탐욕법(Greedy)

물론입니다. 아래는 **탐욕법**(Greedy Algorithm)에 대한 **전체적인 정리**이며, 다음과 같은 구성으로 정리했습니다:

- 📚 탐욕법의 정의와 핵심 원리
- ✅ 탐욕법의 장점과 한계
- 📌 탐욕법의 주요 문제 유형 (예: 편안한 정렬, 최소 비용, 최대 수익 등)
- 💡 예시 문제 (예: 편안한 정렬, 다이아몬드 문제, 최소 비용 배달 등)
- 💻 C++ 예시 코드 (STL 기반 & Non-STL 기반)
- 📚 각 문제에 대한 분석 및 정확한 적용 조건

📚 탐욕법 (Greedy Algorithm) – 전체 정리

1. 정의와 핵심 원리

탐욕법(Greedy Algorithm)은 **현재 상태에서 가장 좋은 선택**을 하며, **지금의 최적 선택을 기반으로 미래의 선택을 결정**하는 알고리즘입니다.

> ✅ 핵심 원칙: **"지금 가장 좋은 선택을 하자"**
> → 즉, **지역 최적**을 선택해 **전체 최적**으로 이어지는 경우에 성공

✅ 탐욕법의 특징

항목	설명
시간 효율성	O(n) 또는 O(n log n) 등으로 빠르게 작동 가능
간단한 구현	코드가 간단하고 직관적
전체 최적 보장 여부	❌ **보장되지 않음** (예: 편안한 정렬 문제)

> ⚠️ **주의**: 탐욕법이 **전체 최적**을 보장하지 않음.
> 예: `0/1 배낭 문제`는 탐욕법으로는 최적 해를 보장할 수 없음.

---

2. 탐욕법의 장점과 한계

항목	설명
✅ 장점	- 구현 쉬움 - 시간 효율성 높음

항목	설명
	- 실생활 문제에 잘 적용됨 (예: 최소 비용, 최대 수익)
❌ 한계	- 전체 최적을 보장하지 않음 - 문제의 구조에 따라 실패 가능 (예: 편안한 정렬 문제)
📌 적용 가능 문제	- 최소 비용 배달 - 최대 수익 문제 - 편안한 정렬 (activity selection) - 다이아몬드 문제 (dijkstra의 확장)

---

3. 주요 문제 유형 및 예시

🔷 1. 편안한 정렬 (Activity Selection Problem)

> **문제**: 여러 활동이 시간에 겹치면, 최대 수의 활동을 선택하라.

- **탐욕법 적용**: **시작 시간이 가장 빠른 활동**부터 선택
- **결과**: 최대 수의 활동 선택 가능 (전체 최적 보장)

✅ 예시

- 활동: (시작, 종료) → (1,4), (3,5), (0,6), (5,7)
- 최적 선택: (0,6), (5,7) → 2개

🔷 2. 최소 비용 배달 (Minimum Cost to Connect All Points)

> **문제**: N개의 점을 연결하여 최소 비용으로 연결 (MST 문제)

- **탐욕법 적용**: **크루스칼 알고리즘**(Kruskal) 또는 **프림 알고리즘**(Prim)
- **결과**: 최소 스패닝 트리 (MST) 생성

🔷 3. 다이아몬드 문제 (Fractional Knapsack)

> **문제**: 무게 제한이 있는 상자에 물건을 넣을 때, **최대 가치를 얻기 위해** 몇 퍼센트를 넣을지 결정

- **탐욕법 적용**: **단위 무게당 가치가 가장 높은 물건**부터 선택
- ✅ **탐욕법이 최적 해를 보장**

🔷 4. 최대 수익 문제 (Job Scheduling)

> **문제**: 여러 작업을 수행할 때, **최대 수익**을 얻기 위해 어떤 작업을 선택할지

- **탐욕법 적용**: **수익이 높은 작업**부터 선택 (시간 제약 있음)

---

4. C++ 예시 코드 (STL 기반 & Non-STL 기반)

✅ 예시 1: 편안한 정렬 (Activity Selection)

📌 문제: 시간에 겹치는 활동 중 최대 수를 선택

```cpp
// STL 기반 (C++11 이상)
#include <iostream>
#include <vector>
#include <algorithm>
#include <utility>

using namespace std;

struct Activity {
    int start, end;
    bool operator<(const Activity& a) const {
        return end < a.end; // 종료 시간 기준으로 정렬
    }
};

int maxActivities(const vector<Activity>& acts) {
    sort(acts.begin(), acts.end());
    int count = 1;
    int lastEnd = acts[0].end;

    for (int i = 1; i < acts.size(); ++i) {
        if (acts[i].start >= lastEnd) {
            count++;
            lastEnd = acts[i].end;
        }
    }
    return count;
}

// 예시 사용
int main() {
    vector<Activity> activities = {
```

```
        {1, 4}, {3, 5}, {0, 6}, {5, 7}
    };
    cout << "최대 활동 수: " << maxActivities(activities) << endl;
    return 0;
}
```

📌 Non-STL 버전 (배열 기반)

```
// Non-STL 버전 (배열 기반, for 문으로 구현)
#include <iostream>
#include <cstdio>
using namespace std;

struct Activity {
    int start, end;
};

int maxActivitiesNonSTL(int n, Activity acts[]) {
    // 시작 시간 기준 정렬 (단순히 인덱스로 정렬)
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n - i - 1; ++j) {
            if (acts[j].end > acts[j+1].end) {
                swap(acts[j], acts[j+1]);
            }
        }
    }

    int count = 1;
    int lastEnd = acts[0].end;

    for (int i = 1; i < n; ++i) {
        if (acts[i].start >= lastEnd) {
            count++;
            lastEnd = acts[i].end;
        }
    }
    return count;
}

int main() {
    Activity acts[4] = {{1,4}, {3,5}, {0,6}, {5,7}};
    cout << "최대 활동 수: " << maxActivitiesNonSTL(4, acts) << endl;
    return 0;
}
```

✅ 예시 2: 다이아몬드 문제 (Fractional Knapsack)

📌 문제: 무게 제한이 10kg인 상자에 물건 넣기

```cpp
// STL 기반
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;

struct Item {
    int value, weight;
    double ratio; // 단위 무게당 가치
    bool operator<(const Item& a) const {
        return ratio > a.ratio; // 큰 비율부터
    }
};

int fractionalKnapsack(int n, int maxWeight, vector<Item>& items) {
    sort(items.begin(), items.end());
    int totalValue = 0;
    int currentWeight = 0;

    for (int i = 0; i < n; ++i) {
        if (currentWeight + items[i].weight <= maxWeight) {
            currentWeight += items[i].weight;
            totalValue += items[i].value;
        } else {
            double fraction = (double)(maxWeight - currentWeight) /
items[i].weight;
            totalValue += items[i].value * fraction;
            break;
        }
    }
    return totalValue;
}

int main() {
    vector<Item> items = {
        {60, 10}, {100, 20}, {120, 30}
    };

    cout << "최대 수익: " << fractionalKnapsack(3, 50, items) << endl;
    return 0;
}
```

📌 Non-STL 버전

```cpp
// Non-STL 버전
#include <iostream>
#include <cstdio>
using namespace std;

struct Item {
    int value, weight, ratio;
};

int fractionalKnapsackNonSTL(int n, int maxWeight, Item items[]) {
    // 비율 정렬 (단순히 for 문으로)
    for (int i = 0; i < n; ++i) {
        for (int j = 등 0; j < n - i - 1; ++j) {
            if (items[j].ratio < items[j+1].ratio) {
                swap(items[j], items[j+1]);
            }
        }
    }

    int totalValue = 0;
    int currentWeight = 0;

    for (int i = 0; i < n; ++i) {
        if (currentWeight + items[i].weight <= maxWeight) {
            currentWeight += items[i].weight;
            totalValue += items[i].value;
        } else {
            double fraction = (double)(maxWeight - currentWeight) /
items[i].weight;
            totalValue += items[i].value * fraction;
            break;
        }
    }
    return totalValue;
}

int main() {
    Item items[3] = {
        {60, 10, 6}, {100, 20, 5}, {120, 30, 4}
    };
    cout << "최대 수익: " << fractionalKnapsackNonSTL(3, 50, items) << endl;
    return 0;
}
```

✅ 예시 3: 최소 비용 배달 (MST – 크루스칼)

📌 문제: N개의 점을 연결하여 최소 비용으로 연결

```cpp
// STL 기반 (크루스칼 알고리즘)
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
using namespace std;

struct Edge {
    int u, v, w;
    bool operator<(const Edge& e) const {
        return w < e.w;
    }
};

int find_parent(vector<int>& parent, int x) {
    if (parent[x] != x) parent[x] = find_parent(parent, parent[x]);
    return parent[x];
}

void union_set(vector<int>& parent, int x, int y) {
    int rx = find_parent(parent, x);
    int ry = find_parent(parent, y);
    if (rx != ry) parent[rx] = ry;
}

int minimumSpanningTree(int n, vector<Edge>& edges) {
    sort(edges.begin(), edges.end());
    vector<int> parent(n);
    for (int i = 0; i < n; ++i) parent[i] = i;

    int totalCost = 0;
    int edgesUsed = 0;

    for (Edge e : edges) {
        if (find_parent(parent, e.u) != find_parent(parent, e.v)) {
            union_set(parent, e.u, e.v);
            totalCost += e.w;
            edgesUsed++;
            if (edgesUsed == n - 1) break;
        }
    }
    return totalCost;
}

int main() {
    vector<Edge> edges = {
        {0,1,4}, {0,2,3}, {1,2,1}, {1,3,2}, {2,3,2}
```

```
    };
    cout << "최소 비용: " << minimumSpanningTree(4, edges) << endl;
    return 0;
}
```

📌 Non-STL 버전

```
// Non-STL 버전 (배열 기반)
#include <iostream>
#include <cstdio>
using namespace std;

struct Edge {
    int u, v, w;
};

int find_parent(int parent[], int x) {
    if (parent[x] != x) parent[x] = find_parent(parent, parent[x]);
    return parent[x];
}

void union_set(int parent[], int x, int y) {
    int rx = find_parent(parent, x);
    int ry = find_parent(parent, y);
    if (rx != ry) parent[rx] = ry;
}

int minimumSpanningTreeNonSTL(int n, Edge edges[]) {
    // 정렬 (단순히 for 문)
    for (int i = 0; i < n-1; ++i) {
        for (int j = 0; j < n-1-i; ++j) {
            if (edges[j].w > edges[j+1].w) {
                swap(edges[j], edges[j+1]);
            }
        }
    }

    int parent[1000];
    for (int i = 0; i < n; ++i) parent[i] = i;

    int totalCost = 0;
    int edgesUsed = 0;

    for (int i = 0; i < n-1; ++i) {
        if (find_parent(parent, edges[i].u) != find_parent(parent,
edges[i].v)) {
            union_set(parent, edges[i].u, edges[i].v);
            totalCost += edges[i].w;
            edgesUsed++;
```

```
            if (edgesUsed == n - 1) break;
        }
    }
    return totalCost;
}

int main() {
    Edge edges[5] = {
        {0,1,4}, {0,2,3}, {1,2,1}, {1,3,2}, {2,3,2}
    };
    cout << "최소 비용: " << minimumSpanningTreeNonSTL(4, edges) << endl;
    return 0;
}
```

---

5. 탐욕법이 성공할 조건

조건	설명
✅ 단순한 선택 기준	(예: 시간, 비용, 수익 등)
✅ 최적 하위 구조	(예: 각 선택이 전체 최적에 기여)
✅ 무게 제한 문제	(예: Fractional Knapsack)
❌ **0/1 Knapsack**	탐욕법은 최적 해를 보장하지 않음

---

6. 요약

항목	내용
🔍 핵심 원리	현재 최적 선택 → 미래 선택 기반
✅ 성공 사례	Fractional Knapsack, Activity Selection, MST
❌ 실패 사례	0/1 Knapsack, 일부 최적화 문제
💡 추천 사용	**시간 제한이 적고, 비용/수익/시간 기준이 명확한 문제**에 적합

✅ 결론

탐욕법은 **간단하고 빠르며 실생활 문제에 매우 유용**합니다.
하지만 **전체 최적을 보장하지 않기 때문에**, 문제의 구조를 잘 이해하고 **탐욕법이 적용 가능한지 판단**하는 것이 중요합니다.

> 📌 **요약**:
>
> - **탐욕법은 "지금 가장 좋은 선택"을 하며, 전체 최적을 보장하지 않음**
> - **STL과 Non-STL 모두 구현 가능**, 실무에서 유연하게 사용 가능
> - **정확한 문제 분석**이 핵심 (예: 0/1 vs Fractional)

이 정리와 코드는 **알고리즘 학습, 코딩 테스트, 실무 개발**에 매우 유용합니다.
필요 시, 각 문제에 대한 **시간 복잡도, 공간 복잡도, 테스트 케이스**도 추가로 제공 가능합니다.