Шинжлэх Ухаан Технологийн Их Сургууль
Мэдээлэл, Холбооны Технологийн Сургууль

F.CSM301 Алгоритмын шинжилгээ ба зохиомж

Бие даалт №2

Шалгасан багш: Д.Батмөнх /F.SW811/
Гүйцэтгэсэн оюутан: М.Баяржавхлан /B222270801/

Улаанбаатар хот
2024 он

Агуулга

Удиртгал

Энэхүү хичээлийн хүрээнд Divide-and-Conquer, Dynamic Programming, Greedy Algorithm гэсэн аргуудын үндсэн алгоритмын зарчмыг судаллаа. Эдгээр аргачлалуудыг ашиглан төрөл бүрийн бодлогуудыг хэрхэн шийдвэрлэх тухай авч үзсэн. Судалгааны явцад алгоритмуудын онцлог шинж чанар, үр ашиг болон хэрэглээний давуу талуудыг илүү ойлгосон.

1 Жишээ бодлого болон тайлбар

1.1 Divide-and-Counqer

```java
public class Solution {

    private int majorityElementRec(int[] nums, int left, int right) {
        if (left == right) {
            return nums[left];
        }

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

        int leftMajority = majorityElementRec(nums, left, mid);
        int rightMajority = majorityElementRec(nums, mid + 1, right);

        if (leftMajority == rightMajority) {
            return leftMajority;
        }

        int leftCount = countInRange(nums, leftMajority, left, right);
        int rightCount = countInRange(nums, rightMajority, left, right);

        return leftCount > rightCount ? leftMajority : rightMajority;
    }

    private int countInRange(int[] nums, int val, int left, int right) {
        int count = 0;
        for (int i = left; i <= right; i++) {
            if (nums[i] == val) {
                count++;
            }
        }
        return count;
    }

    public int majorityElement(int[] nums) {
        return majorityElementRec(nums, 0, nums.length - 1);
    }
}
```

Listing 1: Majority Element

Энэхүү бодлого нь өгөгдсөн массивын дотроос нийт элементүүдийн 1/2-ээс дээш тоогоор давтагдсан элементийг олох зорилготой. Уг бодлогыг бодохдоо **Divide and Conquer** аргыг хэрэгжүүлсэн ба энэхүү арга нь бодлогыг бодохдоо олон жижиг хэсэг болгон хуваах тооцоолол дуусаны дараа рекурсивээр жижиг бодлогуудын үр дүнг дуудан нэгтгэж эцсийн үр дүнг гаргадаг юм.
Divide-and-Conquer хэрэгжилт нь:

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

        int leftMajority = majorityElementRec(nums, left, mid);
        int rightMajority = majorityElementRec(nums, mid + 1, right);
```

Өгөгдсөн массивын дунд цэгийг олж зүүн болон баруун хэсэгт хуваан тухайн элементийн давтагдсан байдлыг тооцоолон үр дүнг буцааж байгаа.

1.2 Dynamic Programming

```
public class Knapsack {
    public int knapsack(int W, int[] weights, int[] values, int n) {
        int[][] dp = new int[n + 1][W + 1];

        for (int i = 1; i <= n; i++) {
            for (int j = 0; j <= W; j++) {
                if (weights[i - 1] <= j) {
                    dp[i][j] = Math.max(dp[i - 1][j],
                    dp[i - 1][j - weights[i - 1]] + values[i - 1]);
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }
        return dp[n][W];
    }
}
```

Listing 2: Knapsack Problem

Энэхүү бодлого нь хулгайч дэлгүүрийн бараанаас өөрийн цүнхнийхээ хэмжээнд тааруулан хамгийн үнэ цэнтэй байх бараагаар цүнхээ дүүргэх нь гол зорилго юм. Уг бодлогыг динамик программчлалын арга ашиглан бодсон ба энэхүү арга нь бодлогын явцад өмнө нь тооцоолсон утгаа дараа, дараагийн тооцоололдоо ашигладгаараа онцлогтой.
Динамик программчлалын хэрэгжилт нь:

```
    dp[i][j] = Math.max(dp[i-1][j], dp[i-1][j-weights[i-1]]+values[i-1]);
```

Энэ нь хулгайч бараагаар цүнхээ дүүргэх явцдаа одоо сонгон авах гэж байгаа бараа ба өмнө нь сонгон авч цүнхэндээ хийсэн байгаа бараа хоёрын аль нь үнэ цэнэ ихтэй байгааг шалгаж байгаа нь динамик программчлалын хэрэгжилт юм.

1.3 Greedy Algorithm

```
public class Solution {
    public int maxProfit(int[] prices) {
        int maxProfit = 0;
        for (int i = 0; i < prices.length - 1; i++) {
            if (prices[i + 1] > prices[i]) {
                maxProfit += prices[i + 1] - prices[i];
            }
        }

        return maxProfit;
    }
}
```

Listing 3: Best Time to Buy and Sell Stock

Энэхүү бодлого нь өдөр бүр өөр өөр үнэтэй хувьцаат массив өгөгдөх ба тухайн өдөр зарах болон худалдан авах шийдвэрийг хамгийн ашигтай байдлаар олж тухайн өдрүүдийн ашгийг тооцоолох юм. Энэ бодлогыг бодохдоо **Greedy algorithm** буюу **хомхойлох аргыг** ашигласан ба энэхүү алгоритм нь тухайн үеийн хамгийн зөв гэсэн сонголтыг авч тухайн сонгон авсан утга нь уг бодлогын үр дүнд хамгийн сайнаар нөлөөлнө гэж найдан авсан утга юм.

Хомхойлох аргын хэрэгжилт нь:

```
for (int i = 0; i < prices.length - 1; i++) {
        if (prices[i + 1] > prices[i]) {
            maxProfit += prices[i + 1] - prices[i];
        }
    }
```

Энэхүү алгоритм нь тухайн үед зөв гэсэн шийдвэрийг авч байгаа ба массивын эхний элементээс дуусах хүртэл дараалсан хоёр өдрийг хооронд нь харьцуулан шалгах замаар хэрэгжсэн. Хамгийн гол нь эцсийн үр дүн ямар гарахаас үл шалтгаалан тухайн нөхцөлд л хамгийн зөв шийдвэрийг гаргах нь чухал юм.

2 Алгоритмууд хоорондын ялгаа

2.1 Recursion vs Divide-and-Conquer

Бодлого: Өгөгдсөн массиваас хамгийн их элементийг олох.

```
    public class MaxElement {
    public int findMax(int[] arr, int n) {
        if (n == 1) {
            return arr[0];
        }

        int maxInRest = findMax(arr, n - 1);

        return Math.max(maxInRest, arr[n - 1]);
    }
}
```

Listing 4: Массив дахь хамгийн их утгатай элементийг олох (Recursion)

```
    public class MaxElement {
    public int findMax(int[] arr, int left, int right) {
        if (left == right) {
            return arr[left];
        }

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

        int leftMax = findMax(arr, left, mid);
        int rightMax = findMax(arr, mid + 1, right);

        return Math.max(leftMax, rightMax);
    }
}
```

Listing 5: Массив дахь хамгийн их утгатай элементийг олох (Divide and Conquer)

Тайлбар: Recursion нь жижиг асуудлыг шийдэхэд илүү тохиромжтой бөгөөд ихэнхдээ нэг асуудал дээр ажилладаг бол Divide and Conquer нь бие даасан олон дэд асуудлыг шийддэг ба бодлогыг бодохдоо Recursion-ийг ашиглах нь уг алгоритмын нэг хэсэг болдог.[1]

2.2 Divid-and-Conquer vs Dynamic Programming

```
    public class MaxSubarray {
    public int maxSubArray(int[] nums) {
        return maxSubArrayRec(nums, 0, nums.length - 1);
    }

    private int maxSubArrayRec(int[] nums, int left, int right) {
        if (left == right) return nums[left]; // Base case

        int mid = left + (right - left) / 2;
        int leftMax = maxSubArrayRec(nums, left, mid);
        int rightMax = maxSubArrayRec(nums, mid + 1, right);
        int crossMax = maxCrossingSum(nums, left, mid, right);

        return Math.max(Math.max(leftMax, rightMax), crossMax);
    }
```

```
    private int maxCrossingSum(int[] nums, int left, int mid, int right) {
        int sum = 0, leftSum = Integer.MIN_VALUE, rightSum = Integer.MIN_VALUE;

        for (int i = mid; i >= left; i--) {
            sum += nums[i];
            leftSum = Math.max(leftSum, sum);
        }

        sum = 0;
        for (int i = mid + 1; i <= right; i++) {
            sum += nums[i];
            rightSum = Math.max(rightSum, sum);
        }

        return leftSum + rightSum;
    }
}
```

Listing 6: Хамгийн их нийлбэртэй залгаа дэд массив (Divide-and Conquer)

```
    public class MaxSubarrayDP {
    public int maxSubArray(int[] nums) {
        int maxSum = nums[0];
        int currentSum = nums[0];

        for (int i = 1; i < nums.length; i++) {
            currentSum = Math.max(nums[i], currentSum + nums[i]);
            maxSum = Math.max(maxSum, currentSum);
        }

        return maxSum;
    }
}
```

Listing 7: Хамгийн их нийлбэртэй залгаа дэд массив (Dynamic Programming)

Тайлбар: Divide-and-Conquer нь массивыг хувааж, хагасыг нь рекурсивээр шийдэж байгаа бол Dynamic Programming нь массивыг давталт ашиглан утгуудыг тооцоолон хадгалан, үүнийгээ дараачийн тооцоололдоо ашиглах замаар бодогдсон.

2.3 Dynamic Programming vs Greedy

```
    public class CoinChangeDP {
    public int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, amount + 1);
        dp[0] = 0;

        for (int coin : coins) {
            for (int i = coin; i <= amount; i++) {
                dp[i] = Math.min(dp[i], dp[i - coin] + 1);
            }
        }

        return dp[amount] > amount ? -1 : dp[amount];
    }
}
```

Listing 8: Coin Change Problem (Dynamic Programming)

```
    public class CoinChangeGreedy {
    public int coinChange(int[] coins, int amount) {
        Arrays.sort(coins);
        int count = 0;

        for (int i = coins.length - 1; i >= 0; i--) {
            while (amount >= coins[i]) {
                amount -= coins[i];
                count++;
            }
        }

        return amount == 0 ? count : -1;
    }
}
```

Listing 9: Coin Change Problem (Greedy)

Тайлбар: Уг бодлогын гол зорилго нь өгөгдсөн дүнг массив дах зоосны үнээр хамгийн бага үйлдлээр солих юм. Хомхойлох арга нь эхлээд хамгийн том зоосыг сонгож, хамгийн бага зоосоор мөнгөн дүнг гаргахыг оролддог бол Динамик программчлал нь бүх байх боломжит утгуудыг тооцоолон хадгалах зарчмаар бодож байна.[2]

Дүгнэлт

Бидний энэхүү хичээлийн хүрээнд судалсан алгоритмууд нь өөр өөрийн онцлог болон давуу болон сул талуудтай бөгөөд аль нь илүү гэж хэлэх боломж байхгүй. Гагцхүү тухай бодлогын өгөгдөл, тооцоолох утгууд болон нөхцөл байдалдаа тааруулан ашигтай ажиллах алгоритмыг бид өөрсдөө хэрэгжүүлэх хэрэгтэй.

Ашигласан ном

[1] “GeekforGeeks сургалтын сайт.” https://www.geeksforgeeks.org/.

[2] “Leet Code бодлогын сантай сайт.” https://leetcode.com/.