Jordan Tam — CS 385-E

C++ References and Pointers

A **reference** acts as an alias for an existing variable and stores the memory address of the variable it refers to. References cannot be reassigned.

A **pointer** is an object that holds a memory address as its value.

`int x = 5`	`x` is a variable of type `int`
`cout << x`	Prints the value x (5).
`int& ref = x`	`ref` is a variable of type `int&`.
`ref = 10`	`x = 10`
`cout << ref`	Prints the value of `x`.
`cout << &ref`	Prints the memory address of `x`.
`int* ptr = &x`	`ptr` is a variable of type `int*`
`cout << ptr`	Prints the memory address of `x`.
`*ptr = 20`	`x = 20`
`cout << *ptr`	Prints the value of `x`.
`cout << &ptr`	Prints memory address of `ptr`.

Bit Operators

<<	left shift
>>	right shift
&	AND
|	OR
^	XOR

Bit Manipulation

Logical shift left: Move every bit n times to the left, patch 0s at end. Equivalent to multiplication by 2^n.

Logical shift right: Move every bit right, discard LSB, pad 0s at beginning.

Arithmetic shift right: Like LSR, but pad MSBs.

Arrays

`int array[5] {0, 1, 2, 3, 4};`

`array[0] = 4;`

`method_for_array(array, length)` // Since fixed arrays decay to a pointer, functions don't have access to array length, so we need to manually pass that number to the function.

Create a dynamic array:

```
int length = 5;
int* dynamicArray = new int[length];
delete[] dynamicArray;
```

Create a dynamic 2D array:

```
int** array = new int*[numRows];
for(int i = 0; i < numRows; i++) {
    array[i] = new int[numCols];
```

std::vector

`#include <vector>`

Initialization: `std::vector<int> vec = {1, 2, 3, 4, 5};`

Access: `vec[0]; vec.at(0); vec.front(); vec.back();`

Add:

- `push_back(n)` - Append n to the end.
- `insert(vec.begin(), n)` - Insert n at the beginning.
- `insert(vec.end(), n)` - Insert n at end.
- `insert(vec.begin() + 1, n)` - Insert n at index 1.

Delete: `pop_back()` - Delete last element.

Size: `size()` - Returns # of elements, return type `size_t`.

Iteration:

```
void display_vector(const vector<int> &nums) {
    for(auto it = nums.cbegin(); it != nums.cend(); ++it) {
        cout << *it << endl;
    }
}
```

std::stringstream

Terminal: `./name_of_program 65 4 hello`

```
argc = 4
argv = [“./name_of_program”, “65”, “4”, “hello”]
    argv[0] = the name of the C++ program
    argv[1] to argv[argc - 1] = all other inputs

int main(int argc, char* argv[]) {
    int num;
    std::istringstream iss;
    iss.str(argv[1]); // Try to convert “65” to an integer
    // If successful, num will hold the integer value 65.
    // iss >> num can fail if the argument is not an integer.
    if(!(iss >> num)) {
        std::cerr << “Input not an integer\n”;
        return 1;
    }
    cout << num;
    return 0;
}
```

Summations

$\sum_{i=1}^{N}(C \times i) = C \times \sum_{i=1}^{N} i$

$\sum_{i=C}^{N} i = \sum_{i=0}^{N-C}(i + C)$

$\sum_{i=C}^{N} i = \sum_{i=0}^{N} i - \sum_{i=0}^{C-1} i$

$\sum_{i=1}^{N}(A + B) = \sum_{i=1}^{N} A + \sum_{i=1}^{N} B$

$\sum_{i=0}^{N}(N - i) = \sum_{i=0}^{N} i$

$\sum_{i=1}^{N} 1 = N$

$\sum_{i=1}^{N} C = C \times N$

$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$

$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6}$

$\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$

$\sum_{i=1}^{N} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{N} \approx \log n = \ln N + \gamma$

Asymptotic Notation

O-notation: $t(n) \le c \cdot g(n)$ for all $n \ge n_0$.

Ω-notation: $t(n) \le c \cdot g(n)$ for all $n \ge n_0$.

θ-notation: $c_2 g(n) \le t(n) \le c_1 g(n)$ for all $n \ge n_0$.

$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2)$

$O(n^2 \log n) < O(n^3) < O(2^n) < O(n!) < O(n^n)$

Log base: $\log_k(n) > log_{k+1}(n)$

N^{th} root: $\sqrt[k]{n} > \sqrt[k+1]{n}$

Using Limits for Comparing Orders of Growth:

$\lim_{n\to\infty} \frac{t(n)}{g(n)} = 0 : t(n) \in O(g(n))$

$\lim_{n\to\infty} \frac{t(n)}{g(n)} = c : t(n) \in O/\Omega/\theta(g(n))$

$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \infty : t(n) \in \theta(g(n))$

L'Hôpital's Rule: $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$

Stirling's formula: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ for large n.

Runtime Analysis

n	`for(i = 1; i ≤ n; i++)`
$\sqrt{n}$	`for(i = 1; i*i ≤ n; i++)`
	`for(i = 1, s = 1; s ≤ n; i++, s+= i)`
$\sqrt[3]{n}$	`for(i = 1; i*i*i ≤ n; i++)`
$\lg(n)$	`for(i = 1; i ≤ n; i *= 2)`
n^2	`for(i = 1; i ≤ n; i++)`
	`for(j = 1; j ≤ n; j+=2)`
$n \ln(n)$	`for(i = 1; i ≤ n; i++)`
	`for(j = 1; j ≤ n; j += i)`

Recurrence Relations

1. Solve for $x(n-1)$ or $x\left(\frac{n}{a}\right)$ and plug it into $x(n)$.
2. Solve for $x(n-2)$ or $x\left(\frac{n}{a^2}\right)$ and plug it into $x(n)$.
3. Create a general equation $x(n)$ based on the pattern found.
4. Set $n - i$ = the base case of $x(n)$ and solve for i.
5. Substitute i out of the original equation and simplify $x(n)$.

Elementary Sorting Algorithm

Bubble Sort: Compares adjacent array elements.

- $\theta(\frac{n(n-1)}{2})$ comparisons.
- Best case for swaps is 0, when array is sorted.
- Worst case is $\theta(\frac{n(n-1)}{2})$, when array is in decreasing order.
- For each i^{th} iteration, traverse array from index 0 to $length - i$. At the end, the i^{th} largest element is in its sorted position.

Bubble Sort Optimized: Only traverses from index 0 to where the last swap occurred in the previous iteration.

Selection Sort: Find smallest element and swap it with the unsorted element with the smallest index.

- For each i^{th} iteration, finds the i^{th} smallest element and moves it to index i.
- $\theta(\frac{n(n-1)}{2})$ comparisons.
- Best case for swaps is 0, when array is sorted.
- Worst case is $length - 1$, where a swap is required on every iteration of the outer loop.

Insertion Sort: Takes the first element in unsorted part of array and traverses through the sorted part backwards to insert it at the correct position.

- Best case: Array is sorted. 0 shifts, 1 comparison per pass.
- Worst case: Array in decreasing order. 1 shift for every comparison and i comparisons per pass for a total of $\frac{n(n-1)}{2}$ comparisons and shifts.
- Average case: Shift half of the sorted array each iteration. $\frac{i}{2}$ comparisons/shifts for a total of $\frac{n(n-1)}{4}$ comparisons/shifts.

```
void bubble_sort(int array[], const int length) {
    for(int i = 0; i < length - 1; i++) {
        for(int j = 0; j < length - 1 - i; j++) {
            if(array[j + 1] < array[j]) {
                swap(array, j, j + 1);
            }
        }
    }
} ---------------------------------------------------
void bubble_sort_opt(int array[], const int length) {
    int unsorted = length;
    while(unsorted > 1) {
        int s = 0;
        for(int j = 1; j < unsorted; j++) {
            if(array[j] < array[j - 1]) {
                swap(array, j - 1, j);
                s = j;
            }
        }
        unsorted = s;
    }
} ---------------------------------------------------
void selection_sort(int array[], const int length) {
    for(int i = 0; i < length - 1; i++) {
        int min_j = i;
        for(int j = i + 1; j < length; j++) {
            if(array[j] < array[min_j]) {
                min_j = j;
            }
        }
        if(min_j != i) {
            swap(array, i, min_j);
        }
    }
} ---------------------------------------------------
void insertion_sort(int array[], const int length) {
    for(int i = 1; i < length; i++) {
        int j, current = array[i];
        for(j = i - 1; j >= 0 && array[j] > current; j--) {
            array[j + 1] = array[j];
        }
        array[j + 1] = current;
    }
}
```

	Bubble	BSO	Selection	Insertion
best	$\theta(n^2)$	$\theta(n)$	$\theta(n^2)$	$\theta(n^2)$
worst	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$
average	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$
overall	$\theta(n^2)$	$O(n^2)$	$\theta(n^2)$	$O(n^2)$

Graphs

	Matrix	**List**
Edge between 2 v?	$\theta(1)$	$O(d), O(V)$
Find all adj v_i	$\theta(V)$	$O(d), O(V)$
Space	$\theta(V^2)$	$\theta(V + E)$
Usage	Small/dense	Large/sparse

BFS: Visit all neighbors of vertex first.

- Counter (order of visits), array (visited?), queue.

DFS: Go as far away from start as possible.

- Counter, array, stack.

Topological Sort (Khan's Algorithm): Sort vertices such that when $e = v_1 \to v_2$, v_1 is visited first.

- Array (current indegree), queue (indegree = 0), sorted list.

Runtime: List = $\theta(V + E)$, Matrix = $\theta(V^2)$

Master Theorem

$T(n) = aT(\frac{n}{b}) + f(n)$

If $f(n) \in \theta(n^d)$, then $T(n) \in$:

$\theta(n^d)$	if $a < b^d$
$\theta(n^d \log_b n)$	if $a = b^d$
$\theta(n^{\log_b a})$	if $a > b^d$

Lomuto Partitioning

ALGORITHM *LomutoPartition*$(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l \le r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l + 1$ **to** r **do**
  if $A[i] < p$
    $s \leftarrow s + 1$; swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

Runtime: The for loop traverses through indices $left + 1$ to $right$, therefore the running time is $\theta(right - left)$, so $\theta(n)$ if you are partitioning the whole array.

Quick Select

```
// Solves the selection problem by recursive partition-based algorithm.
// Input: Subarray A[l..r] of array A[0..n - 1] of orderable elements and
// integer k (0 ≤ k ≤ n - 1)
// Output: The value of the kth smallest element.
Quickselect(A, l, r, k):
    s ← LomutoPartition(A, l, r)
    if s = k - 1 return A[s]
    else if s > k - 1 Quickselect(A, l, s - 1, k)
    else Quickselect(A, s + 1, r, k)
```

Best case: $\theta(n)$ **Worst case:** $\theta(n^2)$ **Average case:** $\theta(n)$

Quicksort (Lomuto Partitioning)

```
Quicksort(A, l, r):
    if(l < r)
        s ← LomutoPartition(A, l, r)
        Quicksort(A, l, s - 1)
        Quicksort(A, s + 1, r)
```

Best case: $\theta(n \lg n)$
Worst case: $\theta(n^2)$
Average case: $\theta(n \lg n)$

MergeSort

```
MERGESORT(A, B, lo, hi)
    IF lo < hi
        mid = lo + (hi-lo)/2
        MERGESORT(A, B, lo, mid)
        MERGESORT(A, B, mid+1, hi)
        MERGE(A, B, lo, mid, hi)

MERGE(A, B, lo, mid, hi)
    i1 = lo, i2 = mid+1, i=lo
    WHILE i1<=mid AND i2<=hi
        IF A[i1]<=A[i2]
            B[i++] = A[i1++]
        ELSE
            B[i++] = A[i2++]
    FOR i1 TO mid
        B[i++] = A[i1++]
    FOR i2 TO hi
        B[i++] = A[i2++]
    Copy B[lo..hi] back into A[lo..hi]
```

Running time: $\theta(n \lg n)$ for best, average, worst, and overall.

Space Complexity: $\theta(n)$

In practice, Merge Sort is slower than Quicksort, even though Quicksort has a worse worst case running time than Merge Sort.

Counting Sort: Counts the # of times each value appears and uses that info to fill in the array with the right # of copies of each value. This works well if the array elements are within a small range $[0, max]$ of possible values.

1. $\theta(n)$: Traverse data array to find max.
2. $\theta(max)$: Allocate an array of counters of size $max + 1$.
3. $\theta(n)$: Count # of times each value appears in data array.
4. $\theta(max)$: Traverse array of counters + $\theta(n)$: Writing to the data array and decrementing the counters each time.

Time Complexity: $\theta(max + n)$

Space Complexity: $\theta(max)$

In general, Counting Sort is used when $max \le n$.

Radix Sort: For each i^{th} iteration, sort integers based on their digit at the 10^{i-1}s place; stable algorithm.

Runtime: $\theta(n)$

Log rules:

$\log_a(b) = c \to a^c = b$	$\log_a(xy) = \log_a(x) + \log_a(y)$
$\log_a(1) = 0$	$\log_a(\frac{x}{y}) = \log_a(x) - \log_a(y)$
$\log_a(a) = 1$	$\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$
$\log_a(x^y) = y \log_a(x)$	$a^{\log_a(x)} = x$
$a^{\log_b(x)} = x^{\log_b(a)}$	$(a^b)^{\log_a(x)} = x^b$

Russian Peasant Multiplication: Method for multiplying two non-negative numbers n and m.

1. If $n > m$, swap their values.
2. Divide n by 2 (right shift) and multiply m by 2 (left shift) and record values. Repeat until $n = 1$.
3. Whenever n is even, strike out corresponding m.
4. Add up remaining values of m.

Binary Reflected Gray Codes (BRGC)

```
def BRGC(n):
    if n == 1:                            # constant
        return ["0", "1"]
    L1 = BRGC(n - 1)                      # T(n - 1)
    L2 = L1[::-1]                         # 2^(n - 1)
    L3 = ["0" + code for code in L1]      # 2^(n - 1) * n
    L4 = ["1" + code for code in L2]      # 2^(n - 1) * n
    return L3 + L4                        # 2^(n - 1)
```

Running Time: Given an integer n, $BRGC(n)$ returns a list of length 2^n so the running time must be at least $\Omega(2^n)$.
$T(n) = T(n-1) + \theta(2(2^{n-1}) + 2(2^{n-1} \cdot n)) = \theta(n2^n)$, so generating gray codes takes slightly more than exponential time.

Generating Lexicographic Permutations

1. Sort letters in alphabetical order and add this permutation.
2. Find the largest i such that $a[i] < a[i+1]$.
3. Find the largest j such that $a[i] < a[j]$.
4. Reverse the order of the elements from $a[i+1]$ to $a[n]$.
5. Add the new permutation to the list and repeat from step 2 until the new permutation is in reverse alphabetical order.

Runtime: Since there are $n!$ possible permutations, the while loop iterates exactly $\theta(n!)$ times. During each iteration, the while loop processes $\theta(\frac{n}{2}) = \theta(n)$ characters. The algorithm's total running time is $\theta(n \cdot n!)$.
If the string contains duplicate letters, the # of possible permutations is $n!$/(# of times each letter is duplicated)!

- "balloon" $\rightarrow 7!/2!/2! = \frac{7!}{2!2!} = 1260$ permutations

Red-Black Tree

- For each node, the height of left and right subtree are within a factor of two. An RBT with n internal nodes has height at most $2\lg(n+1)$.
- Search, insert, and delete are $\theta(\lg n)$.
- RBT insertion/deletion rearrange shape.

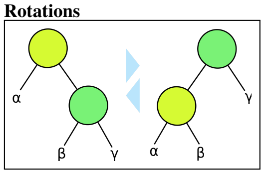
Properties

- Every node has a pointer to left/right child, parent, and a color.

1. Every node is either red or black.
2. Root is black.
3. Null pointers are black nodes.
4. If a node is red, both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same # of black nodes.

- A black node can have a black child.

Rotations



Time Complexity: $\theta(1)$

Insertion

Properties 2 and 4 may be violated during insertion.

Insertion Fixup

1. z's uncle y is RED
   - $p[z]$.color = BLACK
   - y.color = BLACK
   - $p[p[z]]$.color = RED
   - $z = p[p[z]]$

2a. $p[z]$ is a left child and z's uncle y is BLACK and z is a right child.
   - $z = p[z]$
   - left-rotate(z)

3a. $p[z]$ is a left child and z's uncle y is BLACK and z is a left child.
   - $p[z]$.color = BLACK
   - $p[p[z]]$.color = RED
   - right-rotate($p[p[z]]$)

2b. $p[z]$ is a right child and z's uncle y is BLACK and z is a left child.
   - $z = p[z]$
   - right-rotate(z)

3b. $p[z]$ is a right child and z's uncle y is BLACK and z is right child.
   - $p[z]$.color = BLACK
   - $p[p[z]]$.color = RED
   - left-rotate($p[p[z]]$)

Make sure the root's color is BLACK.

Horner's Method

```
Horner(p[0..n], x:)
  p = P[n]
  for i = n-1 down to 0:
    p = P[i] + x * p
  return p
```

Left Right Binary Exponentiation

$LeftRightBinaryExponentiation(a, b(n) = b_I ... b_0)$
  $product \leftarrow a$
  for $i \leftarrow I - 1$ down to 0 do
    $product \leftarrow product \cdot product$
    if $b_i = 1$: $product \leftarrow product \cdot a$
  return $product$

Time Complexity
- Brute force: $\theta(n)$
- LRBE: $\theta(\log_2 n)$

2-3 Tree: Search tree that consists of 2 kinds of nodes:

- **2-node:** Node with one key K and two children.
- **3-node:** Node with two keys $K_1 < K_2$ and three children.

Property: All leaves must be on the same level (length of path from root to any leaf is always same). 2-3 Tree is always perfectly height-balanced.
Insertion: New key K inserted as leaf unless tree is empty. Appropriate leaf found by performing search for K.

- If leaf is 2-node, replace it with 3-node and insert K there as either 1st or 2nd key, depending on whether K is smaller/larger than the node's old key.
- If leaf is 3-node, split leaf in two: smallest of 3 keys is put in first leaf, largest put in second, and middle key is promoted to the old leaf's parent.

Coin Row Problem

index i	0	1	2	3	4	5	...
c_i		c_1	c_2	c_3	c_4	c_5	...
$F(i)$	0	c_1					
$S(i)$	0	0					

For $i \geq 2$: $F(i) = \max(c_i + F(i-2), F(i-1))$
  $S(i) = i - 2$ when $F(i) = c_i + F(i-2)$
  $S(i) = i - 1$ when $F(i) = F(i-1)$

Backtracking

Starting from $F(n)$: Compare $F(i)$ and $F(i-1)$.
If ">" we picked up c_i; if "=" we did not. Go to $F(S(i))$ and repeat.
Add up all the coins.
Time/Space Complexity: $\theta(n)$

Coin Collection Problem

Recurrence Relation

Top-left cell: $F(1,1) = C(1,1)$
First row: $F(1, j) = F(1, j-1) + C(1, j)$
First col: $F(i, 1) = F(i-1, 1) + C(i, 1)$
Else: $F(i, j) = max(F(i-1, j), F(i, j-1)) + C(i, j)$
Backtracking: Starting from the bottom-right cell, go left or up 1 cell based on which has the higher value.
Time/Space Complexity: $\theta(nm)$

Maximum Path Sum

Going top-down:

- Top number: $F(1,1) = T(1,1)$
- Leftmost column: $F(i,1) - F(i-1,1) + T(i,1)$
- Rightmost diagonal: $F(i, j) = max(F(i-1, j-1), F(i-1, j)) + T(i, j)$
- All other cases: $F(i, j) = max(F(i-1, j-1), F(i-1, j)) + T(i, j)$

Time Complexity

- Recursive: $\theta(2^h)$, where h is the height of T.
- Dynamic: $\theta(n)$, where n is the # of elements in T.
- $n = \frac{(h+1)(h+2)}{2}$

Candies Problem

Cases:

1. If $P(i-1) \geq P(i) \leq P(i+1)$: $C(i) = 1$
2. If $P(i-1) < P(i) \leq P(i+1)$: $C(i) = C(i-1) + 1$
3. If $P(i-1) \geq P(i) > P(i+1)$: $C(i) = C(i+1) + 1$
4. If $P(i-1) < P(i) > P(i+1)$:
   $C(i) = \max(C(i-1), C(i+1)) + 1$

Time Complexity: $\theta(n)$

Knapsack Problem

Recurrence Relation: For $1 \leq i \leq n$:
If $w_i \leq j \leq W$:
$F(i, j) = \max(v_i + F(i-1, j-w_i), F(i-1, j))$
If $0 \leq j \leq w_i$: $F(i, j) = F(i-1, j)$

Backtracking

1. If $F(i, j) = F(i-1, j)$, then item i was not selected. Go to $F(i-1, j)$.
2. $F(i, j) > F(i-1, j)$, then item i was selected. Go to $F(i-1, j-w_i)$.
3. Stop when you reach a cell with 0.

Time/Space Complexity: $\theta(nW)$
Backtracking takes $O(n)$. Best case when knapsack contains 0-1 items; worst case when every item was selected.

Warshall's Algorithm

ALGORITHM *Warshall*($A[1..n, 1..n]$)
  //Implements Warshall's algorithm for computing the transitive closure
  //Input: The adjacency matrix A of a digraph with n vertices
  //Output: The transitive closure of the digraph
  $R^{(0)} \leftarrow A$
  for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
      for $j \leftarrow 1$ **to** n **do**
        $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
  return $R^{(n)}$

Time Complexity: $\theta(n^3) = \theta(V^3)$

Floyd's Algorithm

ALGORITHM *Floyd*($W[1..n, 1..n]$)
  //Implements Floyd's algorithm for the all-pairs shortest-paths problem
  //Input: The weight matrix W of a graph with no negative-length cycle
  //Output: The distance matrix of the shortest paths' lengths
  $D \leftarrow W$ //is not necessary if W can be overwritten
  for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
      for $j \leftarrow 1$ **to** n **do**
        $D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$
  return D

Time Complexity: $\theta(n^3) = \theta(V^3)$

Prim's Algorithm

ALGORITHM *Prim*(G)
  //Prim's algorithm for constructing a minimum spanning tree
  //Input: A weighted connected graph $G = \langle V, E \rangle$
  //Output: E_T, the set of edges composing a minimum spanning tree of G
  $V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
  $E_T \leftarrow \varnothing$
  for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
  return E_T

Time Complexity: Using adjacency matrix and unordered array for remaining vertices: $\theta(V^2)$.
Using adjacency list and priority queue implemented as min-heap (where insert, delete operations take $O(\log n)$ time): $O(E \log V)$.

Kruskal's Algorithm

ALGORITHM *Kruskal*(G)
  //Kruskal's algorithm for constructing a minimum spanning tree
  //Input: A weighted connected graph $G = \langle V, E \rangle$
  //Output: E_T, the set of edges composing a minimum spanning tree of G
  sort E in nondecreasing order of the edge weights $w(e_{i_1}) \leq \cdots \leq w(e_{i_{|E|}})$
  $E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
  $k \leftarrow 0$ //initialize the number of processed edges
  while $ecounter < |V| - 1$ **do**
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
      $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
  return E_T

Time Complexity: $O(|E| \log |E|)$ with an efficient union-find and sorting algorithm.

Dijkstra's Algorithm

ALGORITHM *Dijkstra*(G, s)
  //Dijkstra's algorithm for single-source shortest paths
  //Input: A weighted connected graph $G = \langle V, E \rangle$ with nonnegative weights
  // and its vertex s
  //Output: The length d_v of a shortest path from s to v
  // and its penultimate vertex p_v for every vertex v in V
  Initialize(Q) //initialize priority queue to empty
  for every vertex v in V
    $d_v \leftarrow \infty$; $p_v \leftarrow$ **null**
    Insert(Q, v, d_v) //initialize vertex priority in the priority queue
  $d_s \leftarrow 0$; *Decrease*(Q, s, d_s) //update priority of s with d_s
  $V_T \leftarrow \varnothing$
  for $i \leftarrow 0$ **to** $|V| - 1$ **do**
    $u^* \leftarrow DeleteMin(Q)$ //delete the minimum priority element
    $V_T \leftarrow V_T \cup \{u^*\}$
    for every vertex u in $V - V_T$ that is adjacent to u^* **do**
      if $d_{u^*} + w(u^*, u) < d_u$
        $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$
        Decrease(Q, u, d_u)

Time Complexity: Graphs represented by weight matrix and priority queue implemented as unordered array: $\theta(V^2)$
Graphs represented by adjacency list and priority queue implemented as min-heap: $O(E \log V)$.

Disjoint Subsets & Union-Find (Kruskal)

$makeset(x)$ creates a tree containing only vertex x.
Running time: $\theta(1)$, so $\theta(V)$ for all vertices together.
$union(x, y)$ attaches the root of y's tree to the root x's tree. The root of x's tree becomes the parent of y's tree.
Running time: $\theta(1)$
$find(x)$ returns the root of x's tree by following the parent pointers from x to the root.
Running time: $O(V)$
If $find(x) \neq find(y)$, then x and y belong to different, unconnected parts of the MST, so adding edge $< x, y >$ doesn't create a cycle.

Maximum Flow

Ford-Fulkerson Method/Algorithm

FORD-FULKERSON(G, s, t)
  for each edge $(u, v) \in G.E$
    $(u, v).f = 0$
  while there exists a path p from s to t in the residual network G_f
    $c_f(p) = \min\{c_f(u, v) : (u, v)$ is in $p\}$
    for each edge (u, v) in p
      if $(u, v) \in E$
        $(u, v).f = (u, v).f + c_f(p)$
      else $(v, u).f = (v, u).f - c_f(p)$

Edmonds-Karp Algorithm: One implementation of Ford-Fulkerson that uses BFS to find augmenting paths.

Time Complexity

Ford-Fulkerson: $\theta(E \cdot f_{max})$, where f_{max} is the value of the max flow.
Edmonds-Karp: $\theta(VE^2)$

Max-Flow Min-Cut Theorem: Cut the final flow network into two partitions in such a way that s and t are in separate partitions. Check the deleted edges and add up their flows. Edges going from s partition to t partition are positive while edges going the other way are negative. The result will always be equal to the max flow.

Summary of graph algorithms

Warshall: Path between vertices?
Floyd: Shortest path between all **Warshall/Floyd:** dynamic programming.
Prim/Kruskal: Greedy algorithm for computing MST.
Dijkstra: Shortest path from particular source to other vertices. Different from Floyd, which finds the shortest path between all pairs of vertices. Similar to Prim's representation of remaining vertices.