Data Structures and Algorithms

In C++

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
February 16, 2023
United States

Contents

Selection Sort

Concept 1: The selection sort algorithm sorts an array by repeatedly finding the minimum element (if sorting in ascending order) from the unsorted part of the array and putting it at the end of the sorted part of the array. The algorithm maintains two subarrays in a given array:

- A subarray of already sorted elements.
- A subarray of elements that remain to be sorted.

At the start of the algorithm, the first subarray is empty. In each pass through the outer loop of the selection sort, the minimum element from the unsorted subarray is selected and moved to the end of the sorted subarray.

1.1 Psuedocode

```
procedure selection_sort(array : list of sortable items, n :
↪  length of list)
    i := 0
    while i < n - 1
        min_index ← i
        j := i + 1
        while j < n
            if array[j] < array[min_index]
                min_index ← j
            end if
            j = j + 1
        end while
        swap array[i] and array[min_index]
        i = i + 1
    end while
end procedure
```

1.2 Example

```
int main(int argc, const char* argv[]) {
    int arr[] = {2,4,1,3,5}; int n = 5;

    for (int j=0; j <n-1; ++j) {
        int min = j;
        for (int k=j+1; k <n-1; ++k) {
            if (arr[k] < arr[min]) {
                min = k;
            }
        }
        std::swap(arr[j], arr[min]);
    }
}
```

1.3 Complexity

- **Time Complexity:** $O(n^2)$
- **Space Complexity:** $O(1)$

Note:-

The primary advantage of selection sort is that it never makes more than $O(n)$ swaps, which can be useful if the array elements are large and copying them is a costly operation.

Insertion Sort

Concept 2: The insertion sort algorithm sorts a list by repeatedly inserting an unsorted element into the correct position in a sorted sublist. The algorithm maintains two sublists in a given array:

- A sorted sublist. This sublist initially contains a single element (an array of one element is always sorted).
- A sublist of elements to be inserted one at a time into the sorted sublist.

2.1 Psuedocode

```
procedure insertion_sort(array : list of sortable items, n :
↪ length of list)
    i ← 1
    while i < n
        j ← i
        while j > 0 and array[j - 1] > array[j]
            swap array[j - 1] and array[j]
            j ← j - 1
        end while
        i ← i + 1
    end while
end procedure
```

2.2 Example

```
int main(int argc, const char* argv[]) {
    int arr[] = {2,4,1,3,5};
    int n = 5;

    for (int j=1; j<n; ++j) {
        for (int k=j; k>0; --k) {
            if (arr[k-1] > arr[k]) {
                std::swap(arr[k-1], arr[k]);
            }
        }
    }
}
```

2.3 Optimizing Insertion Sort

Performing a full swap of the array elements in each inner for loop iteration is not necessary. Instead, we save the value that we want to insert into the sorted subarray in temporary storage. In place of performing a full swap, we simply copy elements to the right. The saved value can then be inserted into its proper position once that has been located.

This alternative approach can potentially save a considerable number of assignment statements. If N swaps are performed by the inner loop, the original version of insertion sort requires $N \cdot 3$ assignment statements to perform those swaps. The improved version listed below only requires $N + 2$ assignment statements to accomplish the same task.

2.3.1 Psuedocode

```
procedure insertion_sort(array : list of sortable items, n :
↪  length of list)
    i ← 1
    while i < n
        temp ← array[i]
        j ← i
        while j > 0 and array[j - 1] > temp
            array[j] ← array[j - 1]
            j ← j - 1
        end while
        array[j] ← temp
        i ← i + 1
    end while
end procedure
```

2.3.2 Example

```
int arr[] = {5,6,4,3,1};
int n = 5;

for (int j=1; j<n; ++j) {
    int tmp = arr[j];
    int k=j;
    for (; k>0; --k) {
        if (arr[k-1] > tmp) {
            arr[k] = arr[k-1];
        } else {
            break;
        }
    }
    arr[k] = tmp;
}
```

2.4 Complexity

- **Time Complexity:** $O(n^2)$
- **Space Complexity:** $O(1)$

Note:-

The primary advantage of insertion sort over selection sort is that selection sort must always scan all remaining unsorted elements to find the minimum element in the unsorted portion of the list, while insertion sort requires only a single comparison when the element to be inserted is greater than the last element of the sorted sublist. When this is frequently true (such as if the input list is already sorted or partially sorted), insertion sort is considerably more efficient than selection sort. The best case input is a list that is already correctly sorted. In this case, insertion sort has O(n) complexity.

Bubble Sort

Concept 3: Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

3.1 Psuedocode

```
procedure bubble_sort(array : list of sortable items, n : length
↪ of list)
    do
        swapped ← false
        i ← 1
        while i < n
            if array[i - 1] > array[i]
                swap array[i - 1] and array[i]
                swapped ← true
            end if
            i ← i + 1
        end while
    while swapped
end procedure
```

Note:-

If no items are swapped during a pass through the outer loop (i.e., the variable swapped remains false), then the array is already sorted and the algorithm can terminate.

3.2 Example

```
int arr[] = {5,6,4,3,1};
int n = 5;

bool swapped;
do {
    swapped = 0;

    for (int i=0; i<n; ++i) {
        if (arr[i-1] > arr[i]) {
            std::swap(arr[i-1], arr[i]);
            swapped = 1;
        }
    }

} while (swapped);

```

3.3 Optimizing Bubble Sort

The bubble sort algorithm can be optimized by observing that the n-th pass finds the n-th largest element and puts it into its final place. Therefore the inner loop can avoid looking at the last $n - 1$ items when running for the n-th time:

3.3.1 Psuedocode

```
procedure bubble_sort(array : list of sortable items, n : length
↪  of list)
    do
        swapped ← false
        i ← 1
        while i < n
            if array[i - 1] > array[i]
                swap array[i - 1] and array[i]
                swapped ← true
            end if
            i ← i + 1
        end while
        n ← n - 1
    while swapped
end procedure
```

It is common for multiple elements to be placed in their final positions on a single pass. In particular, after every pass through the outer loop, all elements after the position of the last swap are sorted and do not need to be checked again. Taking this into account makes it possible to skip over many elements, resulting in about a worst case 50% improvement in comparison count (though no improvement in swap counts), and adds very little complexity because the new code subsumes the swapped variable:

```
 do
    last ← 0
    i ← 1
    while i < n
        if array[i - 1] > array[i]
            swap array[i - 1] and array[i]
            last ← i
        end if
        i ← i + 1
    end while
    n ← last
while n > 1
```

3.3.2 Example

```
int last;
do  {
    last = 0;
    int j=1;

    for (; j<n; ++j) {
        if (arr[j-1] > arr[j]) {
            std::swap(arr[j-1], arr[j]);
            last = j;
        }
    }
    n = last;

} while (n > 0);
```

3.4 Complexity

- **Time Complexity:** $O(n^2)$
- **Space Complexity:** $O(1)$

Note:-

Other $O(n^2)$ sorting algorithms, such as insertion sort, generally run faster than bubble sort (even with optimizations) and are no more complex. Therefore, bubble sort is not a practical sorting algorithm. The only significant advantage that bubble sort has over most other sorting algorithms (but not insertion sort), is that the ability to detect that the list is sorted is built into the algorithm. When the list is already sorted (best-case), the complexity of bubble sort is only $O(n)$.

Two-Dimensional Array

Recursion

Complexity Analysis

6.1 Time Complexity

Concept 4: Time complexity in algorithms is a way to describe the efficiency of an algorithm in terms of the time it takes to run as a function of the length of the input. It gives us an idea of the growth rate of the runtime of an algorithm as the size of input data increases. Big O notation is a mathematical notation used to express this time complexity, focusing on the worst-case scenario or the upper limit of the algorithm's running time.

Big O notation describes the upper bound of the time complexity, ignoring constants and lower order terms which are less significant for large input sizes. Here are some common Big O notations and their meanings:

6.1.1 Common time complexities

The following is a list of the common time complexities, in order from best to worst

Notation	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^k)$	Polynomial
$O(2^n)$	Exponential
$O(n!)$	Factorial

You can also have multiple variables in your runtime. For example, the time required to paint a fence that is w meters wide and h meters high could be described as $O(wh)$. If you needed p layers of paint, then you could say that the time is $O(whp)$

6.1.2 Constant time

An O(1) time complexity, also known as constant time complexity, describes an algorithm where the time to complete does not depend on the size of the input data set.

6.1.3 Big O, Big Omega, and Big Theta

Academics use big O, big Θ, and big Ω to describe runtimes.

- **Big O** notation (denoted as O) is widely used in academia to describe an upper bound on the time complexity of an algorithm. For instance, an algorithm that prints all the values in an array could be described as $O(N)$. However, it could also be described as $O(N^2)$, $O(N^3)$, or $O(2^N)$, among other possible Big O notations. The algorithm's execution time is at least as fast as each of these, making them upper bounds on the runtime. This relationship is akin to a less-than-or-equal-to relationship. For example, if Bob is X years old (assuming no one lives past age 130), then it would be correct to say that $X \leqslant 130$. Similarly, it would also be correct, albeit less useful, to say that $X \leqslant 1,000$ or $X \leqslant 1,000,000$. While these statements are technically true, they are not particularly informative. Likewise, a simple algorithm to print the values in an array is $O(N)$, but it is also correct to describe it as $O(N^3)$ or any runtime larger than $O(N)$. This illustrates that while multiple Big O notations can technically describe the time complexity of an algorithm, the most informative description is the one that provides the tightest upper bound.
- **Big Ω**: In academia, Ω is the equivalent concept but for the lower bound. Printing the values in an array is $\Omega(N)$ as well as $\Omega(\log N)$ and $\Omega(1)$. After all, you know it won't be faster than those runtimes. The Ω notation is used to describe the best-case scenario or the minimum amount of time an algorithm will take to complete. It ensures that the algorithm's execution time will not be less than the specified complexity, providing a guarantee on the lower limit of the algorithm's performance.
- **Big Θ**: In academia, Θ notation signifies that an algorithm's time complexity has both an upper and a lower bound. That is, an algorithm is $\Theta(N)$ if it is both $O(N)$ and $\Omega(N)$. Θ notation provides a tight bound on runtime, indicating that the algorithm's execution time grows at a rate directly proportional to the size of the input, neither faster nor slower. This precise characterization makes Θ especially useful for describing algorithms where the upper and lower bounds converge to the same complexity, offering a complete understanding of the algorithm's efficiency.

Note:-

In the industry, when people refer to big O notation, they are likely talking about big $\boldsymbol{\Theta}$

6.1.4 Best Case, Worst Case, and Expected (or Average) Case

We can actually describe our runtime for an algorithm in three different ways. Let's look at this from the perspective of quick sort.

- **Best Case:** If all elements of the array are equal, then quick sort will, on average, just traverse through the array once. This is $\mathcal{O}(N)$. (This actually depends slightly on the implementation of quick sort. There are implementations that will run very quickly on a sorted array.)
- **Worst Case:** What if we get really unlucky and the pivot is repeatedly the biggest element in the array? (Actually, this can easily happen. If the pivot is chosen to be the first element in the subarray and the array is sorted in reverse order, we'll have just this situation.) In this case, our recursion doesn't divide the array in half and recursively sort each half, it just shrinks the subarray by one element. We end up with something similar to selection sort and the runtime degenerates to $\mathcal{O}(N^2)$.

- **Expected Case:** Usually, though, these wonderful or terrible situations won't happen. Sure, sometimes the pivot will be very low or very high, but it won't happen over and over again. We can expect a runtime of $\mathcal{O}(N \log N)$.

We rarely discuss best case time complexity because it's not a very useful concept. After all, wecould take essentially any algorithm, special case some input, and then get a O(1) runtime in thebest case.For many – probably most – algorithms, the worst case and the expected case are the same.Sometimes they're different though and we need to describe both of the runtimes

6.2 Space complexity

Time is not the only thing that matters in an algorithm. We might also care about the amount of memory – or space – required by the algorithm. Space complexity is a parallel concept to time complexity. If we need to create an array of size n,this will require $O(n)$ space. If we need a two-dimensional array of size $n \times n$, this will require $O(n^2)$ space.

6.2.1 Constant time

An algorithm has O(1) space complexity when the amount of memory it requires does not grow with the size of the input data set. This means the algorithm needs a constant amount of memory space, regardless of how large the input is.

6.2.2 Space complexity in recursive algorithms

Stack space in recursive calls counts too. For example, code like this would take $O(n)$ time and $O(n)$ space

```
1  // Example 1
2  int sum(int n) {
3  if (n <= 0)
4      return 0;
5  else
6      return n + sum(n - 1);
7  }
```

Each of these calls results in a stack frame with a copy of the variable n being pushed onto the program call stack and takes up actual memory

6.3 Drop the constants

It is entirely possible for $O(n)$ code to run faster than $O(1)$ code for specific inputs. Big O just describes the rate of increase, not the specific time required

For this reason, we drop the constants in runtimes. An algorithm that one might have described as $O(2N)$ is actually $O(N)$. If you're going to try to count the number of instructions, then you'd have to go to the assembly level and take into account that multiplication requires more instructions than addition, how the compiler would optimize something, and all sorts of other details.

That would be horrendously complicated, so don't even start going down that road. Big O allows us to express how the runtime scales. We just need to accept that it doesn't mean that $O(N)$ is always better than $O(N^2)$.

6.4 Drop the non-dominant terms

What do you do about an expression such as $O(N^2 + N)$? That second N isn't exactly a constant. But it's not especially important. We already said that we drop constants. $O(N^2 + N^2)$ is $O(2N^2)$, and therefore it would be $O(N^2)$. If we don't care about the latter N^2 term, why would we care about N? We don't.

We might still have a sum in a runtime. For example, the expression $O(B^2 + A)$ cannot be reduced (without some special knowledge of A and B)

6.5 Multi-Part Algorithms: Add vs. Multiply

Suppose you have an algorithm that has two steps. When do you multiply the runtimes and when do you add them?

```
1  // Program 1
2
3  for (int i=0; i<A; ++i) {
4      cout << arrayA[i];
5  }
6
7  for (int i=0; i<B; ++i) {
8      cout << arrayB[i];
9  }
10
11 // Program 2
12 for (int i=0; i<A; ++i) {
13     for (int j=0; j<B; ++j) {
14         cout << arrayA[i] << ", " << arrayB[j];
15     }
16 }
```

In the first example, we do A chunks of work then B chunks of work. Therefore, the total amount of work is $O(A+B)$.In the second example, we do B chunks of work for each element in A. Therefore, the total amount of work is $O(A * B)$

6.6 Amortized Time

A `C++` vector object allows you to have the benefits of an array while offering flexibility in size. You won't run out of space in the vector since its capacity will grow as you insert elements. A vector is implemented with a dynamic array. When the number of stored in the array hits the array's capacity, the vector class will create a new array with double the capacity and copy all of the elements over to the new array. The old array is then deleted.

How do you describe the runtime of insertion? This is a tricky question. The array could be full. If the array contains N elements, then inserting a new element will take $O(N)$ time. You will have to create a new array of capacity $2N$ and then copy N elements over. This insertion will take $O(N)$ time. However, we also know that this doesn't happen very often. The vast majority of the time, insertion will be in $O(1)$ time.

We need a concept that takes both possibilities into account. This is what amortized time does. It allows us to describe that, yes, this worst case happens every once in a while. But once it happens, it won't happen again for so long that the cost is "amortized."

In this case, what is the amortized time? As we insert elements, we double the capacity when the size of the array is a power of 2. So after X elements, we double the capacity at array sizes 1, 2, 4, 8, 16, ..., X. That doubling takes, respectively, 1, 2, 4, 8, 16, 32, 64, ..., X copies.

What is the sum of $1 + 2 + 4 + 8 + 16 + \ldots + X$? If you read this sum left to right, it starts with 1 and doubles until it gets to X. If you read right to left, it starts with X and halves until it gets to 1. What then is the sum of $X + X/2 + X/4 + X/8 + \ldots + 1$? This is roughly $2X$. (It's $2X - 1$ to be exact, but this is big O notation, so we can drop the constant.).

Therefore, X insertions take $O(2X)$ time. The amortized time for each insertion is therefore $O(1)$.

6.7 Log N Runtimes

We commonly see O(log N) in runtimes. Where does this come from?

Let's look at binary search as an example. In binary search, we are looking for an item search_key in an N element sorted array. We first compare search_key to the midpoint of the array. If $\text{search}_key == array[mid], then we return. If search_key < array[mid], then we search on the left side of the array$

We start off with with an N-element array to search. Then, after a single step, we're down to N/2 elements. One more step, and we're down to N/4 elements. We stop when we either find the value

or we're down to just one element. The total runtime is then a matter of how many steps (dividing N by 2 each time) we can take until N becomes 1.

We could look at this in reverse (going from 1 to 16 instead of 16 to 1). How many times can we multiply N by 2 until we get N?

What is k in the expression $2^k = n$? This is exactly what log expresses.

$$\begin{aligned} 2^k &= n \\ &= \log_2 n = k. \end{aligned}$$

6.8 Recursive runtime

Here's a tricky one. What's the runtime of this code?

```
int f(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return f(n-1) + f(n-1);
    }
}

```

let's derive the runtime by walking through the code. Suppose we call f(4). This calls f(3) twice. Each of those calls to f(3) calls f(2), until we get down to f(1).

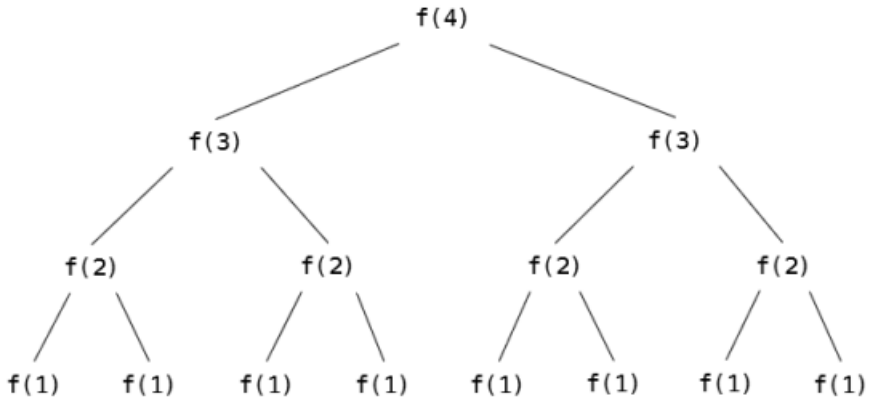


Figure 1:

How many calls are in this tree?

The tree will have depth N. Each node (i.e., function call) has two children. Therefore, each level will have twice as many calls as the one above it.

Therefore, there will be $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^N$ (which is $2^{N+1}1$) nodes.

Try to remember this pattern. When you have a recursive function that makes multiple calls, the runtime will often (but not always) look like $O(\text{branches}^{\text{depth}})$ where branches is the number of time each recursive call branches. In this case, this gives us O(2N).

The space complexity of this algorithm will be $O(N)$. Although we have O(2N) function calls in the tree total, only $O(N)$ exist on the call stack at any given time. Therefore, we would only need to have $O(N)$ memory available.

Shell sort

Concept 5: Shell sort is an advanced variant of insertion sort. It first sorts elements that are far apart from each other and successively reduces the interval (gap) between the elements to be compared. The idea is to arrange the list of elements into a sequence of incrementally more sorted arrays, which are then finally sorted with a simple insertion sort.

The key concept in Shell sort is the use of an interval to compare elements. Initially, elements far apart are compared and swapped if necessary. As the algorithm progresses, the interval decreases, making the array more and more sorted, until the interval is 1. At an interval of 1, the algorithm is essentially performing a standard insertion sort, but by this time, the array is partially sorted, making the insertion sort more efficient.

7.1 Example

```
int arr[] = {6,3,2,1,8};
int n = 5;

for (int iv=n/2; iv>0; iv/=2) {

    for (int i=iv; i<n; ++i) {

        int j;
        int tmp = arr[i];
        for (j = i; j>=iv && arr[j-iv] > arr[j]; j-=iv) {
            arr[j] = arr[j-iv];
        }
        arr[j] = tmp;
    }
}
```

Quick Sort (Recursive)

Concept 6: The quicksort algorithm is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. The steps are:

- Pick an element, called a pivot, from the array.
- Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this reordering, the pivot is in its final, sorted position. This reordering is called the partition operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

8.1 Base case

The base case of the recursion is an array of size zero or one, which is in order by definition and requires no further sorting.

8.2 Pivot selection

The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.

8.3 Psuedocode

8.3.1 What main calls

```
1 procedure quicksort(array : list of sortable items, n : length
  ↪ of list)
2     quicksort(array, 0, n - 1)
3 end procedure
```

8.3.2 Recursive function

```
procedure quicksort(array : list of sortable items, start :
↪  first element of list,
end : last element of list)
    if start < end
        pivot_point ← partition(array, start, end)
        quick_sort(array, start, pivot_point - 1)
        quick_sort(array, pivot_point + 1, end)
    end if
end procedure
```

8.3.3 Partition function

```
procedure partition(array : list of sortable items, start :
↪  first element of list,
end : last element of list)
    mid ← (start + end) / 2
    swap array[start] and array[mid]

    pivot_index ← start
    pivot_value ← array[start]

    scan ← start + 1
    while scan <= end
        if array[scan] < pivot_value
            pivot_index ← pivot_index + 1
            swap array[pivot_index] and array[scan]
        end if
        scan ← scan + 1
    end while

    swap array[start] and array[pivot_index]

    return pivot_index
end procedure
```

8.4 Examples

```
int partition(int arr[], int start, int end) {
    int pivot_index, pivot_value, mid, scan;

    mid = (start + end) / 2;
    std::swap(arr[start], arr[mid]);

    pivot_index = start;
    pivot_value = arr[start];

    scan = start + 1;

    while (scan <= end) {
        if (arr[scan] < pivot_value) {
            ++pivot_index;
            std::swap(arr[pivot_index], arr[scan]);
        }
        ++scan;
    }
    std::swap(arr[start], arr[pivot_index]);

    return pivot_index;
}

void quicksort(int arr[], int start, int end) {
    int pivot_point;
    if (start < end) {
        pivot_point = partition(arr, start, end);
        quicksort(arr, start, pivot_point - 1);
        quicksort(arr, pivot_point + 1, end);
    }
}

void quicksort(int arr[], int n) {
    quicksort(arr, 0, n-1);
}

int main(int argc, const char* argv[]) {

    int arr[]  = {3,6,1,9,12,7,36,24,18,4};
    int n = 10;

    quicksort(arr,n);

    return EXIT_SUCCESS;
}
```

8.5 Complexity

- **Time Complexity:** $O(n \log n)$
- **Space Complexity:** $O(\log n)$

Merge Sort Algorithm

Concept 7: Merge sort works as follows:

- Divide the unsorted list into n sublists, each containing one element. A list of one element is sorted by definition.
- Repeatedly merge sorted sublists (called "runs") to produce longer runs until there is only one run remaining. This is the sorted list

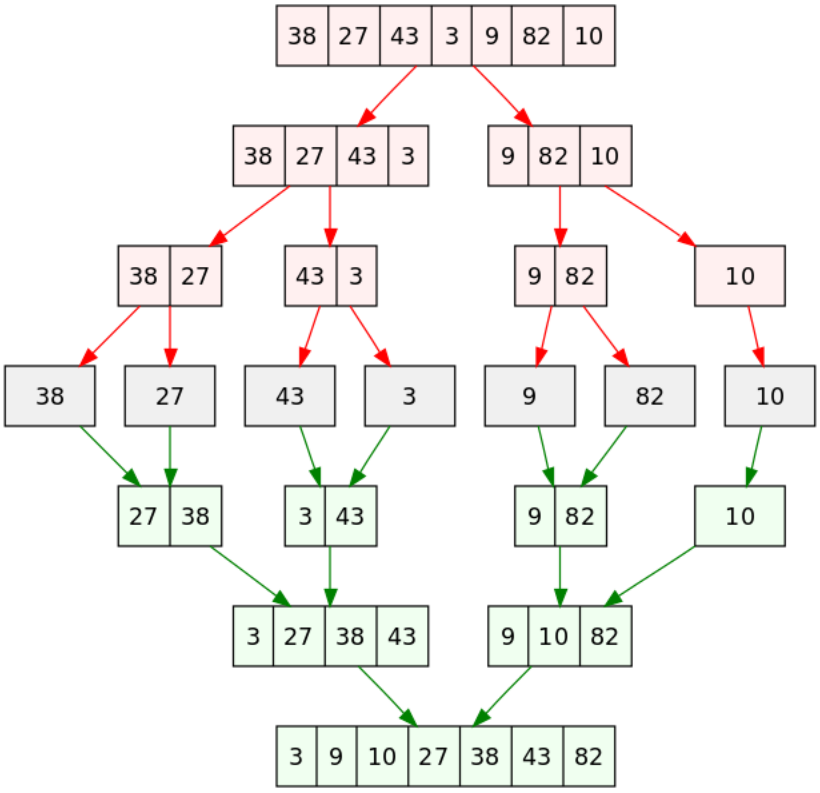


Figure 2:

This algorithm makes use of a variable length temporary array, which is most easily represented in C++ using the **vector** class from the standard library. When implementing the algorithm, include the following code at the top of the source file:

9.1 Psuedocode

```
procedure merge_sort(array : list of sortable items, start :
↪  first element of list,
end : last element of list)
    if start < end
        mid ← (start + end) / 2

        merge_sort(array, start, mid)
        merge_sort(array, mid + 1, end)

        merge(array, start, mid, end)
    end if
end procedure

procedure merge(array : list of items to merge, start : first
↪  element of first sublist, mid : last element of first
↪  sublist,
  end : last element of second sublist)
    vector<int> temp(end - start + 1);

    i ← start
    j ← mid + 1
    k ← 0

    while i <= mid and j <= end
        if array[i] < array[j]
            temp[k] ← array[i]
            i ← i + 1
        else
            temp[k] ← array[j]
            j ← j + 1
        end if
        k ← k + 1
    end while

    while i <= mid
        temp[k] ← array[i]
        i ← i + 1
        k ← k + 1
    end while

    while j <= end
        temp[k] ← array[j]
        j ← j + 1
        k ← k + 1
    end while

    Copy the elements of the vector temp back into array
end procedure
```

9.2 Example

```
void merge(int arr[], int start, int mid, int end) {

    vector<int> temp(end - start + 1);

    int i,j,k;

    i = start;
    j = mid + 1;
    k = 0;

    while (i <= mid && j<= end) {
        if (arr[i] < arr[j]) {
            temp[k] = arr[i];
            ++i;
        } else {
            temp[k] = arr[j];
            ++j;
        }
        ++k;
    }

    while (i <= mid) {
        temp[k] = arr[i];
        ++i;
        ++k;
    }

    while (j <= end) {
        temp[k] = arr[j];
        ++j;
        ++k;
    }

    for ( i=start, j=0; i<=end; ++i, ++j) {
        arr[i] = temp[j];
    }
}

void merge_sort(int arr[], int start, int end) {

    int mid;
    if (start < end) {
        mid = (start + end) / 2;

        merge_sort(arr, start, mid);
        merge_sort(arr, mid+1, end);

        merge(arr, start, mid, end);
    }
}
```

Binary Heap

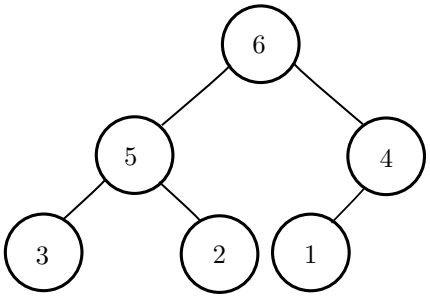
Concept 8: A **binary heap** is a data structure that takes the form of binary tree with two additional constraints

- The binary tree must be complete or almost complete; that is, all levels of the tree, except possibly the last (deepest) one, are fully filled. If the last level of the tree is not complete, the nodes of that level are filled from left to right.
- The key stored in each node is either greater than or equal to or less than or equal to the keys in the node's children.

A binary heap where the parent key is greater than or equal to the child keys is called a **max-heap**; a heap where the parent key is less than or equal to the child keys is called a **min-heap.**

Because a binary heap is always a complete or almost complete binary tree, the tree nodes can be efficiently stored in an array with no wasted space. The top-level node (or root) of the tree is stored in the first element of the array. Then, for each node in the tree that is stored at subscript k. the node's left child can be stored at subscript $2k + 1$ and the right child can be stored at subscript $2k + 2$.

10.1 Example



10.2 Insertion

When placing nodes, we first place the root. From there, we add from left to right.

When adding new nodes to the head, we place them at the bottom. However, this may lead to a violation in the trees order. In this case, we compare the newly inserted node to its parent, swapping them if necessary, we do this until the node is in the correct position.

10.3 Binary heap imbalance

It is not always possible to have a balanced heap. That is, for each level of the tree, each side has the same number of nodes. To account for this, we allow the left sub tree to hold one more than the right sub tree.

10.4 Deletion

Because the binary heap is designed to give us access to the minimum element (min-head), or maximum element (max-head), we can only delete the root node.

Once we delete the root node, you may notice that the shape is disrupted. That is, we no longer have a root node. To solve this, we first get the tree back to its correct shape, and then focus on the invariance.

Note: Invariance in a binary heap referes to the property that ensures the status of the min or max heap.

Heap Sort

Concept 9: **Heapsort** is a comparison-based sorting algorithm that uses an implicit binary heap. Heapsort can be thought of as an improved selection sort: like selection sort, heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. Unlike selection sort, heapsort does not waste time with a linear-time scan of the unsorted region; rather, heap sort maintains the unsorted region in a heap data structure to find the largest element more quickly in each step.

The heapsort algorithm can be divided into two parts:

- In the first part, the elements of an unsorted array are rearranged to create a binary heap (a max-heap if the array is to be sorted in ascending order).
- In the second part, a sorted array is created by repeatedly swapping the largest element from the heap (the root of the heap) with the last element of the heap and then decrementing the heap size (which effectively removes that element from the heap). The heap is updated after each removal to recreate the max-heap property. Once all elements have been removed from the heap, the result is a sorted array.

11.1 Psuedocode

```
procedure heap_sort(array : list of sortable items, n : length of list)
    // end : array subscript
    // Build the heap in array so that largest value is at the
    // root.
    heapify(array, n);

    end = n - 1

    while end > 0
        // array[0] is the root and largest value. The swap
        // moves it in front of the sorted elements.
        swap array[end] and array[0]

        // The heap size is reduced by 1.
        end = end - 1

        // The swap ruined the heap property, so restore it.
        sift_down(array, 0, end);
    end while
end procedure

procedure heapify(array : list of sortable items, n : length of list)
    // start : array subscript
    start = (n - 2) / 2 // Find parent of last element of array
    while start >= 0
        // Sift down the value at subscript 'start' to the proper place
        // such that all values below the start subscript are in max
        // heap order

        sift_down(array, start, n - 1)

        // Go to next parent
        start = start - 1
    end while

    // All elements are now in max heap order
end procedure
```

```
procedure sift_down(array : list of sortable items, start :
↪  starting
  subscript of heap, end : ending subscript of heap)
    // root : array subscript
    // largest : array subscript
    // child : array subscript

    // Repair the heap whose root element is at subscript
↪  'start',
    // assuming the heaps rooted at its children are valid

    root = start

    // While the root has at least one child
    while (2 * root + 1) <= end
        child = 2 * root + 1 // Left child of root
        largest = root // Assume root is largest

        // If left child is larger than root, left child is
↪  largest
        if array[largest] < array[child]
            largest = child
        end if

        // If there is a right child and it is greater than
↪  largest,
        // right child is largest
        if (child + 1) <= end and array[largest] < array[child+1]
            largest = child + 1
        end if

        // If root is largest, no need to continue
        if largest == root
            return
        else
            swap array[root] and array[largest]
            root = largest
        end if

    end while
end procedure
```