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

```cpp
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) {}
};
```

```

// 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.

```
// 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 β 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). $\square$

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
        {
```

```
                dp_value[i][j] = 0;
                dp_put[i][j] = false;
            }
        }
        for (i = n - 2; i >= 0; --i)
        {
            for (j = 0; j <= maximum_weight; ++j)
            {
                if (j >= items[i].w &&
                    dp_value[i + 1][j - items[i].w] + items[i].v > dp_value[i + 1][j])
                {
                    dp_value[i][j] = dp_value[i + 1][j - items[i].w] + items[i].v;
                    dp_put[i][j] = true;
                }
                else
                {
                    dp_value[i][j] = dp_value[i + 1][j];
                    dp_put[i][j] = false;
                }
            }
        }
        // output
        j = maximum_weight;
        for (i = 0; i < n; ++i)
        {
            if (dp_put[i][j])
            {
                output.push_back(i);
                j -= items[i].w;
            }
        }
        return output;
    }
```

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 ith 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

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

$\square$

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. $\square$

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;
```

```
        }
    }
    return output;
}
```

16.2-7

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

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

Proof. Suppose that $a_1 \le a_2 \le a_3 \le \cdots \le a_{n-1} \le a_n$ and $b_1 \le b_2 \le b_3 \le \cdots \le b_{n-1} \le b_n$. 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. □

16.3

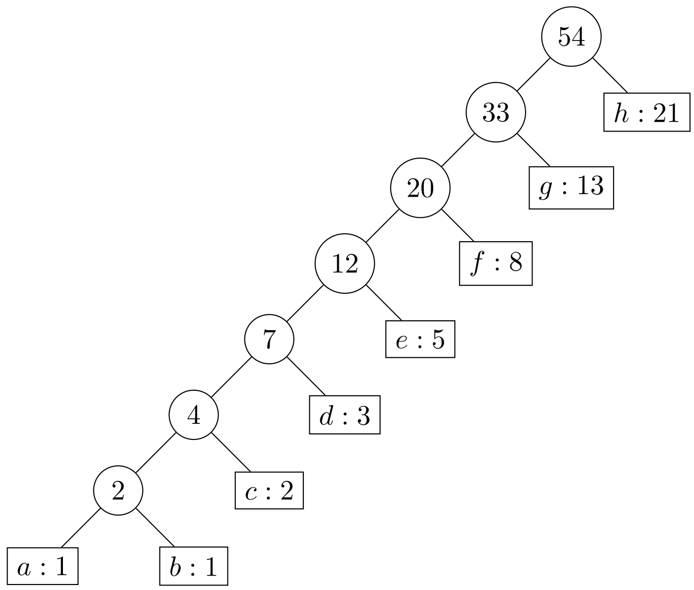
16.3-1

Since $x.freq$ and $y.freq$ are the two lowest leaf frequencies, and we assume that $a.freq \le b.freq$ and $x.freq \le y.freq$, we have $x.freq \le y.freq \le a.freq \le b.freq$. Thus, if $x.freq = b.freq$, then we must have $x.freq = y.freq = a.freq = b.freq$.

16.3-2

Proof. Suppose that a non-full binary tree T correspond to an optimal prefix code, for the purpose of contradiction. Then there exists a leaf node $a \in T$ such that a do not have sibling and, then, $a.freq = a.p.freq$. We construct T' by removing $a.p$ and put a at the postion of $a.p$. Then $d_{T'}(a) = d_T(a) - 1$. Hence $B(T) > B(T')$, which contradicts to that T correspond to an optimal prefix code. □

16.3-3



Lemma 7. For all $k \in \mathbb{N}$, $\sum_{i=1}^{k} F_i < F_{k+2}$.

Proof. *(Base)* Consider $k = 1$. We have $\sum_{i=1}^{1} F_i = F_1 = 1 < 2 = F_3 = F_{k+2}$.

(Induction) Fix $k \geq 1$. Suppose that $\sum_{i=1}^{k} F_i < F_{k+2}$. We want to show that $\sum_{i=1}^{k+1} F_i < F_{k+3}$.

$$\sum_{i=1}^{k+1} F_i = F_{k+1} + \sum_{i=1}^{k} F_i \overset{\text{IH}}{<} F_{k+1} + F_{k+2} = F_{k+3}$$

□

We construct the tree inductively. Suppose that T is a tree contains first n Fibonacci numbers and $T.root.freq = \sum_{i=1}^{n} F_i$. By the lemma, we have

$$T.root.freq = \sum_{i=1}^{n} F_i < F_{n+2} \quad .$$

Thus, to construct T' contains first $n+1$ Fibonacci numbers, we allocate a new node, let $T'.root$ be the new node, let $T'.root.left = T.root$, and let $T'.root.right$ be the node represents F_{n+1}. Then we have

$$T'.root.freq = T.root.freq + F_{n+1} = \sum_{i=1}^{n} F_i + F_{n+1} = \sum_{i=1}^{n+1} F_i \quad .$$

16.3-4

Denote T_x as the subtree rooted at node x. Denote C_x as the set of leaves (characters) in the subtree T_x. Denote M_x as the set of internal nodes (mergers) in the subtree T_x. Denote $h(x)$ as the height of node x.

Lemma 8. For all $x \in T$, we have

$$x.freq = \sum_{c \in C_x} c.freq \qquad (*) \quad .$$

Proof. *(Base)* Let $y \in T$ where $h(y) = 0$. Then y is a leaf, so $C_y = \{y\}$. We have $y.freq = \sum_{c \in C_y} c.freq = y.freq$.

(Induction) Suppose that $(*)$ is true for all $x \in T$ where $h(x) \leq k$. Let $y \in T$ where $h(y) = k+1$. We have

$$y.freq = y.left.freq + y.right.freq \overset{\text{IH}}{=} \sum_{c \in C_{y.left}} c.freq + \sum_{c \in C_{y.right}} c.freq = \sum_{c \in C_y} c.freq \quad .$$

□

Claim 9. For all $x \in T$, we have

$$B(T_x) = \sum_{m \in M_x} (m.left.freq + m.right.freq) \quad .$$

Proof. Note that for all $x \in T$ and for all $m \in M_x$, we have $m.left.freq + m.right.freq = m.freq$, so we want to show that

$$B(T_x) = \sum_{m \in M_x} m.freq \qquad (*) \quad .$$

(Base) Let $y \in T$ where $h(y) = 0$. Then y is a leaf, so $C_y = \{y\}$ and $M_y = \emptyset$. Thus,

$$LHS = B(T_y) = \sum_{c \in C_y} c.freq \cdot d_{T_y}(c) = y.freq \cdot d_{T_y}(y) = y.freq \cdot 0 = 0$$

and

$$RHS = \sum_{m \in M_y} m.freq = 0 \quad .$$

(Induction) Suppose that $(*)$ is true for all $x \in T$ where $h(x) \leq k$. Let $y \in T$ where $h(y) = k+1$. Then $h(y) \geq 1$, and y is an internal node, so $y \notin C_y$. We have

$$\begin{aligned}
B(T_y) &= \sum_{c \in C_y} c.freq \cdot d_{T_y}(c) \\
&= \sum_{c \in C_{y.left}} c.freq \cdot (d_{T_{y.left}}(c) + 1) + \sum_{c \in C_{y.right}} c.freq \cdot (d_{T_{y.right}}(c) + 1) \\
&= \sum_{c \in C_{y.left}} c.freq \cdot d_{T_{y.left}}(c) + \sum_{c \in C_{y.right}} c.freq \cdot d_{T_{y.right}}(c) + \sum_{c \in C_{y.left}} c.freq + \sum_{c \in C_{y.right}} c.freq \\
&= B(T_{y.left}) + B(T_{y.right}) + \sum_{c \in C_y} c.freq \\
&\overset{\text{IH}}{=} \sum_{m \in M_{y.left}} m.freq + \sum_{m \in M_{y.right}} m.freq + \sum_{c \in C_y} c.freq \qquad \text{(since } h(y.left) \leq k \text{ and } h(y.right) \leq k\text{)} \\
&\overset{\text{lemma}}{=} \sum_{m \in M_{y.left}} m.freq + \sum_{m \in M_{y.right}} m.freq + y.freq \\
&= \sum_{m \in M_y} m.freq \quad .
\end{aligned}$$

□

16.3-5

Lemma 10. For all w, x, y, z, if $x > w$ and $y > z$, then $xy + wz > xz + wy$.

Proof.

$$\begin{aligned} xy + wz > xz + wy &\iff y(x-w) + w(y+z) > z(x-w) + w(y+z) \\ &\iff y(x-w) > z(x-w) \\ &\iff x > w \wedge y > z \end{aligned}$$

□

Claim 11. Let $C = \{c_1, c_2, c_3, \cdots, c_{n-1}, c_n\}$ be the alphabet where

$$c_1.freq \geq c_2.freq \geq c_3.freq \geq \cdots \geq c_n.freq \quad .$$

Then there exists a tree T corresponding to an optimal code for C such that

$$d_T(c_1) \leq d_T(c_2) \leq d_T(c_3) \leq \cdots \leq d_T(c_{n-1}) \leq d_T(c_n) \quad .$$

Proof. Let T' be a tree corresponding to an optimal code for C. We construct T corresponding to an optimal code for C such that

$$d_T(c_1) \leq d_T(c_2) \leq d_T(c_3) \leq \cdots \leq d_T(c_{n-1}) \leq d_T(c_n)$$

inductively.

Case 1. There exists integers $i, j \in [1, n]$ where $i < j$ such that $c_i.freq = c_j.freq$ and $d_{T'}(c_i) > d_{T'}(c_j)$. Then we swap c_i and c_j, and we have $B(T')$ unchanged.

Case 2. There exists integers $i, j \in [1, n]$ where $i < j$ such that $c_i.freq > c_j.freq$ and $d_{T'}(c_i) > d_{T'}(c_j)$. Then we swap c_i and c_j, and, by the lemma, since

$$c_i.freq \cdot d_{T'}(c_i) + c_j.freq \cdot d_{T'}(c_j) > c_i.freq \cdot d_{T'}(c_j) + c_j.freq \cdot d_{T'}(c_i) \quad ,$$

we have $B(T')$ decreased. Hence this case is impossible to be trigged since the original T' corresponding to an optimal code for C.

We repeatedly modify T' by the above rules until

$$d_{T'}(c_1) \leq d_{T'}(c_2) \leq d_{T'}(c_3) \leq \cdots \leq d_{T'}(c_{n-1}) \leq d_{T'}(c_n) \quad .$$

□

16.3-6

By Exercise B.5-3, there are exactly $n-1$ internal nodes, so there are exactly $2n-1$ nodes in the full binary tree.

We use the first $2n-1$ bits to specify the structure of the tree by letting each bit indicate the corresponding node is a leaf (0) or an internal node (1). We imagine maintaining a queue. We push the two children of the current processing node into the queue if it is an internal node. Then we move on to the next node in the queue. We repeatedly do this process until the queue becomes empty.

We use the last $n\lceil \lg n \rceil$ bits to specify the characters on all leaves in the order of the leaves processed in the queue.

16.3-7

Claim 12. A full ternary tree with m internal nodes has exactly $2m+1$ leaves.

Proof. A full ternary tree with m internal nodes has $3m$ non-root nodes, so there are $3m+1$ nodes. Then there are $(3m+1)-m = 2m+1$ leaves. □

Let C an alphabet with n elements. If $n = 2m+1$ for some integer m, then we let the **for** loop repeats $m = \frac{n-1}{2}$ times. If $n \neq 2m+1$ for any integer m, then $n-1 = 2m+1$ for some integer m, and we do the following steps:

- dequeue first two nodes (having the lowest frequencies) to x and y,
- allocate a node and let x, y be its chilren (leave the third child empty),
- enqueue the node, so there are $n-1$ elements in the queue,
- let the **for** loop repeats $\frac{(n-1)-1}{2} = \frac{n}{2} - 1$ times.

```
struct Node
{
    Node *child_1;
    Node *child_2;
    Node *child_3;
    int freq;
    char character;
};

Node* Huffman(std::vector<Node*>&& alphabet)
{
    size_t n = alphabet.size(), m, i;
    auto cmp = [](Node *a, Node *b) { return a->freq > b->freq; };
    std::priority_queue<Node*, std::vector<Node*>, decltype(cmp)>
        q(cmp, std::forward<std::vector<Node*> >(alphabet));
    Node *root;
    if (n % 2 == 0)
    {
        root = new Node;
```

```
        root->child_1 = q.top();
        q.pop();
        root->child_2 = q.top();
        q.pop();
        root->child_3 = nullptr;
        root->freq = root->child_1->freq + root->child_2->freq;
        q.push(root);
        m = (n >> 1) - 1;
    }
    else
    {
        m = (n - 1) >> 1;
    }
    for (i = 0; i < m; ++i)
    {
        root = new Node;
        root->child_1 = q.top();
        q.pop();
        root->child_2 = q.top();
        q.pop();
        root->child_3 = q.top();
        q.pop();
        root->freq = root->child_1->freq + root->child_2->freq + root->child_3->freq;
        q.push(root);
    }
    return q.top();
}
```

We can use a similar approach from Lemma 16.2 and 16.3 to prove the correctness.

16.3-8

Let $C = \{0, 1, \cdots, n-1\}$ where $n = 256$. Let $C = \{c_1, c_2, c_3, \cdots, c_{n-1}, c_n\}$ where $\forall i, j \in [1, n], i \neq j \implies c_i \neq c_j$ and

$$c_1.freq \leq c_2.freq \leq c_3.freq \leq \cdots \leq c_{n-1}.freq \leq c_n.freq \quad .$$

Then $2 \cdot a_1.freq > a_n.freq$.

Lemma 13. For all integers $i, j, k \in [1, n]$, we have $a_i.freq + a_j.freq > a_k.freq$.

Proof. Suppose that $a_i.freq + a_j.freq \leq a_k.freq$, for the purpose of contradiction. Since

$$a_i.freq + a_j.freq \geq 2 \cdot a_i.freq \geq 2 \cdot a_1.freq$$

and

$$a_k.freq \le a_n.freq \quad ,$$

we have

$$2 \cdot a_1.freq \le a_n.freq \quad ,$$

which contradicts to $2 \cdot a_1.freq > a_n.freq$. □

Lemma 14. HUFFMAN(C) returns the root of an complete full binary tree where the height is 8.

Proof. Denote T as the result tree of HUFFMAN(C). Denote $h(x)$ as the height of node x in T. Denote T_x as the subtree rooted at x. Denote C_x as the set of leaves in the subtree T_x.

(Base) Before we enter the **for** loop, all of the following IH 1 - 6 hold.

(Induction) Let $Q = \langle q_1, q_2, \cdots, q_f \rangle$. Then $q_1.freq \le q_2.freq \le \cdots \le q_f.freq$. We suppose that the following inductively hypothesis (IH) is true for last iteration:

1. $\{C_q : q \in Q\}$ is a partition of C (i.e. $\bigcup\limits_{q \in Q} C_q = C$ and $\forall a, b \in Q, a \ne b \Longrightarrow C_a \cup C_b = \emptyset$);
2. $h(q_1) \le h(q_2) \le \cdots \le h(q_f)$;
3. $h(q_1) + 1 \ge h(q_f)$;
4. for all integers $i, j \in [1, f]$ where $i < j$, $h(q_i) = h(q_j) \Longrightarrow (\forall a \in C_{q_i}, b \in C_{q_j}, a.freq \le b.freq)$;
5. for all integers $i, j \in [1, f]$ where $i < j$, $h(q_i) < h(q_j) \Longrightarrow (\forall a \in C_{q_i}, b \in C_{q_j}, a.freq \ge b.freq)$;
6. for any $q \in Q$, T_q is a complete full binary tree.

We claim that $h(q_1) = h(q_2)$. Suppose $h(q_1) \ne h(q_2)$, for the purpose of contradiction. Then by IH 2 and 3, we have

$$h(q_1) + 1 = h(q_2) = h(q_3) = \cdots = h(q_{f-1}) = h(q_f) \quad .$$

Note that $|C_a| = 2^{h(a)}$. We have $|C_{q_1}| = 2^{h(q_1)}$ and

$$|C_{q_2}| = |C_{q_3}| = \cdots = |C_{q_{f-1}}| = |C_{q_f}| = 2^{h(q_2)} = 2^{h(q_1)+1} = 2 \cdot 2^{h(q_1)} = 2 \cdot |C_{q_1}| \quad .$$

Then

$$\begin{aligned}
|C_{q_1}| + |C_{q_2}| + |C_{q_3}| + \cdots + |C_{q_{f-1}}| + |C_{q_f}| &= |C_{q_1}| + (f-1) \cdot |C_{q_2}| \\
&= |C_{q_1}| + 2(f-1) \cdot |C_{q_1}| \\
&= (1 + 2(f-1)) \cdot |C_{q_1}| \\
&= (2f-1) \cdot |C_{q_1}| \\
&= (2f-1) \cdot 2^{h(q_1)} \quad .
\end{aligned}$$

By IH 1,

$$|C_{q_1}| + |C_{q_2}| + |C_{q_3}| + \cdots + |C_{q_{f-1}}| + |C_{q_f}| = |C| = 256 = 2^8$$

must be true. Since $(2f-1)$ is an odd integer, $(2f-1) \cdot 2^{h(q_1)}$ cannot a power of 2. Contradiction.

Now we allocate a new node z and let q_1, q_2 be z's children. Since $h(q_1) = h(q_2)$, we have $h(z) = h(q_1) + 1$. By IH 6, we know that z is a complete binary tree (IH 6 holds). We claim that $q_f.freq \leq z.freq$, so z might be enqueued to the end of Q. Note that $C_z = C_{q_1} \cup C_{q_2}$ (IH 1 holds). Since $h(z) = h(q_1) + 1$, by IH 3, we have $h(z) \geq h(g_f)$ (IH 2 holds). If $h(q_3) > h(q_1)$, then $h(q_3) = h(q_1) + 1 = h(z)$. If $h(q_3) = h(q_1)$, then $h(q_3) = h(q_1) = h(z) - 1$. Thus, $h(q_3) + 1 \geq h(z)$ (IH 3 holds).

Case 1. $h(q_f) < h(z)$.

Then $|C_z| \geq 2 \cdot |C_{q_f}|$ (actually, $|C_z| = 2 \cdot |C_{q_f}|$, but $\geq$ is enough). By Lemma 13, we have

$$\sum_{c \in C_z} c.freq > \sum_{c \in C_{q_f}} \quad .$$

By Lemma 8, we have $z.freq > q_f.freq$. Since $h(z) = h(q_1) + 1$, we have $h(q_f) < h(q_1) + 1$, so $h(q_f) = h(q_1)$, which means

$$h(q_1) = h(q_2) = \cdots = h(q_f) \quad .$$

By IH 4, we have

$$\forall a \in C_{q_1} \cup C_{q_2}, b \in C_{q_3}, a.freq \leq b.freq \quad .$$

Since $C_z = C_{q_1} \cup C_{q_2}$, we have

$$\forall a \in C_{q_3}, b \in C_z, a.freq \geq b.freq$$

(IH 5 holds).

Case 2. $h(q_f) = h(z)$.

Since $h(z) = h(q_1) + 1$, we have $h(q_1) < h(q_f)$. By IH 5, we have

$$\forall a \in C_{q_1}, b \in C_{q_f}, a.freq \geq b.freq \quad .$$

By IH 4, we have

$$\forall a \in C_{q_1}, b \in C_{q_2}, a.freq \leq b.freq \quad .$$

Thus,

$$\forall a \in C_z, b \in C_{q_f}, a.freq \geq b.freq$$

(IH 4 holds). We have

$$\sum_{c \in C_z} c.freq \geq \sum_{c \in C_{q_f}} \quad .$$

By Lemma 8, we have $z.freq \geq q_f.freq$.

Therefore, after we dequeue twice (dequeue q_1 and q_2) and enqueue z, all of IH 1 - 6 hold. $\square$

Claim 15. Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

Proof. By lemma 14, HUFFMAN(C) produced a Huffman coding which corresponds to an complete full binary tree where the height is 8, which is exactly the tree that corresponded by the ordinary 8-bit fixed-length code. $\square$

16.3-9

Let $C = \{0, 1, \cdots, n-1\}$ where $n = 256$. Let $C = \{c_1, c_2, c_3, \cdots, c_{n-1}, c_n\}$ where $\forall i, j \in [1, n], i \neq j \implies c_i \neq c_j$ and

$$c_1.freq \leq c_2.freq \leq c_3.freq \leq \cdots \leq c_{n-1}.freq \leq c_n.freq \quad .$$

In the expecting case, we have

$$c_1.freq + 1 \geq c_n.freq$$

Hence $2 \cdot a_1.freq > a_n.freq$. By Exercise 16.3-8, we have shown that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

Updating...