# Chapter 16 Solusion

github.com/frc123/CLRS

1/10/2022

## 16.1

### 16.1-1

```
void OutputAux(const std::vector< std::vector<int> >& dp_selection,
    int i, int j, std::list<int>& output)
{
    if (dp_selection[i][j] > 0)
    {
        OutputAux(dp_selection, i, dp_selection[i][j], output);
        output.push_back(dp_selection[i][j] - 1);
        OutputAux(dp_selection, dp_selection[i][j], j, output);
    }
}

// assume intervals are sorted by finish time
std::list<int> DpActivitySelector(const std::vector<Activity>& intervals)
{
    int n, i, j, l, k, l_size;
    n = (int)(intervals.size());
    // dp_size index start by 1
    std::vector< std::vector<int> > dp_size(n + 2, std::vector<int>(n + 2, 0)),
        dp_selection(n + 2, std::vector<int>(n + 2, -1));
    // compute
    for (l = 2; l <= n + 1; ++l)
    {
        for (i = 0; i <= n + 1 - l; ++i)
        {
            j = i + l;
            for (k = i + 1; k <= j - 1; ++k)
            {
```

```
                if ((i == 0 || intervals[k - 1].s >= intervals[i - 1].f) &&
                    (j == n + 1 || intervals[k - 1].f <= intervals[j - 1].s))
                {
                    l_size = dp_size[i][k] + dp_size[k][j] + 1;
                    if (dp_size[i][j] < l_size)
                    {
                        dp_size[i][j] = l_size;
                        dp_selection[i][j] = k;
                    }
                }
            }
        }
    }
    // output
    std::list<int> output;
    OutputAux(dp_selection, 0, n + 1, output);
    return output;
}
```

The dynamic-programming algorithm runs in $O(n^3)$.

### 16.1-2

```
// assume intervals are sorted by start time
std::list<int> GreedyActivitySelector(const std::vector<Activity>& intervals)
{
    int k, m, n;
    std::list<int> activities;
    n = (int)(intervals.size());
    activities.push_front(n - 1);
    k = n - 1;
    for (m = n - 2; m >= 0; --m)
    {
        if (intervals[k].s >= intervals[m].f)
        {
            activities.push_front(m);
            k = m;
        }
    }
    return activities;
}
```

**Claim 1.** Consider any nonempty subproblem $S_k$, and let $a_m$ be an activity in $S_k$ with the latest start time. Then $a_m$ is included in some maximum-size subset of mutually compatible activities of $S_k$.

**Proof.** Let $A_k$ be a maximum-size subset of mutually compatible activities in $S_k$, and let $a_j$ be the activity in $A_k$ with the latest start time. If $a_j = a_m$, we are done, since we have shown that $a_m$ is in some maximum-size subset of mutually compatible activities of $S_k$. If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be $A_k$ but substituting $a_m$ for $a_j$. The activities in $A'_k$ are disjoint, which follows because the activities in $A_k$ are disjoint, $a_j$ is the last activity in $A_k$ to start, and $s_m \geq s_j$. Since $|A'_k| = |A_k|$, we conclude that $A'_k$ is a maximum-size subset of mutually compatible activities of $S_k$, and it includes $a_m$. □

## 16.1-3

**1. selecting the compatible activity of least duration**

| i | 1 | 2 | 3 |
|---|---|---|---|
| $s_i$ | 1 | 3 | 4 |
| $f_i$ | 4 | 5 | 7 |

By this approach, the solution will be $\{a_2\}$. However, the optimal solutiopn is $\{a_1, a_3\}$.

**2. selecting the compatible activity that overlaps the fewest other remaining activities**

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $s_i$ | 1 | 1 | 1 | 1 | 2 | 3 | 4 | 5 | 5 | 5 | 6 |
| $f_i$ | 2 | 3 | 3 | 3 | 4 | 5 | 6 | 7 | 7 | 7 | 7 |

By this approach, the solution will include $a_7$. However, the optimal solution is $\{a_1, a_6, a_8, a_{11}\}$, and $a_7$ is not compatible with the optimal solution.

**3. selecting the compatible activity with the earliest start time**

| i | 1 | 2 | 3 |
|---|---|---|---|
| $s_i$ | 2 | 3 | 1 |
| $f_i$ | 3 | 4 | 5 |

By this approach, the solution will be $\{a_3\}$. However, the optimal solutiopn is $\{a_1, a_2\}$.

## 16.1-4

```
struct Element
{
    int interval;
    std::list< std::list<int> >::iterator list;

    Element(int interval, std::list< std::list<int> >::iterator list)
        : interval(interval), list(list) {}
};
```

```cpp

// assume intervals are sorted by finish time
std::list< std::list<int> > IntervalGraphColoring(const std::vector<Activity>& intervals)
{
    int i, n;
    n = (int)(intervals.size());
    std::list< std::list<int> > collection;
    auto heap_cmp = [&intervals](const Element& a, const Element& b) {
        return intervals[a.interval].s < intervals[b.interval].s;
    };
    std::priority_queue<Element, std::vector<Element>, decltype(heap_cmp)> heap(heap_cmp);
    std::list< std::list<int> >::iterator curr_list;
    collection.emplace_front();
    curr_list = collection.begin();
    curr_list->push_front(n - 1);
    heap.emplace(n - 1, curr_list);
    for (i = n - 2; i >= 0; --i)
    {
        if (intervals[i].f <= intervals[heap.top().interval].s)
        {
            curr_list = heap.top().list;
            curr_list->push_front(i);
            heap.pop();
        }
        else
        {
            collection.emplace_front();
            curr_list = collection.begin();
            curr_list->push_front(i);
        }
        heap.emplace(i, curr_list);
    }
    return collection;
}
```

### 16.1-5

This algorithm is actually a revision from 16.1-1.

```cpp
// assume intervals are sorted by finish time
std::list<int> DpActivitySelector(const std::vector<Activity>& activities)
```

```
{
    int n, i, j, l, k, l_size;
    n = (int)(activities.size());
    // dp_size index start by 1
    std::vector< std::vector<int> > dp_size(n + 2, std::vector<int>(n + 2, 0)),
        dp_selection(n + 2, std::vector<int>(n + 2, -1));
    // compute
    for (l = 2; l <= n + 1; ++l)
    {
        for (i = 0; i <= n + 1 - l; ++i)
        {
            j = i + l;
            for (k = i + 1; k <= j - 1; ++k)
            {
                if ((i == 0 || activities[k - 1].s >= activities[i - 1].f) &&
                    (j == n + 1 || activities[k - 1].f <= activities[j - 1].s))
                {
                    l_size = dp_size[i][k] + dp_size[k][j] + activities[k - 1].v;
                    if (dp_size[i][j] < l_size)
                    {
                        dp_size[i][j] = l_size;
                        dp_selection[i][j] = k;
                    }
                }
            }
        }
    }
    // output
    std::list<int> output;
    OutputAux(dp_selection, 0, n + 1, output);
    return output;
}
```

# 16.2

## 16.2-1

Suppose that items are sorted by value per pound from high to low.

$$v_1/w_1 \geq v_2/w_2 \geq v_3/w_3 \geq \cdots \geq v_{n-1}/w_{n-1} \geq v_n/w_n \quad .$$

Denote $a_i$ as the item $i$. Let $S_k = \{a_i : k \leq i \leq n\}$.

**Claim 2.** Consider any nonempty subproblem $S_k$. Then as much as possible $a_k$ is included in some maximum-value fractional knapsack composed by some items in $S_k$ where the items in the knapsack weigh at most $W$ pounds.

**Proof.** Let $A_k$ be a maximum-value fractional knapsack composed by some items in $S_k$ where the items in the knapsack weigh at most $W$ pounds. Let $\alpha = \max(a_k, W)$. Let $\beta$ be the number of pounds of $a_k$ in $A_k$. Note that $\alpha \geq \beta$ must be true. If $\alpha = \beta$, then we are done. If $\alpha > \beta$, then we replace any $\alpha - \beta$ pounds non-$a_k$ items in $A_k$ with the same amount of $a_k$. Now we have a knapsack that contains at least the value of the original knapsack (actually, the knapsack after replacement must have the same value as the original knapsack; otherwise, we have a contradiction). □

## 16.2-2

Denote $a_i$ as the item $i$. Let $S_k = \{a_i : k \leq i \leq n\}$. Denote $c[i, j]$ as the maximum value in the 0-1 knapsack composed by some items in $S_i$ where the items in the knapsack at most $j$ pounds. We have the following recursive solution

$$c[i,j] = \begin{cases} 0 & \text{if } i = n \text{ and } j < w_i \quad , \\ v_i & \text{if } i = n \text{ and } j \geq w_i \quad , \\ c[i+1, j] & \text{if } i < n \text{ and } j < w_i \quad , \\ \max(c[i+1, j], c[i+1, j - w_i] + v_i) & \text{if } i < n \text{ and } j \geq w_i \quad . \end{cases}$$

```
std::list<int> ZeroOneKnapsack(const std::vector<Item>& items, int maximum_weight)
{
    int n, i, j;
    n = (int)(items.size());
    std::vector< std::vector<int> > dp_value(n, std::vector<int>(maximum_weight + 1));
    std::vector< std::vector<bool> > dp_put(n, std::vector<bool>(maximum_weight + 1));
    std::list<int> output;
    // compute
    i = n - 1;
    for (j = 0; j <= maximum_weight; ++j)
    {
        if (j >= items[i].w)
        {
            dp_value[i][j] = items[i].v;
            dp_put[i][j] = true;
        }
        else
        {
```

```
19              dp_value[i][j] = 0;
20              dp_put[i][j] = false;
21          }
22      }
23      for (i = n - 2; i >= 0; --i)
24      {
25          for (j = 0; j <= maximum_weight; ++j)
26          {
27              if (j >= items[i].w &&
28                  dp_value[i + 1][j - items[i].w] + items[i].v > dp_value[i + 1][j])
29              {
30                  dp_value[i][j] = dp_value[i + 1][j - items[i].w] + items[i].v;
31                  dp_put[i][j] = true;
32              }
33              else
34              {
35                  dp_value[i][j] = dp_value[i + 1][j];
36                  dp_put[i][j] = false;
37              }
38          }
39      }
40      // output
41      j = maximum_weight;
42      for (i = 0; i < n; ++i)
43      {
44          if (dp_put[i][j])
45          {
46              output.push_back(i);
47              j -= items[i].w;
48          }
49      }
50      return output;
51  }
```

### 16.2-3

Suppose that items are sorted by value from high to low.

$$v_1 \geq v_2 \geq v_3 \geq \cdots \geq v_{n-1} \geq v_n \quad .$$

Then we have

$$w_1 \leq w_2 \leq w_3 \leq \cdots \leq w_{n-1} \leq w_n \quad .$$

Denote $a_i$ as the item $i$. Let $S_k = \{a_i : k \leq i \leq n\}$.

**Claim 3.** Consider any nonempty subproblem $S_k$ with some nonempty solution. Then $a_k$ is included in some maximum-value subset (0-1 knapsack) of $S_k$ where all the items in the subset weigh at most $W$ pounds.

**Proof.** Let $A_k$ be a maximum-value subset of $S_k$ where all the items in the subset weigh at most $W$ pounds. Let $a_j$ be the item in $A_k$ with the largest value. Then $a_j$ has the smallest weight among the items in $A_k$. If $a_j = a_k$, then we are done. If $a_j \neq a_k$, let the set $A'_k = A_k - \{a_j\} \cup \{a_k\}$. Since items are sorted by value from high to low, $a_k$ has the largest value and the smallest weight among the items in $S_k$. Since $A_k, A'_k \subseteq S_k$, the total values in $A'_k$ must be greater than or equal to (actually, must be equal to) $A_k$, and the total weights in $A'_k$ must be smaller than or equal to (actually, must be equal to) $A_k$, which means all the items $A'_k$ weigh at most $W$ pounds also. We conclude that $A'_k$ is a maximum-value subset of $S_k$ where all the items in the subset weigh at most $W$ pounds, and $a_k \in A'_k$. □

```
// assume v mono decreasing and w mono increasing
std::list<int> ZeroOneKnapsack(const std::vector<Item>& items, int maximum_weight)
{
    int k, n;
    std::list<int> output;
    n = (int)(items.size());
    for (k = 0; k < n; ++k)
    {
        if (items[k].w <= maximum_weight)
        {
            maximum_weight -= items[k].w;
            output.push_back(k);
        }
        else
        {
            break;
        }
    }
    return output;
}
```

## 16.2-4

We start at the beginning and go as far as possible each time. In other words, keep going until the water is not enough to get to the next stop.

Denote $a_i$ as the $i$th place at which he can refill his water. Denote $d_i$ as distance from the starting point to $a_i$. Assume that

$$d_1 < d_2 < d_3 \cdots < d_{n-1} < d_n \quad .$$

Let $S_k = \{a_i : k < i \leq n\}$, which is the subproblem of minimize the number of stops from $a_k$. Assume $m > 0$. Let $C_k = \{a_i \in S_k : d_i \leq d_k + m\}$, which contains the candidates to the next stop from $a_k$.

**Claim 4.** Consider any nonempty subproblem $S_k$ where $k \neq n$, and let $a_l$ be the stop in $C_k$ with the greatest distance from $a_k$. Then $a_l$ is included in some minimum-size subset $B$ of $S_k$ such that $a_k, a_n \in B$ and

$$\forall a_i \in B, \exists a_j \in B \text{ such that } |d_j - d_i| \leq m \quad .$$

**Proof.** Let $A_k$ be a minimum-size subset of $S_k$ such that $a_k, a_n \in A_k$ and

$$\forall a_i \in A_k, \exists a_j \in A_k \text{ such that } |d_j - d_i| \leq m \quad .$$

Let $a_t$ be the stop in $A_k$ with the smallest distance from $a_k$. If $a_t = a_l$, then we are done. If $a_t \neq a_l$, let the set $A'_k = A_k - \{a_t\} \cup \{a_l\}$. By the definition of $C_k$, we have $a_t \in C_k$, so $d_t < d_l$. We claim that $a_t \neq a_n$ since

$$a_t = a_n \Longrightarrow d_t = d_n \Longrightarrow d_n < d_l$$

which leads to a contradiction. Since $a_l \in C_k$, $d_l - d_k \leq m$. Since $a_t \in A_k$, there exists $a_j$ such that $d_j - d_t \leq m$, so $d_j - d_l \leq m$. We conclude that $A'_k$ is a minimum-size subset of $S_k$ such that $a_k, a_n \in A'_k$ and

$$\forall a_i \in A'_k, \exists a_j \in A'_k \text{ such that } |d_j - d_i| \leq m \quad .$$

□

## 16.2-5

Let the left endpoint of the first interval be the leftmost point in the set, let the left endpoint of the second interval be the leftmost point in the set that is not contained by the first interval, let the left endpoint of the third interval be the leftmost point in the set that is not contained by the first interval and the second interval, and so forth.

Assume $x_1, x_2, \cdots, x_n$ are strictly monotonous increasing (by sorting and removing repeated elements):

$$x_1 < x_2 < x_3 < \cdots < x_{n-1} < x_n \quad .$$

Let $S_k = \{x_i : k \leq i \leq n\}$, which is the subproblem of minimize the number of set of unit-length closed intervals that contains all points in $S_k$.

**Claim 5.** Consider any nonempty subproblem $S_k$. Then $[x_k, x_k + 1]$ is included in some minimum-size set of unit-length closed intervals that contains all points in $S_k$.

**Proof.** Actually, $[x_k, x_k + 1]$ must be included in any set of unit-length closed intervals that contains all points in $S_k$. Otherwise, $x_k$ cannot be contained, which leads to a contradiction. □

### 16.2-6

We apply weighted selection in worst-case linear time to this problem.

```
// return list of pair where the first element of the pair is
// the index of item in the container after the funtion returns
// and the second element of the pair is the weight of the item in the knapsack
std::list< std::pair<int, int> > FractionalKnapsack
    (std::vector<Item>& items, int maximum_weight)
{
    std::list< std::pair<int, int> > output;
    int weight_sum = 0;
    for (Item& item : items)
    {
        weight_sum += item.w;
    }
    if (weight_sum <= maximum_weight)
    {
        // put all items into kanpsack
        for (size_t i = 0; i < items.size(); ++i)
        {
            output.emplace_back(i, items[i].w);
        }
    }
    else
    {
        for (Item& item : items)
        {
            item.v_div_w = (double)(item.v) / (item.w);
        }
        // note that a partition around the pivot will be performed also
        cotl::WeightedSelect(items.begin(), items.end(), maximum_weight - 1,
            [](const Item& a) { return a.w; },
            [](const Item& a, const Item& b) { return b.v_div_w - a.v_div_w; });
        for (size_t i = 0; i < items.size(); ++i)
        {
            int pick_weight = std::min(maximum_weight, items[i].w);
            maximum_weight -= pick_weight;
            output.emplace_back(i, pick_weight);
            if (maximum_weight == 0)
                break;
```

```
38            }
39        }
40        return output;
41    }
```

### 16.2-7

We put each of $A$ and $B$ into an array and sort these two arrays.

**Claim 6.** If $a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_{n-1} \leq a_n$ and $b_1 \leq b_2 \leq b_3 \leq \cdots \leq b_{n-1} \leq b_n$, then $\prod_{i=1}^{n} a_i^{b_i}$ is the maximum payoff.

**Proof.** Suppose that $a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_{n-1} \leq a_n$ and $b_1 \leq b_2 \leq b_3 \leq \cdots \leq b_{n-1} \leq b_n$. Suppose that $\prod_{i=1}^{n} a_i^{b_i}$ is not the maximum payoff, for the purpose of contradiction. Then there exists a sequence $(b'_n)$ difference to $(b_n)$ such that $\prod_{i=1}^{n} a_i^{b'_i}$ is the maximum payoff, which means $\prod_{i=1}^{n} a_i^{b'_i} > \prod_{i=1}^{n} a_i^{b_i}$. Then there exists integers $i, j \in [1, n]$ such that $a_i < a_j$ and $b'_i > b'_j$. Consider the difference between $a_i^{b'_i} a_j^{b'_j}$ and $a_i^{b'_j} a_j^{b'_i}$. Since $a_i^{b'_i} a_j^{b'_j} = a_i^{b'_j} a_j^{b'_j} a_i^{b'_i - b'_j}$ and $a_i^{b'_j} a_j^{b'_i} = a_i^{b'_j} a_j^{b'_j} a_j^{b'_i - b'_j}$, we have $a_i^{b'_i} a_j^{b'_j} < a_i^{b'_j} a_j^{b'_i}$, which contradicts to $\prod_{i=1}^{n} a_i^{b'_i}$ is the maximum payoff. □

**Updating...**