## *MCT*

```cpp
public:
    static auto mctTab(const std::vector<unsigned int>& _c) -> vtype {
        std::vector<std::vector<std::optional<vtype>>> cache(_c.size(), std::vector<std::optional<vtype>>(_c.size(), std::nullopt));

        for (sz it = 0; it != _c.size(); ++it)
            cache[it][it] = 0;

        for (sz i = _c.size() - 1; i >= 1; i--) {
            for (sz j = i + 1; j < _c.size(); j++) {
                sz min_cost = std::numeric_limits<vtype>::max();

                for (sz k = k = i; k < j; ++k) {
                    sz steps = _c[i - 1] * _c[k] * _c[j]
                        + cache[i][k].value() + cache[k + 1][j].value();

                    if (steps < min_cost) min_cost = steps;
                }

                cache[i][j] = min_cost;
            }
        }

        return cache[1][_c.size() - 1].value();
    }
```

## *NOccurrences*

```cpp
namespace my {
    template<typename FwdIt, typename Ty>
    FwdIt lower_bound(FwdIt _first, FwdIt _last, const Ty& _val) {
        return lower_bound(_first, _last, _val, std::less<>{});
    }

    template<typename FwdIt, typename Ty, typename Pred>
    FwdIt lower_bound(FwdIt _first, FwdIt _last, const Ty& _val, Pred _pred) {
        auto left = _first;
        auto right = _last ;
        auto mid = left;

        while (left <= right) {
            auto mid = left + std::distance(left, right) / 2;

            if (_val < mid)
                right = mid - 1;
            else if (mid < _val)
                mid = left + 1;
            else
                return mid;
        }
    }

    template<typename FwdIt, typename Ty>
    FwdIt upper_bound(FwdIt _first, FwdIt _last, const Ty& _val) {
        return upper_bound(_first, _last, _val, std::less<>{});
    }
```

```cpp
    template<typename FwdIt, typename Ty>
    FwdIt upper_bound(FwdIt _first, FwdIt _last, const Ty& _val) {
        return upper_bound(_first, _last, _val, std::less<>{});
    }

    template<typename FwdIt, typename Ty, typename Pred>
    FwdIt upper_bound(FwdIt _first, FwdIt _last, const Ty& _val, Pred _pred) {
        auto left = _first;
        auto right = _last ;
        auto mid = left;

        while (left <= right) {
            auto mid = left + std::distance(left, right) / 2;

            if (mid < _val)
                mid = left + 1;
            else if (_val < mid)
                right = mid - 1;
            else
                return mid;
            }
        }
    }

class NOccurences {
private:
    template<typename FwdIt, typename Ty>
    static std::size_t find(FwdIt _begin, FwdIt _end, Ty _element) {

        auto lft_it = my::lower_bound(_begin, _end, _element);
        auto rht_it = my::upper_bound(_begin, _end, _element);

        return std::distance(lft_it, rht_it);
    }

public:
    static void test() {
        std::vector<int> vec{ 1, 2, 3, 3, 3, 4, 5, 6, 7, 8, 9, 10 };
        //std::vector<int> vec{ 1, 2, 3, 3, 3, 4, 5 };

        auto occurences = find(vec.begin(), vec.end(), 3);

        std::cout << occurences << std::endl;
    }

};
```

---

*Bitonic*

---

```cpp
class BitonicSequence {
private:
    template<typename FwdIt>
    static auto find(FwdIt _first, FwdIt _last) -> FwdIt {
        using value_type = typename FwdIt::value_type;

        auto left = _first;
        auto right = _last - 1;
        auto mid = left;

        while (left <= right) {
            auto mid = left + std::distance(left, right) / 2;

            if (*(mid-1) < *mid && *mid > *(mid+1)) {
                return mid;
            }

            if (*mid < *(mid+1))
                left = mid + 1;
            else
                right = mid - 1;
        }

        return mid;
    }

public:
    static void test() {
        std::vector<int> bt_seq{ 1, 2, 3, 4, 5, 6, 7, 8, 3, 2, 1 };

        auto iter = find(bt_seq.begin(), bt_seq.end());

        std::cout << *iter << std::endl;
    }

};
```



---

*TwoSum*

---

```cpp
class TwoSum {
private:
    template<typename FwdIt, typename Ty>
    static std::optional<std::pair<FwdIt, FwdIt>> find(FwdIt _first, FwdIt _last, Ty _sum) {
        using value_type = typename FwdIt::value_type;
        std::unordered_map<value_type, FwdIt> hash;

        for (auto iter = _first; iter != _last; ++iter)
            hash.insert({ *iter, iter });

        for (auto iter = _first; iter != _last; ++iter) {
            Ty complement = _sum - *iter;

            auto found = hash.find(complement);
            if (found != hash.end())s
                return std::pair{ iter, found->second };
        }

        return {};
    }

public:
    static void test() {
        std::vector<int> vec{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        auto found = find(vec.begin(), vec.end(), 9);

        if (found.has_value())
        std::cout << *found->first << ", " << *found->second << std::endl;
    }

};
```

*KNeighbour*

```cpp
class KNearestNeighbour {
public:
    template<typename CType, typename Pred>
    static int partition(typename CType& arr, int l, int r, Pred _comp)
    {
        typename CType::value_type x = arr[r];
        int i = l;

        for (int j = l; j <= r - 1; ++j)
        {
            if (_comp(arr[j], x))
            {
                std::swap(arr[i], arr[j]);
                i++;
            }
        }
        std::swap(arr[i], arr[r]);
        return i;
    }

    template<typename CType, typename Pred>
    static void nth_element(CType& arr, int l, int r, unsigned int k, Pred _comp)
    {
        if (k > r - l + 1)
            throw std::invalid_argument("Invalid Arguments");

        int index = partition(arr, l, r, _comp);

        if (index - l == k - 1)
            return;

        if (index - l > k - 1)
            nth_element(arr, l, index - 1, k, _comp);
        else
            nth_element(arr, index + 1, r, k - index + l - 1, _comp);
    }
```

```cpp
static double handleMedian(std::vector<int>& _container) {

    auto begin = _container.begin();
    auto end = _container.end();
    const auto sample_size = std::distance(begin, end);

    if (sample_size % 2 != 0) {
        auto iter = begin + sample_size / 2;
        nth_element(_container, 0, _container.size() - 1, sample_size / 2, std::less<int>{});
        auto val = *iter;
        _container.erase(iter);
        return val;
    }
    else {
        //auto mid_m1 = begin + sample_size / 2;
        //auto mid_p1 = begin + (sample_size - 1) / 2;
        auto mid_m1 = sample_size / 2;
        auto mid_p1 = (sample_size - 1) / 2;
        nth_element(_container, 0, _container.size() - 1, sample_size / 2, std::less<int>{});
        nth_element(_container, 0, _container.size() - 1, (sample_size - 1) / 2, std::less<int>{});

        return static_cast<double>(_container[mid_m1] + _container[mid_p1]) / 2;
    }

}
```

```cpp
template<typename CType, typename BckIt>
static void kNearestNeighbour(CType _container, BckIt _back_inserter, const size_t _k) {

    double median = handleMedian(_container);

    std::vector<std::pair<double, int>> table;

    for (auto iter{ _container.begin() }; iter != _container.end(); ++iter) {
        table.emplace_back(std::pair{ std::abs(median - *iter), *iter });
    }


    nth_element(table, 0, table.size() - 1, _k,
        [](std::pair<double, int> lhs, std::pair<double, int> rhs) {
            return lhs.first < rhs.first;
        });

    for (auto iter = table.begin(); iter != table.begin() + _k * 2; ++iter) {
        *(_back_inserter++) = iter->second;
    }

}

static void test() {
    std::vector<int> arr{ 3, 5, 8, 9, 2, 1, 7, 4, 6 };

    std::vector<int> res;

    kNearestNeighbour(arr, std::back_inserter(res), 2);

    for (auto it : res)
        std::cout << it << ", ";

}
```