CPP CHEATSHEET

A. Divide and Conquer

1. Quick Sort with Median of Three

```
#include <iostream>

#include <vector>

using namespace std;

// Function to swap two elements in an array

void swap(int &a, int &b)

{ int temp = a;

 a = b;

 b = temp; }
```

This function takes two integer references (a and b) and swaps their values.

```
// Function to find median of three elements

int medianOfThree(vector<int> &arr, int left, int right)

{  int mid = left + (right - left) / 2;

  if (arr[left] > arr[mid])

    swap(arr[left], arr[mid]);

  if (arr[left] > arr[right])

    swap(arr[left], arr[right]);

  if (arr[mid] > arr[right])

    swap(arr[mid], arr[right]);

  return mid; }
```

This function finds the median among the elements at indices left, mid, and right within the array arr. It ensures that arr[left] is less than or equal to both arr[mid] and arr[right].

```
// Function to partition the array using median of three pivot

int partition(vector<int> &arr, int left, int right)

{  int pivotIndex = medianOfThree(arr, left, right);

  int pivot = arr[pivotIndex];

  swap(arr[pivotIndex], arr[right]);

  int i = left - 1;

  for (int j = left; j < right; ++j)

  {  if (arr[j] <= pivot) {

     ++i;

     swap(arr[i], arr[j]);

   }}

            swap(arr[i + 1], arr[right]);

  return i + 1; }
```

This function partitions the array arr around the pivot element. It first finds the pivot index by using the medianOfThree function. Then it places the pivot at the end of the array (arr[right]) and rearranges elements so that elements smaller than the pivot are placed to the left of it and elements greater than the pivot are placed to the right of it.

```
// Quick sort function

void quickSort(vector<int> &arr, int left, int right)

{  if (left < right)

 {

   int pivotIndex = partition(arr, left, right);

   quickSort(arr, left, pivotIndex - 1);

   quickSort(arr, pivotIndex + 1, right); }

}
```

This is the recursive implementation of the Quick Sort algorithm. It recursively sorts sub-arrays by choosing a pivot, partitioning the array around the pivot, and then sorting the sub-arrays on the left and right of the pivot.

```
int main()

{  vector<int> arr = {8, 2, 4, 9, 1, 7, 5};

  cout << "Given array: ";

  for (int num : arr)

    cout << num << " ";

  cout << endl;

  quickSort(arr, 0, arr.size() - 1);

  cout << "Sorted array: ";

  for (int num : arr)

    cout << num << " ";

  cout << endl;

  return 0; }
```

The main function initializes an array, prints the original array, sorts it using the Quick Sort algorithm, and then prints the sorted array.

1. **Best Case:** O(n log n) - When the median of three pivot selection strategy is effectively splitting the array into two nearly equal halves in each recursive call.
2. **Average Case:** O(n log n) - Same as the best case.
3. **Worst Case:** O(n^2) - This can happen when the selected pivot is always the smallest or largest element in the array, leading to highly unbalanced partitions. However the median of three strategy significantly reduces the probability of this worst-case scenario compared to a simple first or last element selection as the pivot.

2. Closest Pair Problem

```
#include <iostream>

#include <vector>

#include <cmath>

#include <algorithm>

#include <limits>

using namespace std;

// Structure to represent a point in 2D plane

struct Point {

  int x, y; };
```

This structure represents a point in a 2D plane with integer coordinates x and y.

```
// Function to calculate the Euclidean distance between two points
double distance(const Point &p1, const Point &p2) {
  return sqrt(pow(p1.x - p2.x, 2) + pow(p1.y - p2.y, 2)); }
```

This function calculates the Euclidean distance between two points using the distance formula.

```
// Function to compare points based on their x-coordinates
bool compareX(const Point &p1, const Point &p2) {
  return p1.x < p2.x; }
// Function to compare points based on their y-coordinates
bool compareY(const Point &p1, const Point &p2) {
  return p1.y < p2.y; }
```

These functions are used for sorting points based on their x-coordinates and y-coordinates respectively.

```
// Function to find the minimum distance between two points in a strip
double stripClosest(vector<Point> &strip, double d) {
  double min_dist = d;
  sort(strip.begin(), strip.end(), compareY);
  for (size_t i = 0; i < strip.size(); ++i) {
    for (size_t j = i + 1; j < strip.size() && (strip[j].y - strip[i].y) < min_dist; ++j) {
      double dist = distance(strip[i], strip[j]);
      if (dist < min_dist)
        min_dist = dist; }
  }
  return min_dist; }
```

This function finds the minimum distance between two points in a strip of points (where the x-coordinates are within a certain range).

```
// Function to find the minimum distance between two points using Divide and Conquer
double closestPairUtil(vector<Point> &points, size_t left, size_t right) {
  if (right - left <= 3) {
    double min_dist = numeric_limits<double>::max();
    for (size_t i = left; i < right; ++i) {
      for (size_t j = i + 1; j < right; ++j) {
        double dist = distance(points[i], points[j]);
        if (dist < min_dist)
          min_dist = dist; }
    }
    return min_dist; }
  size_t mid = left + (right - left) / 2;
  Point midPoint = points[mid];
  double dl = closestPairUtil(points, left, mid);
  double dr = closestPairUtil(points, mid + 1, right);
  double d = min(dl, dr);
  vector<Point> strip;
  for (size_t i = left; i < right; ++i) {
    if (abs(points[i].x - midPoint.x) < d)
      strip.push_back(points[i]); }
  return min(d, stripClosest(strip, d)); }
```

This function recursively finds the minimum distance between two points in a set of points using the Divide and Conquer approach.

```
// Function to find the closest pair of points
double closestPair(vector<Point> &points) {
  sort(points.begin(), points.end(), compareX);
  return closestPairUtil(points, 0, points.size()); }
```

This function sorts the points based on their x-coordinates, and then calls closestPairUtil to find the closest pair of points.

```
int main() {
  vector<Point> points = {{2, 3}, {12, 30}, {4, 1}, {7, 16}, {5, 5}};
  cout << "Given points: ";
  for (const auto &point : points)
    cout << "(" << point.x << ", " << point.y << ") ";
  cout << endl;
  double min_dist = closestPair(points);
  cout << "Closest pair: ";
  // Printing points with minimum distance
  for (size_t i = 0; i < points.size(); ++i) {
    for (size_t j = i + 1; j < points.size(); ++j) {
      if (distance(points[i], points[j]) == min_dist)
        cout << "(" << points[i].x << ", " << points[i].y << ") and ("
          << points[j].x << ", " << points[j].y << ") with distance = " << min_dist << endl; }
  }
  return 0; }
```

The main function initializes a vector of points, prints the given points, finds the closest pair of points, and then prints the closest pair along with their distance.

1. **Best Case:** O(n log n) - When the points are already sorted by x-coordinate, which minimizes the time taken in sorting.
2. **Average Case:** O(n log n) - Same as the best case.
3. **Worst Case:** O(n log n) - This is the complexity of the divide-and-conquer algorithm. However, the actual running time may vary depending on the number of points and their distribution in the plane.

3. Integer Exponentiation

```
#include <iostream>
using namespace std;
// Function to compute a^b using Divide-and-Conquer
long long power(long long a, long long b) {
  if (b == 0)
    return 1;
  long long half_power = power(a, b / 2);
```

```
  if (b % 2 == 0)
    return half_power * half_power;
  else
    return a * half_power * half_power; }
```

This function takes two long long integers a and b as input and returns a raised to the power b. It uses a Divide-and-Conquer approach to compute the exponentiation efficiently. If b is even, it recursively computes a^(b/2) and returns the square of that value. If b is odd, it computes a^(b/2) and returns a multiplied by the square of that value.

```
int main() {
  long long a = 3, b = 5;
  cout << "Compute " << a << "^" << b << endl;
  cout << "Result: " << power(a, b) << endl;
  return 0; }
```

In the main function, it initializes two long long variables a and b with values 3 and 5 respectively. Then it prints a message to indicate the computation being performed (Compute a^b), calls the power function to compute a^b, and finally prints the result.

1. **Best Case:** O(log n) - When the exponent b is even and the algorithm follows the path of dividing b by 2 in each step until it reaches 0.
2. **Average Case:** O(log n) - Same as the best case.
3. **Worst Case:** O(log n) - This is the complexity of the divide-and-conquer algorithm, which is significantly better than the naive approach of repeated multiplication, which has a time complexity of O(n).

4. Majority Element

```
#include <iostream>
#include <vector>
using namespace std;
// Function to find the majority element in a subarray
int findMajorityElement(vector<int> &nums, int left, int right) {
  if (left == right)
    return nums[left];
  // Divide the array into two halves
  int mid = left + (right - left) / 2;
  int leftMajority = findMajorityElement(nums, left, mid);
  int rightMajority = findMajorityElement(nums, mid + 1, right);
  // If both halves have the same majority element, return it
  if (leftMajority == rightMajority)
    return leftMajority;
  // Otherwise, count occurrences of both elements
  int leftCount = 0, rightCount = 0;
  for (int i = left; i <= right; ++i) {
    if (nums[i] == leftMajority)
      leftCount++;
    else if (nums[i] == rightMajority)
      rightCount++; }
  // Return the majority element
  return (leftCount > rightCount) ? leftMajority : rightMajority; }
```

This function recursively finds the majority element within a subarray defined by the indices left and right.

```
// Function to validate if the given element is a majority element
bool isMajorityElement(vector<int> &nums, int candidate) {
  int count = 0;
  for (int num : nums) {
    if (num == candidate)
      count++; }
  return count > nums.size() / 2; }
```

This function checks if the given element candidate is indeed a majority element in the array nums.

```
// Function to find the majority element
int majorityElement(vector<int> &nums) {
  int candidate = findMajorityElement(nums, 0, nums.size() - 1);
  if (isMajorityElement(nums, candidate))
    return candidate;
  return -1; // If majority element doesn't exist }
```

This function first finds a candidate majority element using the findMajorityElement function and then validates if the candidate is indeed a majority element using the isMajorityElement function.

```
int main() {
  vector<int> nums = {3, 2, 3, 1, 3, 4, 3, 8, 3};
  cout << "Given array: [";
  for (int i = 0; i < nums.size(); ++i) {
    cout << nums[i];
    if (i != nums.size() - 1)
      cout << ", "; }
  cout << "]" << endl;
  int majority = majorityElement(nums);
  if (majority != -1)
    cout << "Majority element: " << majority << " (appears " <<
(nums.size() / 2 + 1) << " times)" << endl;
  else
    cout << "No majority element found" << endl;
  return 0; }
```

The main function initializes a vector of integers nums, prints the given array, finds the majority element, and prints the result.

1. **Best Case: O(n log n)** - When the majority element is the same in all the halves.
2. **Average Case: O(n log n)** - Same as the best case.
3. **Worst Case: O(n log n)** - This is the complexity of the divide-and-conquer algorithm. However, the actual running time may vary depending on the distribution of the majority element and the splits of the array.

5. Convex Hull

```
#include <iostream>
```

```
#include <vector>

#include <algorithm>

using namespace std;

// Structure to represent a point in 2D plane

struct Point {

  int x, y; };
```

This structure represents a point in a 2D plane with integer coordinates x and y.

```
// Function to find the orientation of three points (p, q, r)

// Returns:

// 0 if p, q, r are collinear

// 1 if clockwise

// 2 if counterclockwise

int orientation(const Point &p, const Point &q, const Point &r) {

  int val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);

  if (val == 0)

    return 0;          // Collinear

  return (val > 0) ? 1 : 2; // Clockwise or Counterclockwise }
```

This function calculates the orientation of three points p, q, and r. It returns:
0 if the points are collinear.
1 if they are in clockwise order.
2 if they are in counterclockwise order.

```
// Function to find the convex hull using Divide-and-Conquer

void convexHullUtil(vector<Point> &points, int left, int right, vector<Point> &hull) {

  if (right - left + 1 <= 3) {

    for (int i = left; i <= right; ++i)

      hull.push_back(points[i]);

    return; }

  // Find the mid point

  int mid = left + (right - left) / 2;

  Point midPoint = points[mid];

  // Recursively find the convex hull for the left and right halves

  vector<Point> leftHull, rightHull;

  convexHullUtil(points, left, mid, leftHull);

  convexHullUtil(points, mid + 1, right, rightHull);

  // Merge the convex hulls of left and right halves

  int leftmost = 0, rightmost = 0;

  for (int i = 0; i < leftHull.size(); ++i) {

    if (leftHull[i].x < leftHull[leftmost].x)

      leftmost = i; }

  for (int i = 0; i < rightHull.size(); ++i) {

    if (rightHull[i].x > rightHull[rightmost].x)

      rightmost = i; }

  int upperTangentLeft = leftmost, upperTangentRight = rightmost;

  int lowerTangentLeft = leftmost, lowerTangentRight = rightmost;

  bool upperTangentFound = false, lowerTangentFound = false;

  while (!upperTangentFound || !lowerTangentFound) {

    upperTangentFound = true;

    while (orientation(rightHull[upperTangentRight], leftHull[upperTangentLeft], leftHull[(upperTangentLeft + 1) % leftHull.size()]) != 2) {

      upperTangentLeft = (upperTangentLeft + 1) % leftHull.size(); }

    lowerTangentFound = true;

    while (orientation(leftHull[lowerTangentLeft], rightHull[lowerTangentRight], rightHull[(rightHull.size() + lowerTangentRight - 1) % rightHull.size()]) != 2) {

      lowerTangentRight = (rightHull.size() + lowerTangentRight - 1) % rightHull.size(); }

  }

  // Merge the upper and lower tangents to form the convex hull

  for (int i = upperTangentLeft; i != lowerTangentLeft; i = (i + 1) % leftHull.size())

    hull.push_back(leftHull[i]);

  hull.push_back(leftHull[lowerTangentLeft]);

  for (int i = lowerTangentRight; i != upperTangentRight; i = (i + 1) % rightHull.size())

    hull.push_back(rightHull[i]);

  hull.push_back(rightHull[upperTangentRight]); }
```

This function recursively finds the convex hull for a subset of points defined by the indices left and right. It divides the set of points into two halves, finds the convex hull for each half, and then merges the convex hulls.

```
// Function to find the convex hull

vector<Point> convexHull(vector<Point> &points)

{

  vector<Point> hull;

  if (points.size() < 3)

    return hull;

  // Sort the points based on their x-coordinates

  sort(points.begin(), points.end(), [](const Point &p1, const Point &p2)

    { return p1.x < p2.x || (p1.x == p2.x && p1.y < p2.y); });

  // Find the convex hull using Divide-and-Conquer

  convexHullUtil(points, 0, points.size() - 1, hull);

  return hull; }
```

This function sorts the points based on their x-coordinates and then calls convexHullUtil to find the convex hull using the Divide-and-Conquer approach.

```
int main()

{

  vector<Point> points = {{1, 1}, {4, 6}, {8, 2}, {5, 4}, {2, 3}};

  cout << "Given points: [";

  for (int i = 0; i < points.size(); ++i) {

    cout << "(" << points[i].x << ", " << points[i].y << ")";
```

```
    if (i != points.size() - 1)
        cout << ", "; }
    cout << "]" << endl;
    vector<Point> hull = convexHull(points);
    cout << "Convex Hull: [";
    for (int i = 0; i < hull.size(); ++i) {
        cout << "(" << hull[i].x << ", " << hull[i].y << ")";
        if (i != hull.size() - 1)
            cout << ", "; }
    cout << "]" << endl;
    return 0; }
```

The main function initializes a vector of points, prints the given points, finds the convex hull, and prints the result.

1. **Best Case: O(n log n)** - When the points are already sorted by x-coordinate, which minimizes the time taken in sorting and finding the convex hulls.
2. **Average Case: O(n log n)** - Same as the best case.
3. **Worst Case: O(n log n)** - This is the complexity of the divide-and-conquer algorithm, which is dominated by sorting. However, the actual running time may vary depending on the distribution of points.

B. Dynamic Programming

1. Word Break

```
#include <iostream>
#include <string>
#include <unordered_set>
#include <vector>
using namespace std;
bool wordBreak(const string &s, const unordered_set<string>
&wordDict) {
    int n = s.size();
    // dp[i] indicates whether substring s[0...i-1] can be segmented
into words in the dictionary
    vector<bool> dp(n + 1, false);
    dp[0] = true; // Empty string can be segmented
    for (int i = 1; i <= n; ++i) {
        for (int j = 0; j < i; ++j) {
            // Check if substring s[j...i-1] is in the dictionary and s[0...j-1]
can be segmented
            if (dp[j] && wordDict.count(s.substr(j, i - j))) {
                dp[i] = true;
                break; // No need to check further }
        }
    }
    return dp[n]; }
```

This function takes a string s and a set of words wordDict as input. It uses dynamic programming to determine if the string s can be segmented into words from the dictionary. It iterates through each prefix of the string and checks if it can be segmented.

```
int main()
{
    unordered_set<string> wordDict = {"apple", "pear", "pie"};
    string s = "applepie";
    cout << "Dictionary: [";
    for (const auto &word : wordDict) {
        cout << "\"" << word << "\", "; }
    cout << "]" << endl;
    cout << "String: \"" << s << "\"" << endl;
    if (wordBreak(s, wordDict)) {
        cout << "Result: True (\"apple pie\")" << endl; }
    else {
        cout << "Result: False" << endl; }
    return 0; }
```

The main function initializes a set of words wordDict and a string s. It prints the dictionary and the input string, then calls the wordBreak function to check if the string can be segmented. Finally, it prints the result.

1. **Best Case: O(n^2)** - If the input string is short and can be segmented using a few dictionary words.
2. **Average Case: O(n^2)** - For most practical inputs, where the input string can be segmented into words from the dictionary using dynamic programming efficiently.
3. **Worst Case: O(n^2)** - In scenarios where the input string is long and cannot be segmented efficiently using dynamic programming, leading to a full traversal of the DP table.

2. Subset Sum

```
#include <iostream>
#include <vector>
using namespace std;
bool subsetSum(const vector<int> &nums, int target) {
    int n = nums.size();
    // dp[i][j] indicates whether there is a subset of nums[0...i-1] that
adds up to j
    vector<vector<bool>> dp(n + 1, vector<bool>(target + 1, false));
    // An empty subset can always achieve a sum of 0
    for (int i = 0; i <= n; ++i) {
        dp[i][0] = true; }
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= target; ++j) {
            if (j < nums[i - 1]) {
                // If the current number is greater than the target, we cannot
include it
                dp[i][j] = dp[i - 1][j]; }
            else {
                // We can either include the current number or exclude it
                dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i - 1]]; }
        }
```

```
    }
    return dp[n][target]; }
```

This function takes a vector of integers nums and an integer target as input. It uses dynamic programming to determine if there is a subset of nums that adds up to the target sum.

```
int main()
{
    vector<int> nums = {3, 2, 7};
    int target = 5;
    cout << "Set: [";
    for (int i = 0; i < nums.size(); ++i) {
        cout << nums[i];
        if (i != nums.size() - 1)
            cout << ", "; }
    cout << "] Target sum: " << target << endl;
    if (subsetSum(nums, target)) {
        cout << "Result: True (Subset: [";
        bool first = true;
        for (int num : nums) {
            if (target >= num && subsetSum(nums, target - num)) {
                if (!first)
                    cout << ", ";
                cout << num;
                target -= num;
                first = false; }
        }
        cout << "])" << endl; }
    else {
        cout << "Result: False" << endl; }
    return 0; }
```

The main function initializes a vector of integers nums and an integer target. It prints the given set and the target sum, then calls the subsetSum function to check if there exists a subset with the target sum. Finally, it prints the result along with the subset if it exists.

1. **Best Case: O(n * target)** - If the target sum is small or cannot be achieved by any subset, leading to early termination.
2. **Average Case: O(n * target)** - For most practical inputs, where the target sum can be achieved efficiently using dynamic programming.
3. **Worst Case: O(n * target)** - In scenarios where the target sum is large and can be achieved by many subsets, leading to a full traversal of the DP table.