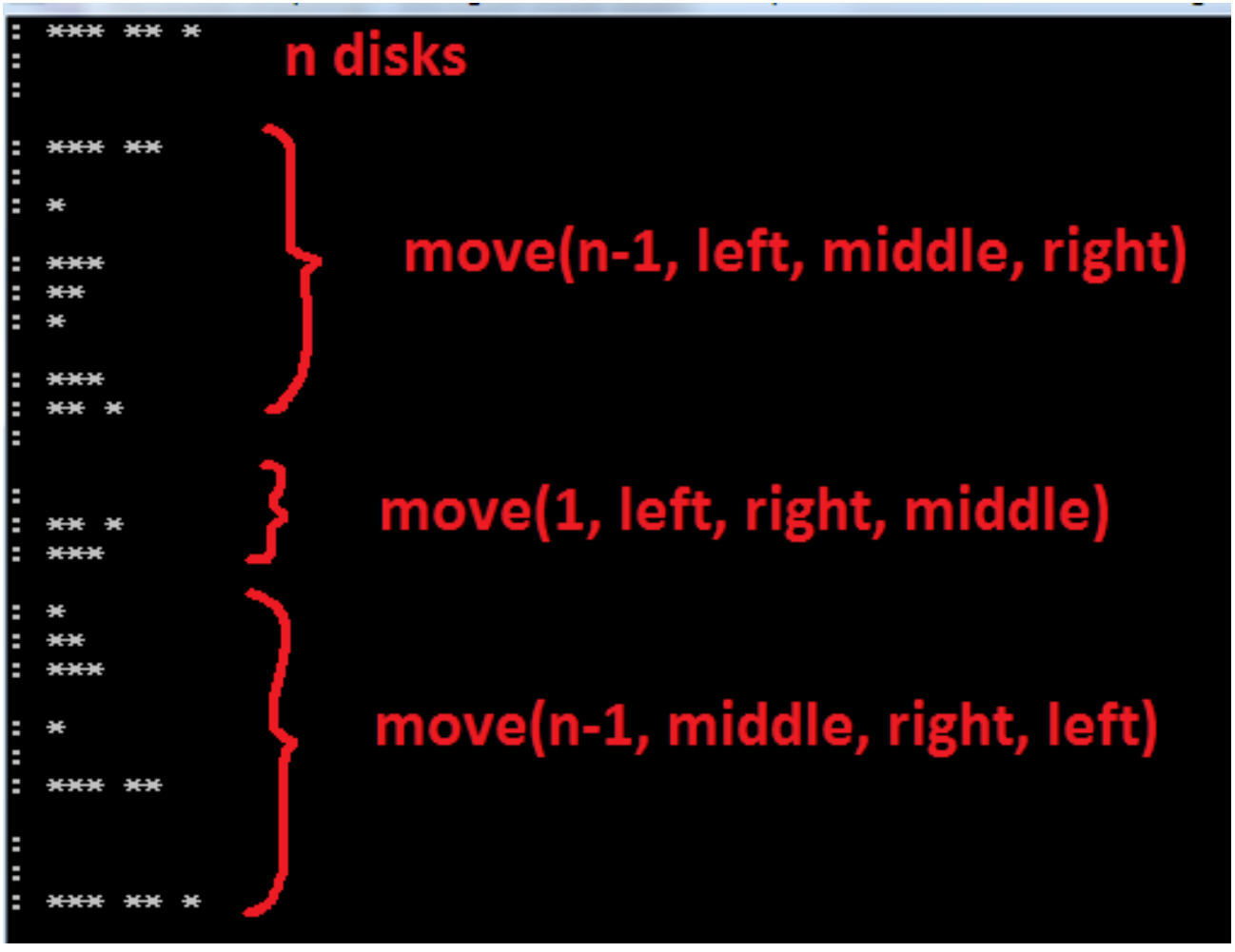
- **move(n-1,left,mid,right)**
- **move(1,left,right,mid)**
- **move(n-1,mid,right,left)**

*Note:* the "somehow"s aren't so mysterious. If we can move 1 disk around, we can move 2 around since 2 depends upon 1. And if we can move 2 around, we can move 3 around since 3 depends upon 2, etc.

The Tower of Hanoi has a recursive solution.

# Tower of Hanoi

The display below illustrates the solution with $n = 3$ where the left, middle, and right, are top, middle, and bottom. The number of stars indicate the size of a disk. Steps proceed in sequential order down.



# Tower of Hanoi

We could also wonder how many steps it takes to solve the problem with $n$ disks.

To solve with $n$ disks, we need to solve two sub-problems with $n-1$ disks each, plus a 1 disk problem.

The one disk problem is just one step.

If $T(n)$ is the number of steps with $n$ disks then:
$T(2) = 2T(1) + 1 = 3$.
$T(3) = 2T(2) + 1 = 7$.
$T(4) = 2T(3) + 1 = 15$.

The pattern we see is that $T(n) = 2^n - 1$.

The solution algorithm is very costly, which we'll discuss later...

# Generating All Permutations

Consider code to generate all permutations of *n* elements of a **std::vector<int>**. For example, if the vector stored:

**{1, 2}** then it should generate all 2 permutations:
1 2
2 1

**{1, 2, 3}** then it should generate all 6 permutations:
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

etc.

# Generating All Permutations

Our basic plan is to "place" items from the beginning to the end, left to right.

If we have *reached the end* then we have *completed a permutation*.

Since items will be placed from the left to right, *unplaced* items will *lie to the right of the placed items.*

At *each position* where an *item needs to be placed*, we will give *all unplaced items a chance to take that spot*.

# Generating All Permutations

```
void permute(std::vector<int>& v, size_t place=0) {

   size_t v_size = v.size();
   if (place == v_size-1 ) { // at last position
      for (auto i : v) { std::cout <<i <<" "; } // print the values
      std::cout <<'\n';
   }
   else { // then still more to place
      for(size_t other = place;
          other < v_size; ++other) {
          // give all unplaced elements a chance to be at position

          std::swap(vec[place], vec[other]); // place unplaced
          permute(vec, place+1); // have placed one more
          std::swap(vec[place], vec[other]); // undo for next unplaced
      }
   }
}
```

# Generating All Permutations

We can call upon this function via:

```
std::vector<int> v {1,2,3,4,5};
permute(v); // will print all 5! = 120 permutations
```

# Generating All Permutations

Notice the loop structure:

```
for(size_t other = place;
    other <= v_size; ++other)
```

In general it is best to avoid unnecessarily member function invocations as part of a loop condition check. The style above is more efficient than

```
// Inefficient: has to invoke member function in each condition check
for(size_t other = place; other < v.size(); ++other)
```

Unless there is a danger the vector size could change, the value of **last** is known for the duration of the loop.

# Sorting Algorithms

Recursion can be extremely useful in sorting data. Suppose that we wish to sort a list of numbers in increasing order. There are many approaches, but we consider three.

- *Selection sort*: repeatedly find the minimum of the unsorted entries and swap the minimum with the first of the unsorted entries.
- *Bubble sort*: move forward through the list swapping adjacent entries if they are not in sorted order, cycling back over the unsorted parts until all the items have been sorted.
- *Merge sort:* repeatedly split the list into smaller lists and those smaller lists into smaller lists, etc.; then, merge the small lists into larger sorted lists, and merge those into even larger sorted lists, etc.

# Sorting Algorithms: Selection Sort

Consider sorting **7 2 1 8 1**.

Parentheses indicate the current term being considered,
* indicates an iteration's minimum value, and
underline indicates the number is in place.

7 2 1 8 1 $\rightarrow$7* (2) 1 8 1 $\rightarrow$7 2* (1) 8 1 $\rightarrow$7 2 1* (8) 1 $\rightarrow$7 2 1* 8 (1) $\rightarrow$
7 2 1* 8 1 $\rightarrow$<u>1</u> 2 7 8 1 $\rightarrow$

<u>1</u> 2* (7) 8 1 $\rightarrow$<u>1</u> 2* 7 (8) 1 $\rightarrow$<u>1</u> 2* 7 8 (1) $\rightarrow$<u>1</u> 2 7 8 1* $\rightarrow$<u>1</u> <u>1</u> 7 8 2 $\rightarrow$

<u>1</u> <u>1</u> 7* (8) 2 $\rightarrow$<u>1</u> <u>1</u> 7* 8 (2) $\rightarrow$<u>1</u> <u>1</u> 7 8 2* $\rightarrow$<u>1</u> <u>1</u> <u>2</u> 8 7 $\rightarrow$

<u>1</u> <u>1</u> <u>2</u> 8* (7) $\rightarrow$<u>1</u> <u>1</u> <u>2</u> 8 7* $\rightarrow$<u>1</u> <u>1</u> <u>2</u> <u>7</u> 8 $\rightarrow$

<u>1</u> <u>1</u> <u>2</u> <u>7</u> <u>8</u>

# Sorting Algorithms: Selection Sort

We consider the implementation of selection sort, **selSort**, where the user can call this function by passing a **std::vector<int>**, to be sorted, along with a lower and upper range of **size_t** positions where the sorting is to take place inclusively.

We will make use of a library function, **std::swap**. As with smart pointers, the swap function, when applied to two **int**s (or most variable types in general) it will swap their values.

We will implement **selSort** recursively: placing the smallest element of the desired range in its place and then calling **selSort** on the next range of values.

# Sorting Algorithms: Selection Sort I

```
/**
This will sort a vector via selection sort
@param v the vector to sort
@param beg the lowest index position to include in the sort
@param end the largest index position to include in the sort
*/
void selSort(std::vector<int>& v, size_t beg, size_t end){

    // check the indices are valid for the sort
    if ( ( end > v.size() - 1) || (beg > end) || (beg > v.size()-1) ) {
        throw std::logic_error("invalid input range"); // if not throw exception
    }

    if (beg == end) { // done if this is the case: nothing to sort
        return;
    }
```

# Sorting Algorithms: Selection Sort II

```
    size_t minpos = beg; // index of smallest value

    for (size_t i = beg + 1; i <= end; ++i) { // loop over compare values
        if (v[i] < v[minpos]) { // update index to smallest value if new smallest
            minpos = i;
        }
    }

    // place smallest element of unsorted range in its place
    std::swap(v[beg], v[minpos]);

    // sort next part of vector now that beg is in place
    selSort(v, beg + 1, end);
}
```

# Sorting Algorithms: Bubble Sort

Consider sorting **7 2 1 8 1**.

Parentheses indicate the current terms being compared,
underline indicates an item is in place.

7 2 1 8 1 $\rightarrow$(7 2) 1 8 1 $\rightarrow$2 (7 1) 8 1 $\rightarrow$2 1 (7 8) 1 $\rightarrow$2 1 7 (8 1) $\rightarrow$
2 1 7 1 $\underline{8}$ $\rightarrow$

(2 1) 7 1 $\underline{8}$ $\rightarrow$1 (2 7) 1 $\underline{8}$ $\rightarrow$1 2 (7 1) $\underline{8}$ $\rightarrow$1 2 1 $\underline{7}$ $\underline{8}$ $\rightarrow$

(1 2) 1 $\underline{7}$ $\underline{8}$ $\rightarrow$1 (2 1) $\underline{7}$ $\underline{8}$ $\rightarrow$1 1 $\underline{2}$ $\underline{7}$ $\underline{8}$ $\rightarrow$

(1 1) $\underline{2}$ $\underline{7}$ $\underline{8}$ $\rightarrow$1 $\underline{1}$ $\underline{2}$ $\underline{7}$ $\underline{8}$ $\rightarrow$

$\underline{1}$ $\underline{1}$ $\underline{2}$ $\underline{7}$ $\underline{8}$

# Sorting Algorithms: Bubble Sort

We consider the implementation of bubble sort, **bubSort**, where the user can call this function by passing a **std::vector<int>**, to be sorted, along with a lower and upper range of **size_t** positions where the sorting is to take place inclusively.

We will implement **bubSort** recursively: calling itself until there is an iteration where no swaps are necessary.

The largest items will be put in their place first because with each comparison, the large values gets pushed down the list.

# Sorting Algorithms: Bubble Sort I

```
/**
This will sort a vector via bubble sort
@param v the vector to sort
@param beg the lowest index position to include in the sort
@param end the largest index position to include in the sort
*/
void bubSort(std::vector<int>& v, size_t beg, size_t end){
    // check range is valid

    // throw if invalid
    if ( (end > v.size() - 1) || (beg > end) || (beg > v.size()-1) ) {
        throw std::logic_error("invalid input range");
    }

    if ( beg == end) { // if no range to swap
        return; // done if this is the case
    }
```

# Sorting Algorithms: Bubble Sort II

```
    /* perform a series of comparisons of adjacent elements and swap so
    lower value is left up upper value */

    for (size_t i = beg; i < end; ++i) { // loop over adjacent pairs
        if (v[i] > v[i + 1]) { // swap if left bigger than right
            std::swap(v[i], v[i + 1]);
        }
    }

    // with end in place, sort over smaller range
    bubSort(v, beg, end-1);
}
```

# Sorting Algorithms: Bubble Sort I

Both selection sort and bubble sort could be implemented quite easily without recursion, too. Consider selection sort without recursion:

```
void selSort(std::vector<int>& v, size_t beg, size_t end) {

   if ( (end > v.size() - 1) || (beg > end) ) { // if invalid range
      throw std::logic_error("invalid input range"); // then throw exception
   }

   for(size_t i = beg; i < end; ++i) { // loop over unsorted range

      size_t minpos = i; // assume minpos is leftmost initially
```

# Sorting Algorithms: Bubble Sort II

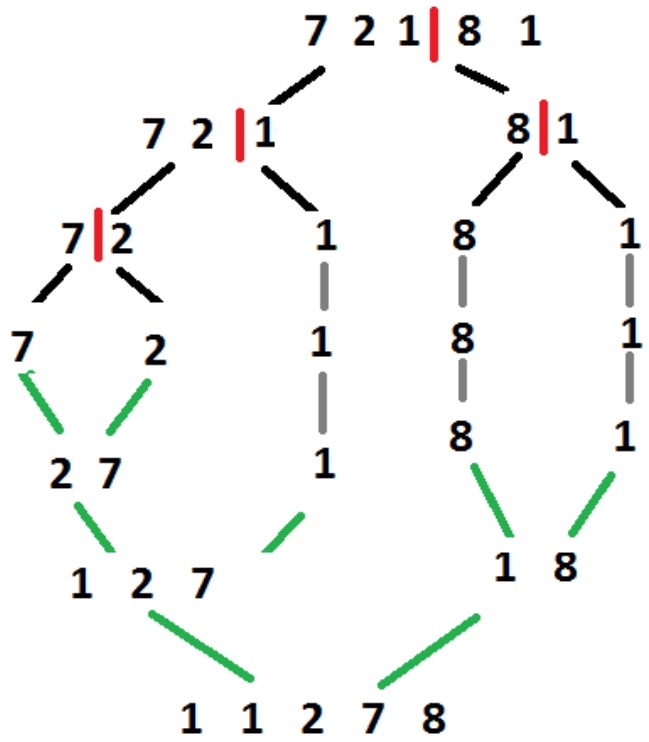
```
    for(size_t j=i+1; j <= end; ++j) { // compare v[j] to v[i]

      if( v[j] < v[minpos] ) { // update minpos if new min found
        minpos = j;
      }
    }

    std::swap(v[i], v[minpos]); /* place new min value in lowest available
      position */
  }
}
```

# Sorting Algorithms: Merge Sort

Consider sorting **7 2 1 8 1**.

The list is repeatedly split into more manageable chunks, which are in turn split, and so on. At the simplest level, these become a single element.

The single elements can merge with each other, which can merge with other small lists, which merge to form larger lists, which in turn can merge and so on. Splitting is in red and merging in green.



# Sorting Algorithms: Merge Sort

We consider the implementation of **mergeSort** where the user can call this function by passing a **std::vector<int>**, to be sorted, along with a lower and upper range of **size_t** positions where the sorting is to take place inclusively.

A list will be split by performing integer division on the number of elements to sort.

# Sorting Algorithms: Merge Sort

For example, to sort the 9 elements of the vector **v** from **v[4]** to **v[12]** inclusive, we split this by computing (4+12)/2 = 8 and sorting **v[4]** through **v[8]** separately from **v[9]** through **v[12]**. Then we merge these two sorted lists.

Or to sort the 4 elements from **v[9]** through **v[12]** inclusive, we split at (9+12)/2 = 10 and sort **v[9]** through **v[10]** separately from **v[11]** through **v[12]**. Then we merge these two sorted lists.

To accomplish this task, we will need a function **mergeSorted** that merges two separate, sorted lists.

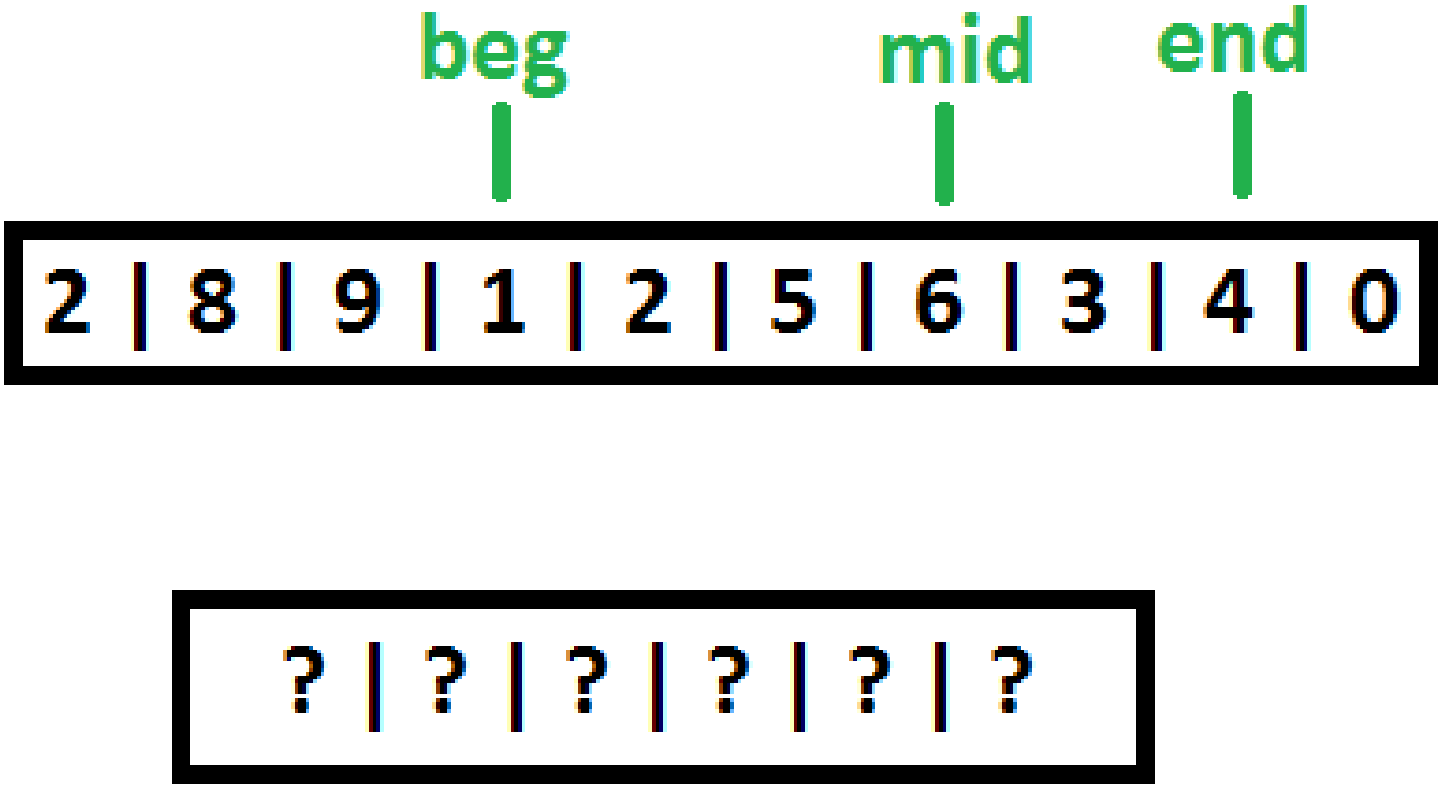
# Sorting Algorithms: Merge Sort

The function **mergeSorted** is declared below in a form that works to merge two consecutive ranges of a vector into a sorted order.

```
/**
This function merges two consecutive ranges of a vector that are themselves already sorted, and overwrites the vector elements so that all the elements in the overall range are in order.
@param v the vector that is being sorted on a range
@param beg the lowest index range to be included in the first list
@param mid the highest index range to be included in the first list with mid+1 being the lowest index range to be included in the second list
@param end the highest index range to be included in the second list
*/
void mergeSorted(std::vector<int>& v, size_t beg, size_t mid, size_t end);
```

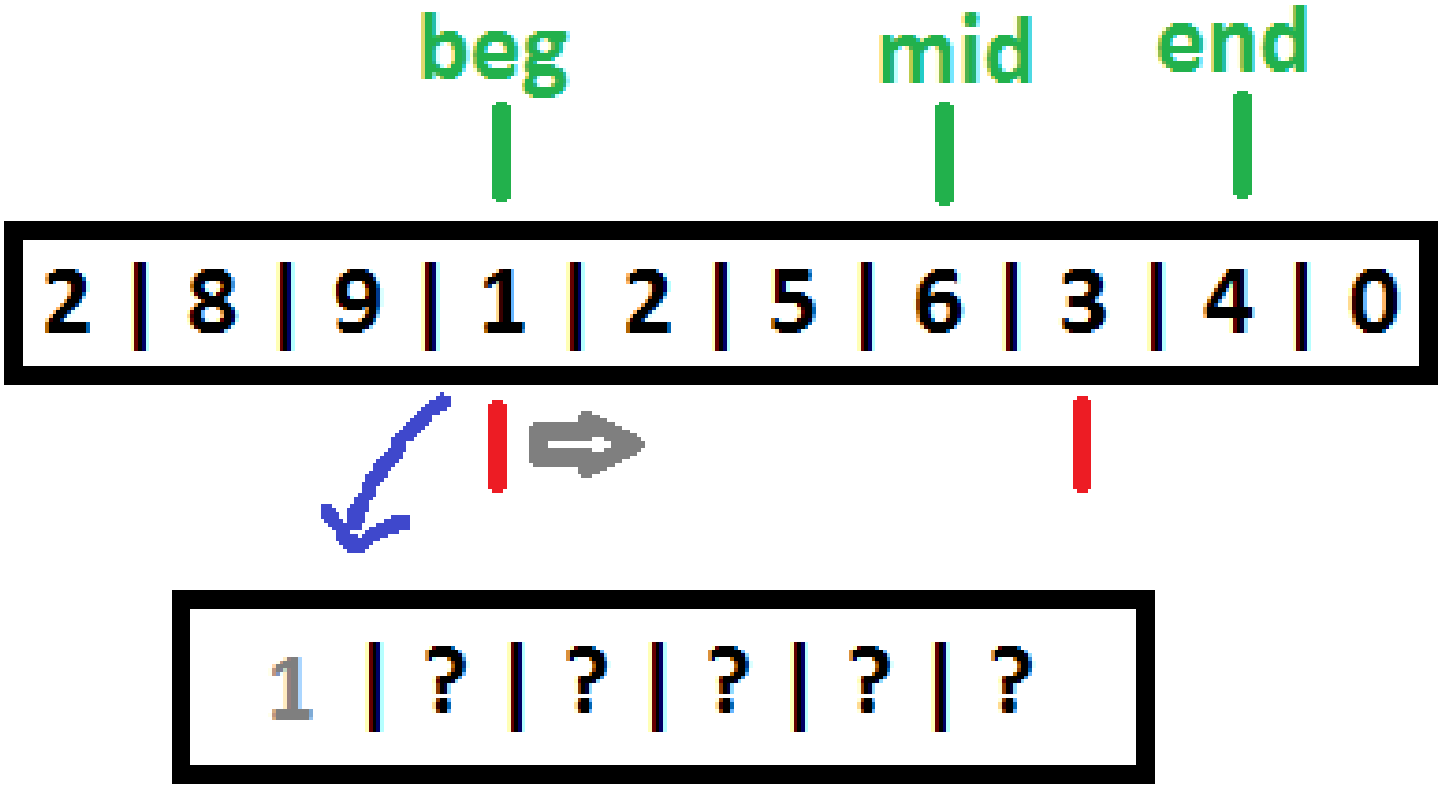
# Sorting Algorithms: Merge Sort

A cartoon depiction of the algorithm is below: elements of the two parts are compared and the smaller is added to a temporary vector and the appropriate index of the first/second list increases; when a list is exhausted, all the remaining elements are added. Then the temporary elements overwrite the range of values to be merge sorted.



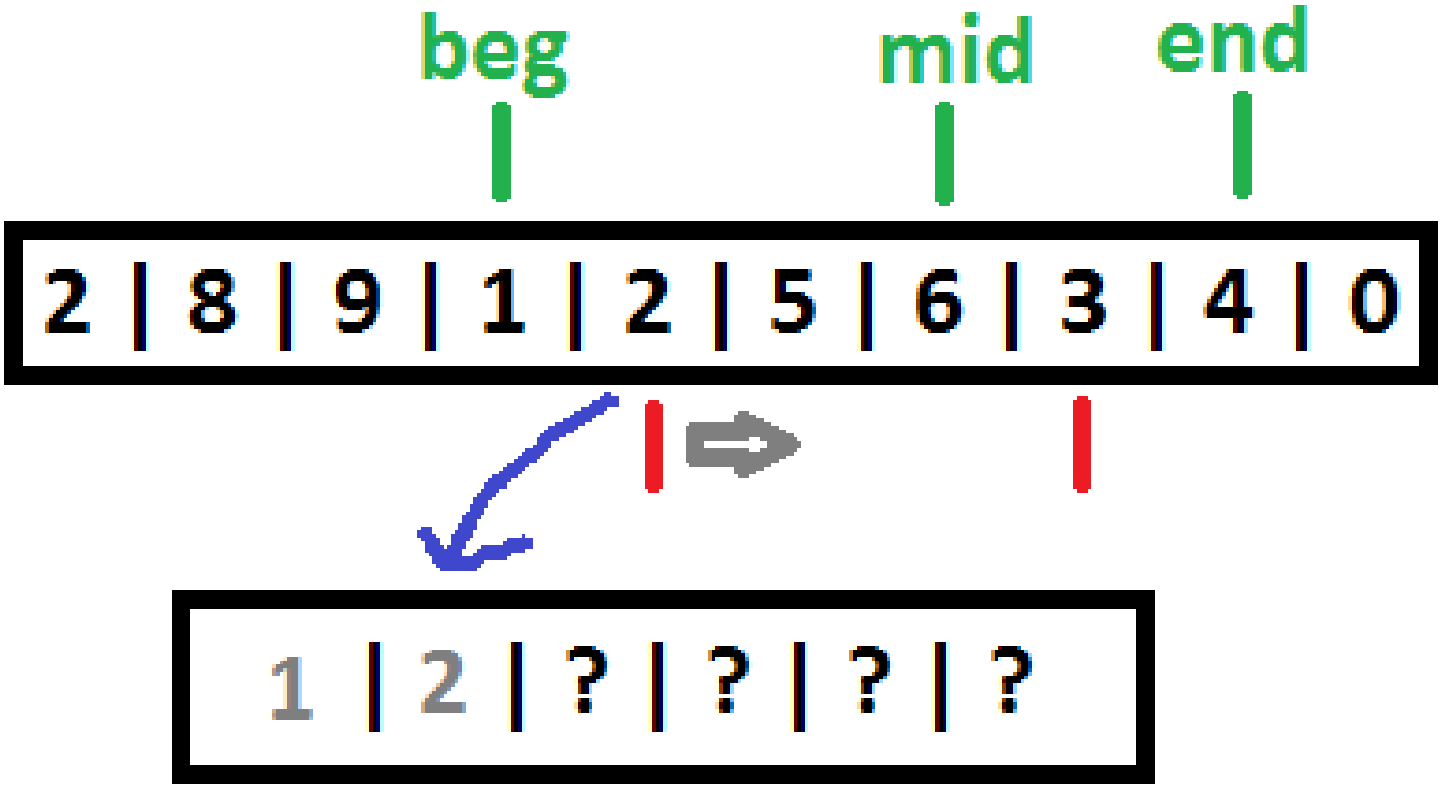
# Sorting Algorithms: Merge Sort

A cartoon depiction of the algorithm is below: elements of the two parts are compared and the smaller is added to a temporary vector and the appropriate index of the first/second list increases; when a list is exhausted, all the remaining elements are added. Then the temporary elements overwrite the range of values to be merge sorted.



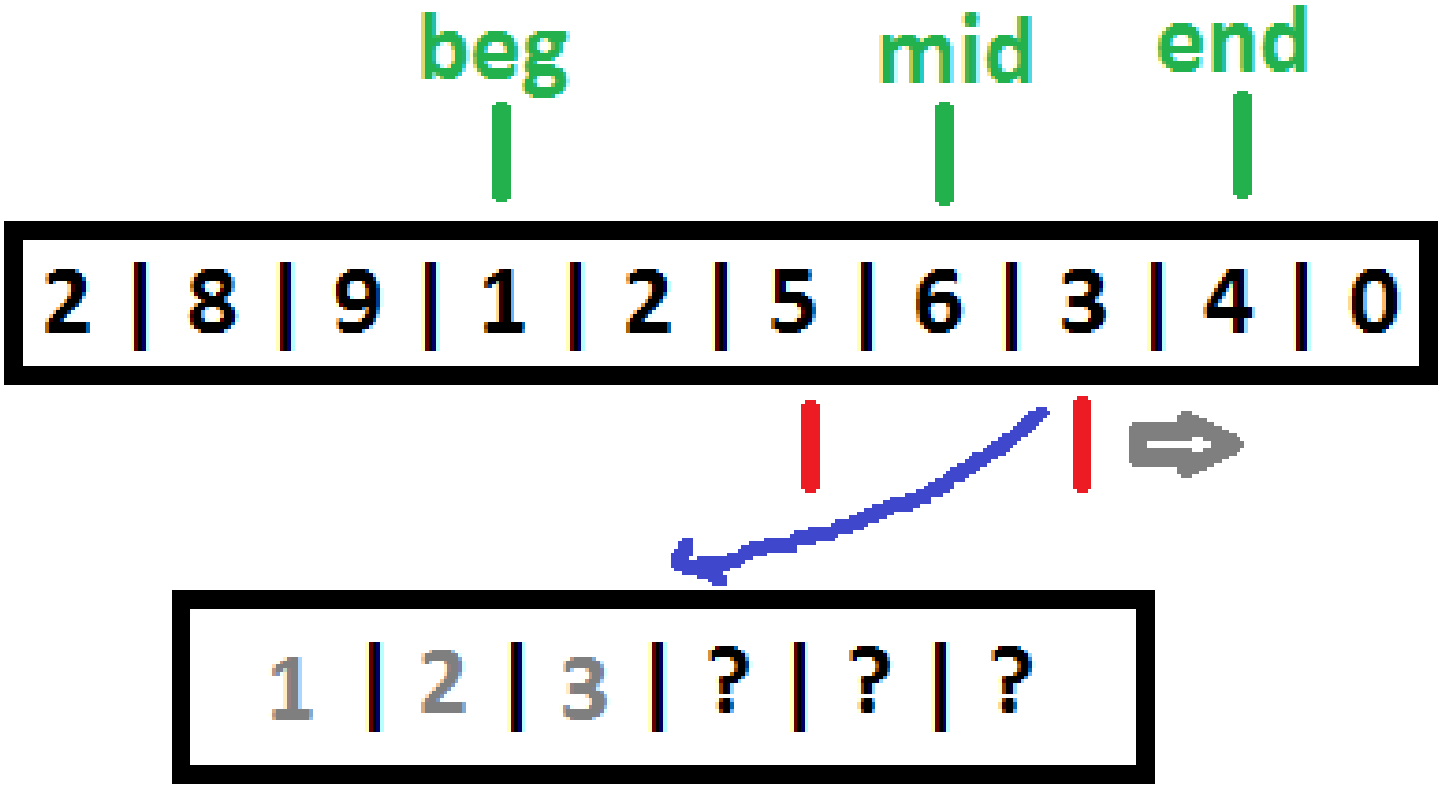
# Sorting Algorithms: Merge Sort

A cartoon depiction of the algorithm is below: elements of the two parts are compared and the smaller is added to a temporary vector and the appropriate index of the first/second list increases; when a list is exhausted, all the remaining elements are added. Then the temporary elements overwrite the range of values to be merge sorted.



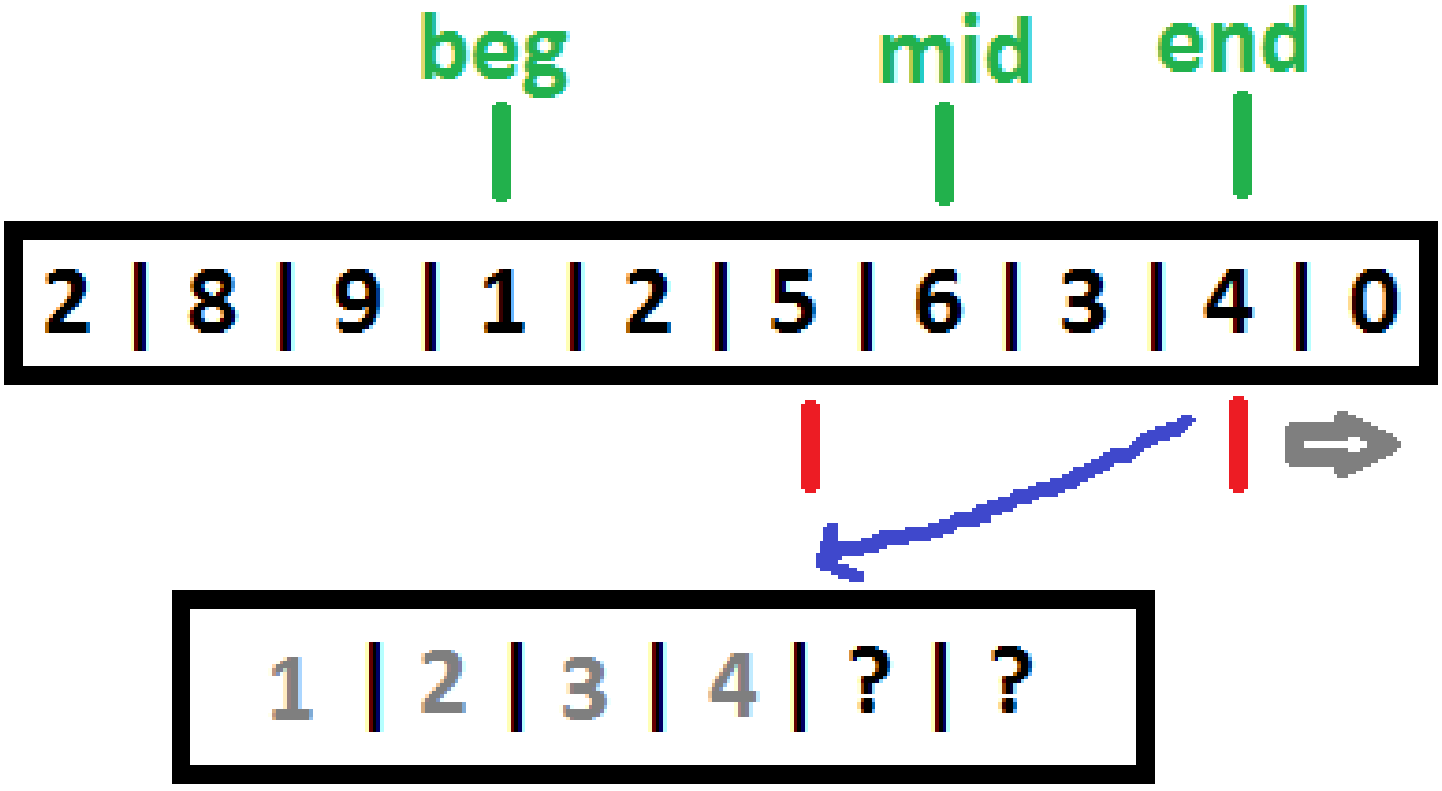
# Sorting Algorithms: Merge Sort

A cartoon depiction of the algorithm is below: elements of the two parts are compared and the smaller is added to a temporary vector and the appropriate index of the first/second list increases; when a list is exhausted, all the remaining elements are added. Then the temporary elements overwrite the range of values to be merge sorted.



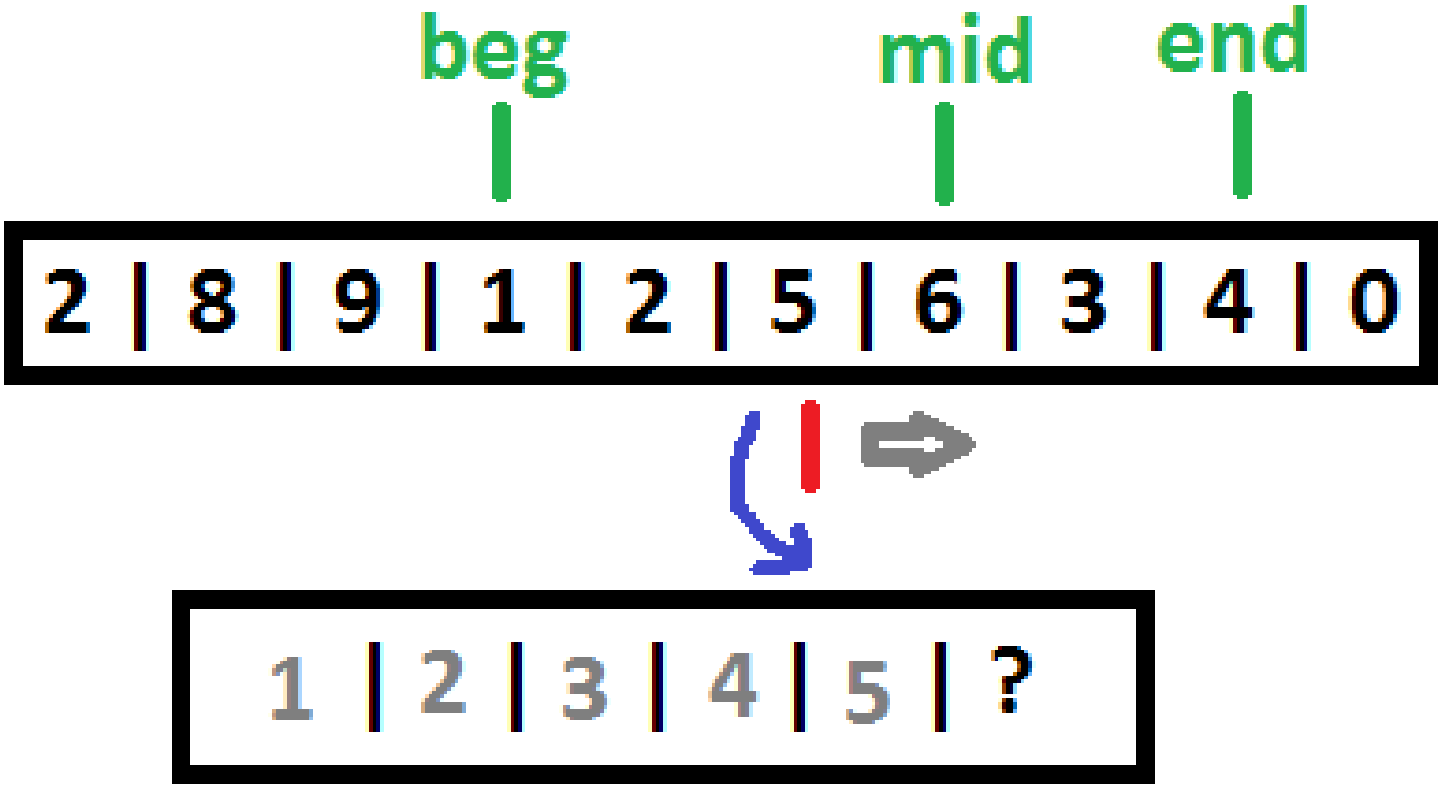
# Sorting Algorithms: Merge Sort

A cartoon depiction of the algorithm is below: elements of the two parts are compared and the smaller is added to a temporary vector and the appropriate index of the first/second list increases; when a list is exhausted, all the remaining elements are added. Then the temporary elements overwrite the range of values to be merge sorted.



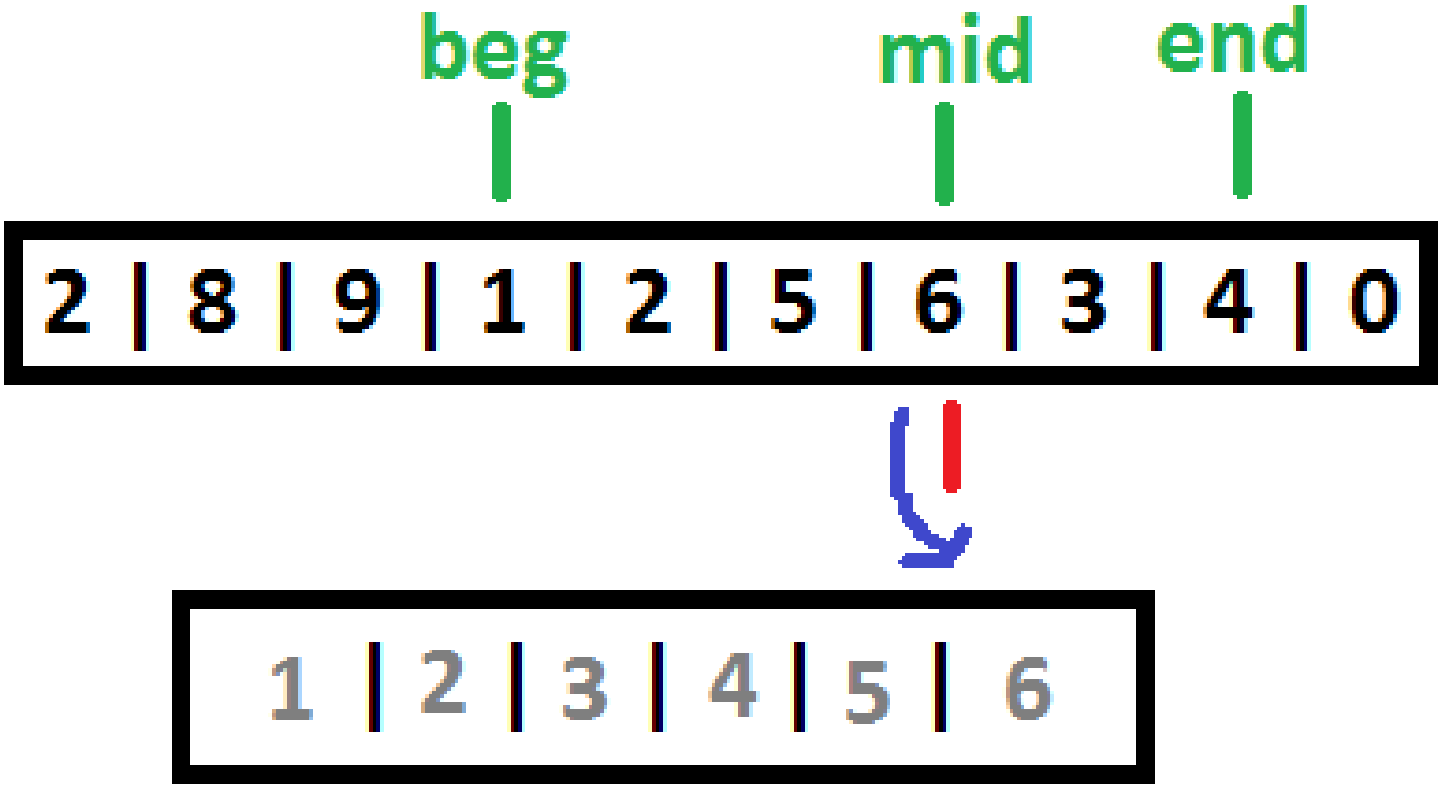
# Sorting Algorithms: Merge Sort

A cartoon depiction of the algorithm is below: elements of the two parts are compared and the smaller is added to a temporary vector and the appropriate index of the first/second list increases; when a list is exhausted, all the remaining elements are added. Then the temporary elements overwrite the range of values to be merge sorted.



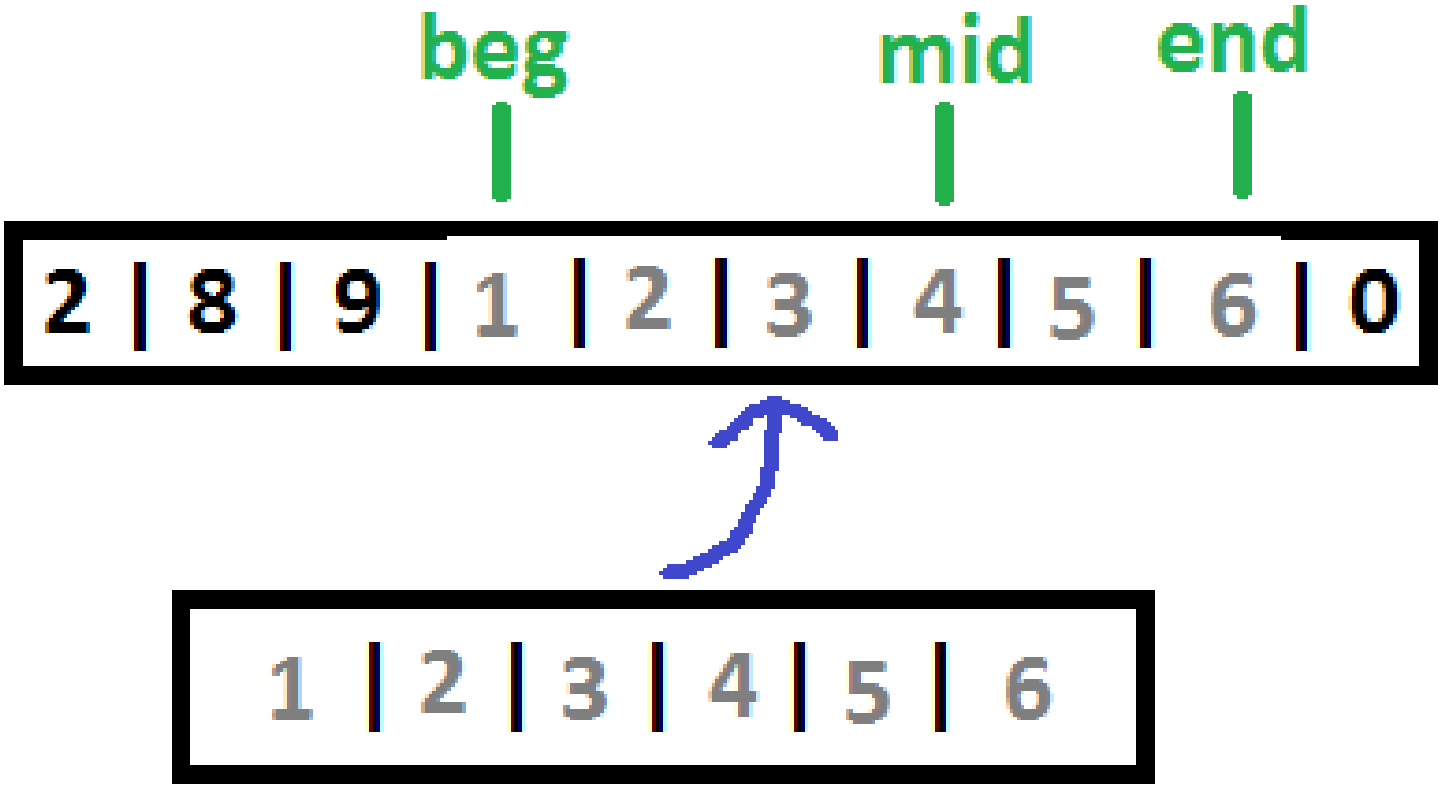
# Sorting Algorithms: Merge Sort

A cartoon depiction of the algorithm is below: elements of the two parts are compared and the smaller is added to a temporary vector and the appropriate index of the first/second list increases; when a list is exhausted, all the remaining elements are added. Then the temporary elements overwrite the range of values to be merge sorted.



# Sorting Algorithms: Merge Sort

A cartoon depiction of the algorithm is below: elements of the two parts are compared and the smaller is added to a temporary vector and the appropriate index of the first/second list increases; when a list is exhausted, all the remaining elements are added. Then the temporary elements overwrite the range of values to be merge sorted.



# Sorting Algorithms: Merge Sort I

```
void mergeSorted(std::vector<int>& v, size_t beg, size_t mid, size_t end){
   /* set positions of first and second parts of vector and create vector to
   store the result */

   // first index, second index, temporary vector's index
   size_t first = beg, second = mid+1, vecIndex = 0;

   std::vector<int> res(end-beg+1); // size due to inclusive range of values

   while ( (first <= mid) && (second <= end) ){ // while both in range

      // check if first value pointed to is less than second
      if (v[first] < v[second]) {
         /* if so, store first value in vector and increment first index and
            vector index */
         res[vecIndex++] = v[first++];
      }
```

# Sorting Algorithms: Merge Sort II

```
    else {
        /* if not, store the second value in vector and increment second
        index and vector index */
        res[vecIndex++] = v[second++];
}

// only one of the two while loops below will run

while (first <= mid) { // while first in range
    res[vecIndex++] = v[first++];
}

while (second <= end) { // while second in range
    res[vecIndex++] = v[second++];
}
```

# Sorting Algorithms: Merge Sort III

```
    // now copy over correctly ordered values to v
    size_t resSize = res.size();
    for (vecIndex = 0; vecIndex < resSize; ++vecIndex) {
        v[beg + vecIndex] = res[vecIndex];
    }
}
```

# Sorting Algorithms: Merge Sort

The function **mergeSorted** creates a vector to store the proper ordering and moves an index through both sorted lists.

Depending on which list index corresponds to the smaller value, that value is added to the correctly ordered vector and both the index of the container with the smaller element and the index of the vector storing the sorted lists is incremented.

This is done until one of the two lists is exhausted.

Then, the remaining elements are added to the vector with proper ordering. After this, the elements of the original vector are overwritten by the correctly ordered elements.

# Sorting Algorithms: Merge Sort I

With **mergeSorted** written, the actual **mergeSort** is quite a simple.

```
/** This will sort a vector via merge sort
@param v the vector of ints to sort
@param beg the lowest index position to include in the sort
@param end the largest index position to include in the sort
*/
void mergeSort(std::vector<int>& v, size_t beg, size_t end){

   if (beg == end) { // nothing to do in this case
      return;
   }

   size_t mid = (beg+end) / 2; // find middle position with int division
```

# Sorting Algorithms: Merge Sort II

```
    mergeSort(v, beg, mid); // sort the first half

    mergeSort(v, mid + 1, end); // sort the second half

    // with first and second halves sorted, the two lists can be merged

    mergeSorted(v, beg, mid, end);
}
```

# Sorting Algorithms: Merge Sort

We can use **static** variables to track how many times a function is called. Like static member variables, a local static member variable is stored in the text segment of memory. The local variable **counter** will be initialized only once, not in every function call.

```
/**
This function increments a count when told to and returns its count value
@param increase whether or not to increment the counter
@return the count value
*/
int count(bool increase) {
    static int counter = 0; // initialized to 0 only once
    if(increase) { // if should increase
        ++counter; // then increase
    }
    return counter;
}
```

# Sorting Algorithms: Merge Sort

We could count how many times **mergeSort** gets called by inserting:

```
void mergeSort(std::vector<int>& v, size_t beg, size_t end){

    count(true); // count each time this function is entered

    // rest of mergeSort code...
}
```

Then, to check the count, we could simply

```
std::cout << count(false);
```

# Searching Algorithms: Binary Search

Consider a sorted **std::vector<int>** (or a vector of any other type with **operator<** and **operator==**), sorted in increasing order, and suppose we wish to determine if a given value is an element in the vector.

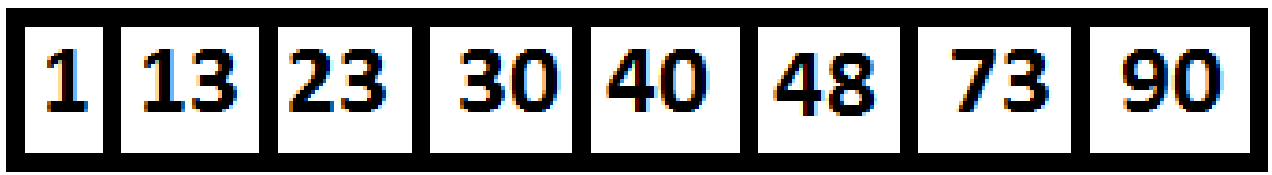
We can employ a “divide and conquer” approach: start in the middle. If the middle element equals the desired value, we’re done.

If the middle is less than the desired value, we repeat the entire search on the upper half; otherwise, we repeat the entire search on the lower half.

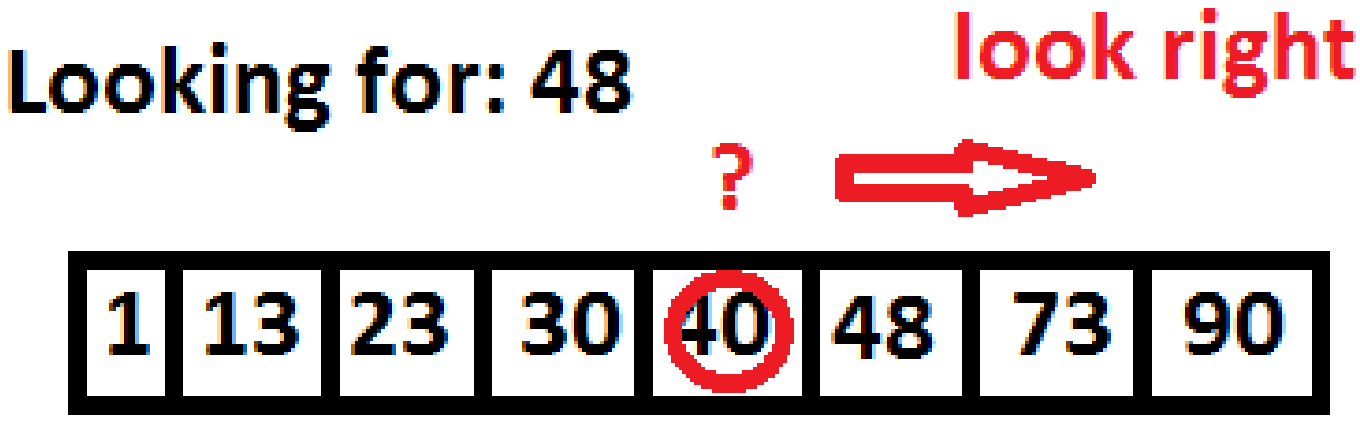
We will write code to search through the sorted vector and return **true** if the value is found and otherwise return **false**.

# Searching Algorithms: Binary Search

**Looking for: 48**

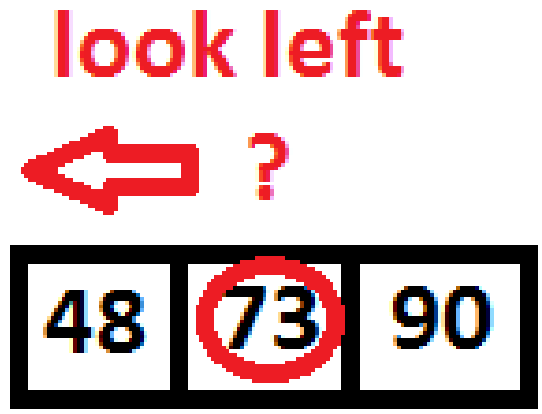


# Searching Algorithms: Binary Search



# Searching Algorithms: Binary Search

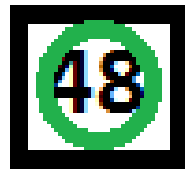
**Looking for: 48**



# Searching Algorithms: Binary Search

**Looking for: 48**

found it!



# Searching Algorithms: Binary Search I

```
/**
Searching if a value is found over a range of iterators.
@param beg the first iterator
@param past_end the iterator just past the search range
@param val the value to search for
@return whether the value is found within range
*/
bool binSearch(const std::vector<int>::const_iterator beg,
    const std::vector<int>::const_iterator past_end, int val) {

    if (beg == past_end) { // no range to search over
        return false; // so not found
    }

    if (beg == (past_end-1) ) { // only one item in range
        return *beg == val; // whether element is value
    }
```

# Searching Algorithms: Binary Search II

```
auto len = past_end - beg; // number of elements
auto mid = beg + len/2; // move half-range past left to midpoint

if(*mid == val) { // check midpoint
    return true; // and return true if it stores value
}

else if(val > *mid) { // must search right half if val bigger than mid-value
    return binSearch(mid+1, past_end, val);
}
```

# Searching Algorithms: Binary Search III

```
    /* must search left half if val less than mid-value: note that mid is not
    included again in search range because the upper bound is not
    included in the search */

    else { // must search left half if val less than mid-value
        return binSearch(beg, mid, val);
    }
}
```

# Searching Algorithms: Binary Search

The local type of the iterator is **const_iterator**, ensuring that if we perform this function upon a **const std::vector<int>**, it can be done.

It is **const** because we don't want to accidentally change **beg** or **end** during the logic of checking values, etc. It is not a reference because an iterator is a lightweight object, effectively just storing a pointer, so little is gained in efficiency, if anything, by passing it as a reference.

# Searching Algorithms: Binary Search

In running this function, we must have the vector in sorted order.

```
std::vector<int> v { 1, 3, 8, 8, 9 };
binSearch(std::begin(v), std::end(v), 9); // == true
binSearch(std::begin(v), std::end(v) - 1, 9); // == false
binSearch(std::begin(v), std::end(v), 3); // == true

const std::vector<int> w = v;
binSearch(std::begin(w), std::end(w), 4); // == false
```

Because **w** is constant, **std::begin(w)** and **std::end(w)** return **std::vector<int>::const_iterator**s.

# Searching Algorithms: Binary Search

There is iterator arithmetic: in subtracting two random access iterators, the result is of type **std::vector<int>::difference_type** (this is what **len** really is), which is a signed integer type representing the difference in location between two iterators.

Arithmetic can be done on this integer type, and when adding this type to an iterator, we advance or go back as many steps as specified. Thus, **mid** is still a **std::vector<int>::const_iterator** type.

# Costing of Algorithms: Importance

For small sets of data, most reasonable algorithms perform comparably in terms of the time they take. But for large sets of data, the time expended by an algorithm becomes an important deciding factor in what algorithm to choose or implement.

We often describe the **cost of an algorithm** (how long it takes or how many steps must be done) in **big-O notation**, such as saying an algorithm is $O(n^2)$ or $O(n \log n)$ with respect the size of the input $n$.

The big-O notation describes the dominant component to the time or number of steps of the algorithm: what sort of dependence we would observe if $n$ were large. Being $O(n^2)$ roughly means that the time it takes for the algorithm to run is proportional to $n^2$, etc.

# Costing of Algorithms: Importance

When trying to pick out the big-O behaviour, we need only concern ourselves with the largest term, ignoring all other terms. In addition, constant pre-factors are ignored completely. Suppose that $n$ is a very large number ($n \to \infty$, say):

$11n^3 = O(n^3)$: pre-factor of 11 ignored: the term is overall cubic in $n$. Every time the number of data points (or whatever $n$ represents) doubles, the algorithm takes $2^3 = 8$ times longer!

$n + 144 = O(n)$: when $n$ is large, 144 is negligible so overall we only see the $n$

# Costing of Algorithms: Importance

$5n^2 + 2n + 10 = O(n^2)$: the $2n$ and 10 are negligible compared to $5n^2$ and we ignore the pre-factor.

In being $O(n^2)$, every time $n$ is *doubled*, the overall cost goes up by roughly a *factor of 4*.

If $n = 100$, $5n^2 + 2n + 10 = 50210$...

If $n = 200$, $5n^2 + 2n + 10 = 200410$, approximately 4 times that of when $n = 100$...

# Important Big-O Results

In costing algorithms, it is important to understand the relative sizes of different terms. In general we have, in increasing dominance,

$$O(1) \ll O(\log n) \ll O(n^p) \ll O(b^n)$$

where $p > 0$ and $b > 1$.

So an algorithm that has a cost of $O(n^3)$ is much faster (for large $n$) than an algorithm that has a cost $O(2^n)$, etc.

*Remark:* the Tower of Hanoi is $O(2^n)$ which is terribly costly to solve. But without other options to solve it, we have to live with it.

# Important Big-O Results

Recall that all logs are related by a multiplicative constant hence,

$O(\log_2 n) = O(\log_{10} n) = O(\log_e n) = ...$

We generally just write $O(\log n)$ where log is often viewed as base e $= 2.71828182846...$, the base of natural logarithms, but the base doesn't really matter.

# Important Big-O Results

Note $\log$ could have any base and grows *much slower* than any positive power of $n$. Thus, an algorithm that is $O(\log n)$ vastly outperforms an algorithm that is $O(n^2)$ or even $O(n)$.

All powers of $n$ grow *much slower* than an exponential $b^n$ for any $b > 1$. Thus, even an algorithm that is $O(n^9)$ would outperform an algorithm that is $O(2^n)$ for $n$ large enough.

An $O(1)$ algorithm *is the best*: no dependence upon $n$.

In practice $O(\log n)$ performs about the same as $O(1)$ because $\log$ grows so slowly.

# Costing of Algorithms: Estimating

The time it takes an algorithm to run depends upon the number of times a chunk of memory is accessed and/or modified, among possibly other factors.

For simplicity, we assume that every time a vector is indexed or an arithmetic operation is performed, this takes the same amount of time and treat the time of passing parameters to another function as negligible. *This may not actually be true and it does take time to pass parameters!*

In general for costing, however, absolute precision is seldom required.

# Costing of Algorithms: Estimating

When we look at an algorithm with an input size of $n$, we wish to estimate $T(n)$, the time/number of steps required to process the input.

For very general algorithms, this can be difficult to estimate, but there are some tricks we can consider.

# Costing of Algorithms: Palindrome

Consider the recursive implementation of **isPalindrome**.

```
bool isPalindrome(const std::string_view& s) {
    if ( s.size() <= 1 ) {
        return true;
    }
    else {
        if ( s[0] != s[s.size()-1] ) {
            return false;
        }
    }
    return isPalindrome( s.substr(1,s.size()-2) );
}
```

# Costing of Algorithms: Palindrome

Let $T(n)$ denote the number of steps required in evaluating the function on a string of size $n$. Then for $n > 1$, i.e. not a base case, we expect the time $T(n)$ to satisfy the **recurrence relation**

$$T(n) = 10 + T(n-2)$$

where the 10 represents:

- the time to check the size of the vector and
- compare it against the value 1,
- subscript the first element of the vector,
- subscript the last element of the vector which also includes
- computing the size of the vector and
- subtracting one from it and then
- comparing those values,
- then we call size,
- subtract 2 from it
- and invoke substr (pretend it's one step)

and the $T(n-2)$ is the time it takes to run the same algorithm on the shortened string.

# Costing of Algorithms: Palindrome

Since $T(n) = 10 + T(n-2)$ we can write a *telescoping sum*:

$$
\begin{aligned}
T(n) = T(n) \quad & -T(n-2) \\
& +T(n-2) - T(n-4) \\
& \qquad + T(n-4) \qquad -T(n-6) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
& \qquad\qquad\qquad\qquad\qquad\qquad - T(x) \\
& \qquad\qquad\qquad\qquad\qquad\qquad + T(x)
\end{aligned}
$$

$$
T(n) = \overbrace{10 + 10 + \ldots + 10}^{\text{floor(n/2)times}} + T(x)
$$

where $x = 0$ if $n$ is even and $x = 1$ if $n$ is odd.

# Costing of Algorithms: Palindrome

We thus have for $n > 1$ that

$$T(n) = \begin{cases} 5n + T(0), & \text{n even} \\ 5(n-1) + T(1), & \text{n odd.} \end{cases}$$

$$= \begin{cases} 5n, & \text{n even} \\ 5(n-1), & \text{n odd} \end{cases} + 1$$

where $T(0)$ and $T(1)$ are both constants and equal to 1. Thus, the overall complexity is $O(n)$.

# Costing of Algorithms: Palindrome I

The **isPalindrome** also had a non-recursive implementation. If the **std::string** size is 0 or 1 then it may cost 2 steps, say, just the cost of checking the size of the string and comparing it to 1.

```
bool isPalindrome(const std::string& s) {
    if ( s.size() <= 1 ) {
        return true;
    }

    for ( std::size_t low = 0, up = s.size()-1; low <= up; - - up, ++low ) {
        if ( s[low] != s[up] ) {
            return false;
        }
    }
    return true;
}
```

# Costing of Algorithms: Palindrome II

For $n > 2$, there is this minimal 2-step cost plus $\approx 4$ steps before commencing the loop.

The loop should run $n/2$ times if $n$ is even and $(n+1)/2$ times if $n$ is odd.

Within the loop, we need to access two values of the string and compare them, increase and decrease the indices, and compare the indices, 6 operations, say. Then we have

$$T(n) = 6 + \begin{cases} 6n/2, & \text{even} \\ 6(n+1)/2, & \text{odd.} \end{cases}$$

This, like the recursive case, is $O(n)$.

# Costing of Algorithms: Palindrome III

It appears the number of steps in the recursive and non-recursive versions are *not the same*; but they can still have the same *big-O runtime*, i.e., linear in the length of the string.

**Note:** although two algorithms can have the same cost in big-O, they may have different precise runtimes/numbers of steps.

# Costing of Algorithms: Palindrome

We could get away with being a bit more abstract and still obtain the same result. Suppose $s_1$ represents the time/number of steps to compare the size of the vector against 1 and $s_2$ represents the time/number of steps inside each loop.

We could then say that

$$T(n) = s_1 + \begin{cases} s_2 n/2, & \text{even} \\ s_2(n+1)/2, & \text{odd} \end{cases}$$

and since $s_1$ and $s_2$ are just numbers independent of $n$, this is still $O(n)$.

# Costing of Algorithms: Merge Sort

Both bubble and selection sort can be proven to be $O(n^2)$ algorithms (by noting that $T(n) = cn + T(n-1)$, using the *telescoping strategy* along with the fact that $1 + 2 + ... + n = n(n+1)/2$), which makes them very slow. It turns out, we can get significantly better performance with merge sort.

Let $T(n)$ be the time to sort a vector of size $n$ and suppose that $n = 2^k$ for some integer $k$. At the very least, it is true that $2^k \leq n \leq 2^{k+1}$ for some $k$.

This $n = 2^k$ assumption makes the division of the sorting list nicer because each division will have the same number of elements.

# Costing of Algorithms: Merge Sort

We should have that

$$T(n) = \overbrace{T(\frac{n}{2})}^{\text{sort n/2 items}} + \overbrace{T(\frac{n}{2})}^{\text{sort n/2 items}} + \overbrace{cn}^{\text{merge 2 lists of n/2 items}} + \overbrace{d}^{\text{logical checks}} .$$

The time it takes to sort $n$ elements is the time it takes to sort each of the smaller sets of elements of size $n/2$, plus the time it takes to merge the two lists of size $n/2$ and the time it takes to do some comparisons and math with iterators.

This merging process should be proportional to $n$ since merging effectively runs two loops with $n/2$ iterations in each.

# Costing of Algorithms: Merge Sort

If $T(n) = 2T(\frac{n}{2}) + cn + d$ then
$T(\frac{n}{2}) = 2T(\frac{n}{4}) + c\frac{n}{2} + d$ and
$T(\frac{n}{4}) = 2T(\frac{n}{8}) + c\frac{n}{4} + d$ and so on...

Then,

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + cn + d = 2(2T(\frac{n}{4}) + c\frac{n}{2} + d) + cn + d \\
&= 4T(\frac{n}{4}) + 2cn + 3d = 4(2T(\frac{n}{8}) + c\frac{n}{4} + d) + 2cn + 3d \\
&= 2^3 T(\frac{n}{2^3}) + 3cn + 7d = 2^3(2T(\frac{n}{2^4}) + c\frac{n}{2^3} + d) + 3cn + 7d \\
&= \ldots \\
&= 2^k T(\frac{n}{2^k}) + kcn + (2^k - 1)d \\
&= nT(1) + c(\log_2 n)n + (n-1)d
\end{aligned}
$$

where we used that $n = 2^k$ and $k = \log_2 n$ in the final equality. Thus, since $n \log_2 n$ dominates over $n$, we have that $T(n) = O(n \log n)$.

# Costing of Algorithms: Binary Search

With binary search, through each recursive step, the size of the list of items to search through effectively goes down by a factor of 2.

With an approximately constant overhead $c$ within each iteration, we have a recurrence relation

$$T(n) = T(\frac{n}{2}) + c.$$

If $n = 2^k$ then we have, after working with the recurrence relation, that

$$T(n) = T(1) + ck = O(\log n)$$

is the cost of binary search.

# Costing of Algorithms: Sequential Search

There is also a rather obvious but slow search algorithm whereby we look through the vector in sequential order until we possibly find a match.

If the vector has $n$ elements then we expect $O(n)$ for runtime because we have to check up to $n$ elements.

# Costing of Algorithms: Sequential Search

```
/**
This method does a sequential search on a container
@param beg the beginning vector iterator
@param past_end iterator just past the end of range
@val the value sought after
@return whether it was found
*/
bool seqSearch(std::vector<int>::const_iterator beg,
    const std::vector<int>::const_iterator& past_end, int val) {
    while(beg != past_end) { // while not past valid range
        if(*beg == val) { // check value
            return true; // if equal then match found
        }
        ++beg;
    }
    return false; // no match found if beg reaches end
}
```

# Aside: Lower and Upper Bounds to Costs

We have (and will continue) to avoid the more general question of listing lower and upper bounds for the number of steps an algorithm takes.

In more a more general theory of algorithms, we could use notation

$T(n) = \Omega(1)$ to indicate the algorithm takes at least a time $O(1)$ (in the case **isPalindrome** is done in a step or two, without having to read through many of the characters).

$T(n) = \Theta(n)$ to indicate the smallest upper bound on runtime is $O(n)$, in the worst case when almost all the characters need to be checked.

# Aside: Mutual Recursion and Parsing a Math Expression

As a further example of recusion, we will process an expression such as **(-x*3+1.1)*2+2/x+4**
as an actual function $f(x) = 2(-3x + 1.1) + \frac{2}{x} + 4$.

For example, if $x = 2$ then $f(x)$ is $2 \times (-4.9) + 2/2 + 4 = -4.8$.

We'll assume the user can enter anything with **+, -, *, /, (,)** for operations and **x** will denote a symbolic variable. The user can also enter decimal values. No space characters are permitted.

# Aside: Mutual Recursion and Parsing a Math Expression

To accomplish our task, we will:

- use **mutual recursion** (whereby functions involved in recursion call each other) to parse the expression adhering to order of operations and
- use the **std::function** templated class to store our function.

# Aside: Mutual Recursion and Parsing a Math Expression

A few important observations:

- the function $f(x) = x$, i.e., the identity, is the lambda **[](double x)->double { return x; }**;
- the function $f(x) = c$, i.e., a constant function, is the lambda **[c](double x)->double{ return c; }**; and
- the zero function $f(x) = 0$ is the lambda **[](double x)->double{return 0;}**.

# Aside: Mutual Recursion and Parsing a Math Expression

We are going to be working with **std::function<double(double)>** functions, i.e., those that take an input argument of type **double** (the "x") and output a **double**.

Writing **std::function<double(double)>** is a lot... so let's abbreviate that via

```
using mathFun = std::function<double(double)>;
```

# Aside: Mutual Recursion and Parsing a Math Expression

Mathematically, we can add, subtract, multiple, or divide functions so we assume that we have overloaded operators such as:

**mathFun operator+(const mathFun&, const mathFun&);**

,etc.

# Aside: Mutual Recursion and Parsing a Math Expression

Here's the idea involving recursion:

**(-x*3+1.1)*2 + 2/x + 4**

- To evaluate the **expression**, we wish to evaluate each **term** and combine them with **+** and **-.**
  The terms are *(-x*3+1.1)*2*, *2/x*, and *4*.
  - To evaluate a **term**, we wish to evaluate each of their **factor**s and combine them with * and /.
    The *(-x*3+1.1)*2* has factors of *(-x*3+1.1)* and *2*; the *2/x* has factors of *2* and *x*; the *4* has only itself as a factor.
    - To evaluate a **factor**, we can directly interpret the result if it is simple enough; otherwise, we need to evaluate the **factor** as an **expression**.
      *-x*3+1.1* must be parsed as a combination of terms; the *2*, *4*, and *x* can be handled directly.

# Aside: Mutual Recursion and Parsing a Math Expression I

Now we can write our code to parse a function written in a stream.

```
// declare all functions so they can be referenced before their definitions

mathFun combineTerms(std::istream& in);
mathFun combineFactors(std::istream& in);
mathFun evaluateFactor(std::istream& in);
```

# Aside: Mutual Recursion and Parsing a Math Expression II

```
// define combineTerms
mathFun combineTerms(std::istream& in) {
    mathFun res = combineFactors(in); // evaluate term

    bool more = true;
    while (more) { // while more to read signified by + and -

        char op = in.peek(); // check if next char is a + or -

        if (op == '+' || op == '-') { // if add or subtract
            in.get(); // remove the operator

            mathFun val = combineFactors(in); // and evaluate the next term
```

# Aside: Mutual Recursion and Parsing a Math Expression III

```
        if (op == '+') { // add terms
            res = res + val;
        }
        else { // subtract terms
            res = res - val;
        }
    }

    else { // if no more + or - then done!
        more = false;
    }
  }

  return res; // return the result
}
```

# Aside: Mutual Recursion and Parsing a Math Expression IV

```
// define combineFactors
mathFun combineFactors(std::istream& in) {
    mathFun res = evaluateFactor(in); // evaluate term

    bool more = true;
    while (more) { // while more to read signified by * and /

        char op = in.peek(); // check if next char is a * or /

        if (op == '*' || op == '/') { // if times or divide
            in.get(); // remove the operator

            mathFun val = evaluateFactor(in); // and evaluate the next term
```

# Aside: Mutual Recursion and Parsing a Math Expression V

```
        if (op == '*') { // multiply factors
            res = res * val;
        }
        else { // divide factors
            res = res / val;
        }
    }

    else { // if no more * or / then done!
        more = false;
    }
  }

  return res; // return the result
}
```

# Aside: Mutual Recursion and Parsing a Math Expression VI

```
// define evaluateFactor
mathFun evaluateFactor(std::istream& in) {

    // set res to zero function f(x) = 0
    mathFun res = [](double x)->double{ return 0; };

    char c = in.peek(); // peek at char

    /* if c is a bracket, extract it and evaluate the terms within, then remove
    the closing bracket */

    if (c == '('){
        in.get(); // remove (
        res = combineTerms(in); // compute sum/difference of terms
        in.get(); // remove )
    }
```

# Aside: Mutual Recursion and Parsing a Math Expression VII

```
else if (c == 'x') { // case of the x function, i.e. f(x)=x

    in.get(); // remove the char x
    res = [](double x){ return x; }; // set the function
}

else if ( c == '+' || c == '-' ) { // just a sign bit
    ; /* EMPTY STATEMENT: leave res as 0 function */
}
```

# Aside: Mutual Recursion and Parsing a Math Expression VIII

```
  else { // then constant numeric function, i.e. f(x)=number

    // initialize to bogus largest double value
    double ans = std::numeric_limits<double>::max();

    in >> ans; // read from stream

    res = [ans](double x)->double{ return ans; };
  }

  return res;
}
```

# Aside: Mutual Recursion and Parsing a Math Expression

With this code, we can write a very simple main routine and interpret the user's inputs!

```
// stuff
int main() {
    std::cout << "Enter a function of x, f(x)=";
    auto f = combineTerms(std::cin);
    std::cout << "Enter a number: ";
    double d = std::numeric_limits<double>::max();
    std::cin >> d;
    std::cout << "f("<<d <<")="<<f(d);
    return 0;
}
```

```
Enter a function of x, f(x)=(-x*3+1.1)*2+2/x+4
Enter a number: 2
f(2)=-4.8
```

# Aside: Mutual Recursion and Parsing a Math Expression

**Remarks:**

Note the lambda needed to capture the numeric value **res** in order to work:
**res = [ans](double x)->double{ return ans; };**

Recall the **peek** member function returns the next **char** from the stream, but does not extract it, hence “peek” and not “get”.

# Aside: Mutual Recursion and Parsing a Math Expression

Note that in parsing the purely numeric input, it is necessary to consider the case that nothing can be read: consider reading in “+x”.

We know that **evaluateFactor** will be called on the stream with “+x” and that **res** starts at **0**. And **0** should be returned and then **x** can be added to it. If we tried to read in a double out of **+x**, the stream would fail:
**iss >> ans;** // **where ans is some double**

We initialize **ans** to **std::numeric_limits<double>::max()**, the largest possible **double** value so that a failed input would be very easy to detect.

**max** is a public **static member function** of the templated class **std::numeric_limits** so we can invoke the function without an instance of the class even existing.

# Summary

- Many processes can be formulated in terms of recursion; the thought process may be more challenging, but recursion is extremely useful.
- Recursive functions should always have a termination point otherwise a program can stay stuck or overflow the stack.
- Functions can call each other in a recursive manner.
- The **std::function** template allows for the storage of function objects.
- In assessing “cost” for algorithms, we speak in big-O terms: the dominant qualitative behaviour of runtime for a given data set.
- Selection sort has a cost of $O(n^2)$ and repeatedly cycles through the unsorted part of a list, placing the smallest element first.
- Bubble sort has a cost of $O(n^2)$ and repeatedly moves forward through the unsorted part of a list, swapping adjacent items to put them in order.
- Merge sort has a cost of $O(n \log n)$ and repeatedly merges sorted lists formed by dividing the original list.
- Binary search has a cost of $O(\log n)$ and repeatedly decides which half of the list to search; sequential search has a cost of $O(n)$ and looks sequentially.

# Exercises I

1. Consider a class **node:**

```
struct node {
 double val = 0;
 node *child = nullptr;
};
```

Write a function **sum_nodes** that accepts a **node*** and returns the sum of its **val** and the **val**s of all of its "children". For example:

```
node first, second, third;
first.val = 1.1;
second.val = 2.2;
third.val = 3.3;
first.child = &second;
second.child = &third;
std::cout << sum_nodes(&first);
```

should output **6.6**. If the **node*** passed in has no non-null children, the sum should just be its **val**; and if the **node*** passed in is **nullptr**, the sum should be zero.

# Exercises II

2. Using $1 + 2 + ... + n = n(n+1)/2$, prove that both bubble sort and selection sort have a cost of $O(n^2)$.
3. Write a function **get_all** that accepts: a **std::vector** storing **int**s, and a predicate as a function pointer (in this case the predicate has signature **bool(int)**); it should return a **std::vector**<**std::vector**<**int**>**::const_iterator**> storing all positions where the predicate is true. For example:

```
bool even(int i) { return (i % 2) == 0; }

// ...
std::vector<int> v {1,2,3,4,5};
auto res = get_all(v, even);
for(auto it :  res ) { std::cout <<*it <<''; }
```

should print

**2 4**

# Exercises III

4. **Quick Sort** is an algorithm that behaves similarly to merge sort. Here is the idea: given a list of size $n$, pick an item at random (or, in practice, just pick the leftmost/rightmost element). Call that value **val**. By passing through the list from one end to the other and performing swaps, ensure the chosen value is in its rightful place with all values to **val**'s left being less than or equal to **val** and all values to **val**'s right being greater than or equal to **val**. Repeat the process to sort the elements to **val**'s left and right.
   - Write a **recursive** implementation of this algorithm for a **std::vector**<**double**>.
   - Argue that on average, it has a time recurrence relation $T(n) = an + b + 2T(n/2)$ and determine the average cost.
   - Argue that at worst, it has a time recurrence relation $T(n) = an + b + T(n-1)$ and determine the worst case cost.

# Exercises IV

5. The Fibonacci sequence is defined as follows: start with values $a_0$ and $a_1$. Define $a_2 = a_0 + a_1$, $a_3 = a_1 + a_2$, and so on, with each new term being the sum of the two previous terms.
   - Write a function **fib_rec** that accepts **int** values for $a_0$, $a_1$, and an **int** $n \geq 0$ for how many terms to compute, which returns $a_n$. Write this function *recursively*.
   - Now write **fib_iter** with the same inputs and return value but with a direct, non-recursive implementation.
   - Argue that the recurrence relation for **fib_rec** obeys $T(n) \geq 2T(n-2)$ and use this to show that $T(n) = O(b^n)$ for some $b > 0$.
   - Show that the cost $T(n)$ for **fib_iter** is $O(n)$.
   - Which of **fib_rec** and **fib_iter** is more efficient?

# Exercises V

6. The **bisection method** is an algorithm in scientific computing to estimate the zeros of a function $f(x)$, i.e., the values of $x$ so that $f(x) = 0$. The process is as follows: choose a small tolerance threshold $\tau$ ($10^{-6}$, for example), a maximum number of iterations $N$ (100, say), and begin with a number $a$ so that $f(a) < 0$ and a number $b$ so that $f(b) > 0$. Define $c = (a + b)/2$.
   Until the algorithm has had $N$ iterations or until $|f(c)| < \tau$: if $f(a)f(c) < 0$ replace $b$ by $c$; otherwise replace $a$ by $c$. Update $c = (a + b)/2$.
   Write this algorithm recursively and iteratively.
   Test it on finding the square root of 2: choose $f(x) = x^2 - 2$ with $a = 0$ and $b = 2$ initially.