THE UNIVERSITY OF VERMONT
COLLEGE OF ENGINEERING &
MATHEMATICAL SCIENCES

COMPLEXITY

Examples and practice

CS 124 / Department of Computer Science

Sum of cubes. What is complexity?

```
int sumOfCubes(int n) {
    int partialSum = 0;
    for (int i = 1; i <= n; ++i) {
        partialSum += i * i * i;
    }
    return partialSum;
}
```

Sum of cubes. What is complexity?

```
int sumOfCubes(int n) {
    int partialSum = 0;
    for (int i = 1; i <= n; ++i) {
        partialSum += i * i * i;
    }
    return partialSum;
}
```

Time complexity: $O(n)$

Sum of 2D array. What is complexity?

```
int twoDArraySum(std::vector<std::vector<int>> a) {
    int result = 0;
    for (int i = 0; i < a.size(); i++) {
        std::vector<int> row = a[i];
        for (int j = 0; j < a[i].size(); j++) {
            result = result + row[j];
        }
    }
    return result;
}
```

Sum of 2D array. What is complexity?

```
int twoDArraySum(std::vector<std::vector<int>> a) {
    int result = 0;
    for (int i = 0; i < a.size(); i++) {
        std::vector<int> row = a[i];
        for (int j = 0; j < a[i].size(); j++) {
            result = result + row[j];
        }
    }
    return result;
}
```

Time complexity: $O(n^2)$

Maximum Subsequence Sum

Suppose you have a sequence of integers (in a vector) and you want to know the maximum sum of any contiguous subsequence. For example, the maximum subsequence sum of -2, 11, -4, 13, -5, -2 is 20 (the 11, -4, 13 subsequence).

We're going to analyze four different algorithms that solve this problem, and determine their time complexity.

Max Sub Sum: Algorithm #1

```
int maxSubSum1(const std::vector<int> &a) {
    int maxSum = 0;
    for (int i = 0; i < a.size(); ++i) {  // left boundary
        for (int j = i; j < a.size(); ++j) {  // right boundary
            int thisSum = 0;
            for (int k = i; k <= j; ++k) {
                thisSum += a[k];
            }
            if (thisSum > maxSum) {
                maxSum = thisSum;
            }
        }
    }
    return maxSum;
}
```

Max Sub Sum: Algorithm #1

```
int maxSubSum1(const std::vector<int> &a) {
    int maxSum = 0;
    for (int i = 0; i < a.size(); ++i) {  // left boundary
        for (int j = i; j < a.size(); ++j) {  // right boundary
            int thisSum = 0;
            for (int k = i; k <= j; ++k) {
                thisSum += a[k];
            }
            if (thisSum > maxSum) {
                maxSum = thisSum;
            }
        }
    }
    return maxSum;
}
```

Time complexity: $O(n^3)$

Max Sub Sum: Algorithm #2

```
int maxSubSum2(const std::vector<int> &a) {
    int maxSum = 0;
    for (int i = 0; i < a.size(); ++i) {
        int thisSum = 0;
        for (int j = i; j < a.size(); ++j) {
            thisSum += a[j];
            if (thisSum > maxSum) {
                maxSum = thisSum;
            }
        }
    }
    return maxSum;
}
```

Max Sub Sum: Algorithm #2

```
int maxSubSum2(const std::vector<int> &a) {
    int maxSum = 0;
    for (int i = 0; i < a.size(); ++i) {
        int thisSum = 0;
        for (int j = i; j < a.size(); ++j) {
            thisSum += a[j];
            if (thisSum > maxSum) {
                maxSum = thisSum;
            }
        }
    }
    return maxSum;
}
```

Time complexity: $O(n^2)$

Max Sub Sum: Algorithm #3

```
int maxSumRec(const std::vector<int> &a, int left, int right) {
    if (left == right) {
        if (a[left] > 0) {
            return a[left];
        } else {
            return 0;
        }
    }
    int center = (left + right) / 2; // divide into two halves
    int maxLeftSum = maxSumRec(a, left, center);  // recursive call
    int maxRightSum = maxSumRec(a, center + 1, right);  // recursive call
    int maxLeftBorderSum = 0, leftBorderSum = 0;
    for (int i = center; i >= left; --i) {
        leftBorderSum += a[i];
        if (leftBorderSum > maxLeftBorderSum) {
            maxLeftBorderSum = leftBorderSum;
        }
    }
    int maxRightBorderSum = 0, rightBorderSum = 0;
    for (int j = center + 1; j <= right; ++j) {
        rightBorderSum += a[j];
        if (rightBorderSum > maxRightBorderSum) {
            maxRightBorderSum = rightBorderSum;
        }
    }
    int combined = maxLeftBorderSum + maxRightBorderSum;
    return std::max(maxLeftSum, std::max(maxRightSum, combined));
}
```

Max Sub Sum: Algorithm #3

```
int maxSumRec(const std::vector<int> &a, int left, int right) {
    if (left == right) {
        if (a[left] > 0) {
            return a[left];
        } else {
            return 0;
        }
    }
    int center = (left + right) / 2; // divide into two halves
    int maxLeftSum = maxSumRec(a, left, center);  // recursive call
    int maxRightSum = maxSumRec(a, center + 1, right);  // recursive call
    int maxLeftBorderSum = 0, leftBorderSum = 0;
    for (int i = center; i >= left; --i) {
        leftBorderSum += a[i];
        if (leftBorderSum > maxLeftBorderSum) {
            maxLeftBorderSum = leftBorderSum;
        }
    }
    int maxRightBorderSum = 0, rightBorderSum = 0;
    for (int j = center + 1; j <= right; ++j) {
        rightBorderSum += a[j];
        if (rightBorderSum > maxRightBorderSum) {
            maxRightBorderSum = rightBorderSum;
        }
    }
    int combined = maxLeftBorderSum + maxRightBorderSum;
   return std::max(maxLeftSum, std::max(maxRightSum, combined));
}
```

Time complexity: $O(n \log n)$

Max Sub Sum: Algorithm #4

```
int kadanesAlgo(const std::vector<int> &a) {
    int bestSum = 0;
    int currentSum = 0;
    for (int i = 0; i < a.size(); ++i) {
        currentSum = std::max(0, currentSum + a[i]);
        bestSum = std::max(bestSum, currentSum);
    }
    return bestSum;
}
```

Time complexity: $O(n)$

Max Sub Sum: Comparison

Algorithm	Lines of code	Time complexity
#1 brute force	10	$O(n^3)$
#2 improved brute force	9	$O(n^2)$
#3 recursive divide and conquer	21	$O(n \log n)$
#4 Kadane's algorithm	7	$O(n)$

Different kinds of complexity

Time Complexity
How does run time vary with input?

Auxiliary complexity
Auxiliary complexity measures the amount of temporary memory needed by the algorithm — not including inputs. How do auxiliary requirements vary with input?

Space Complexity
How does space requirement vary with input? Space complexity does include the input variables.

What is complexity?

```
int sum(int a, int b, int c) {
    int sum = a + b + c;
    return sum;
}
```

What is complexity?

```
int sum(int a, int b, int c) {
    int sum = a + b + c;
    return sum;
}
```

Time complexity: $O(1)$

Auxiliary complexity: $O(1)$

Space complexity: $O(1)$

What is complexity?

```
int sum(std::vector<int> nums) {
    int sum = 0;
    for (int i = 0; i < nums.size(); ++i){
        sum += nums[i];
    }
    return sum;
}
```

What is complexity?

```
int sum(std::vector<int> nums) {
    int sum = 0;
    for (int i = 0; i < nums.size(); ++i){
        sum += nums[i];
    }
    return sum;
}
```

Time complexity: O(n)

Auxiliary complexity: O(1)

Space complexity: O(n)

Complexities with Multiple Variables

Sometimes there are two independent variables at play that will affect the complexity of the algorithm. For example, having two vectors of different sizes.

In this case, you will need two variables to describe the complexity, *e.g*. $O(N + K)$, $O(N \times K)$, *etc*.

If it is not clear what the variables represent, you will need to state it explicitly.

What is complexity?

```
vector<vector<int>> mult(vector<int> nums1, vector<int> nums2) {
    vector<vector<int>> product;
    product.resize(nums1.size());
    for (int i = 0; i < nums1.size(); ++i) {
        product[i].resize(nums2.size());
        for (int j = 0; j < nums2.size(); ++j) {
            product[i][j] = nums1[i] * nums2[j];
        }
    }
    return product;
}
```

Example

Here's an example. Let
`nums1 = {1, 2, 3, 4}` and
`nums2 = {5, 0, 2}`,
then `product` will look like this:

$$\text{m rows}\left\{\overbrace{\begin{array}{ccc} 5 & 0 & 2 \\ 10 & 0 & 4 \\ 15 & 0 & 6 \\ 20 & 0 & 8 \end{array}}^{\text{n columns}}\right.$$

What is complexity?

```
vector<vector<int>> mult(vector<int> nums1, vector<int> nums2) {
    vector<vector<int>> product;
    product.resize(nums1.size());
    for (int i = 0; i < nums1.size(); ++i) {
        product[i].resize(nums2.size());
        for (int j = 0; j < nums2.size(); ++j) {
            product[i][j] = nums1[i] * nums2[j];
        }
    }
    return product;
}
```

What is complexity?

```
vector<vector<int>> mult(vector<int> nums1, vector<int> nums2) {
    vector<vector<int>> product;
    product.resize(nums1.size());
    for (int i = 0; i < nums1.size(); ++i) {
        product[i].resize(nums2.size());
        for (int j = 0; j < nums2.size(); ++j) {
            product[i][j] = nums1[i] * nums2[j];
        }
    }
    return product;
}
```

Time complexity: $O(m \times n)$

Auxiliary complexity: $O(m \times n)$

Space complexity: $O(m \times n)$