Lab 02. Divide And Conquer

The divide and conquer is a paradigm that breaks down a problem into two or more sub-problems of the same or related type, until trivial state (i.e., you know the answer directly, in most case, you may use a set of if-else constraints to get the answer).

When we design algorithm with divide and conquer paradigm, we need to solve the same type problems many times (original problem and sub-problems). Thus, when we implement function which using divide and conquer algorithm, the function usually recursivelly call itself to solve sub-problems. The psudo code may like followed:

```
#include <vector>

ReturnType solve(DataSet input) {
    // if the input is trivial, return the answer
    if (in_trivial_case_1()) {
        return ReturnType::V1;
    }
    if (in_trivial_case_2()) {
        return ReturnType::V2;
    }
    // if not, break down the problem to sub-problems
    std::vector<DataSet> subProblems = divide(input);
    // solve sub-problems recursively
    std::vector<ReturnType> subProblemReturns;
    for (size_t i = 0; i < subProblems.size(); ++i)
        subProblemReturns.push_back(solve(subProblems[i]));
    // aggregate the information from sub-problems to find the true answer
    ReturnType answer = merge(subProblemReturns);
    return answer;
}
```

Now, we can try to solve problem with divide and conquer paradigm. Suppose we want to design an algorithm which finding the largest element in an unorderd set. We may solve the problem as followed:

1. Break down the problem - divide given set to k non-overlapped partitions.
2. Solve the sub-problems - find the largest element of each partition.
3. Aggregate the result of sub-proboems - find the largest element from set of step 2.

We can implement the above algorithm as above pseudo code.

For speed, pass the parameter as slice, in other words, pass the legal range rather than copy whole subset. Response type depands on what we are finding, i.e., the type of given data.

```cpp
#include <vector>
typedef double DataType;
std::vector<DataType> given;
struct DataSet {
    size_t begin, count
};
```

Now, suppose we choose $k = 2$. We can implement the algorithm as followed:

```cpp
DataType solve(DataSet input) {
    // if the input is trivial, return the answer
    if (input.count == 1) // if we have only 1 element, then it's the biggest one
        return given[input.begin];
    // if not, break down the problem to sub-problems
    // divide problem to 2 sub-problems with the same size
    std::vector<DataSet> subProblems(2);
    // 1st part
    subProblems[0].begin = input.begin;
    subProblems[0].count = input.count / 2;
    // 2nd part
    subProblems[1].begin = input.begin + subProblems[0].count;
    subProblem[1].count = input.count - subProblem[0].count;
    // solve sub-problems recursively
    std::vector<DataType> subProblemReturns;
    for (size_t i = 0; i < subProblems.size(); ++i)
        subProblemReturns.push_back(solve(subProblems[i]));
    // aggregate the information from sub-problems to find the true answer
    DataType answer;
    if (subProblemReturns[0] > subProblemReturns[1])
        answer = subProblemReturns[0];
    else
        answer = subProblemReturns[1];
    return answer;
}
```

Question: Rewrite the above algorithm to find 3rd-largest element.

> Hint: your may try to find the 3 largest elements, instead of just finding the 3rd-largest one.

Master theorem

If we have a determinitic program works without recursive calls, it usually runs in polynomial time. And, in most case, your answer should be n^k, n shows the input size, and k is the maximum nested depth. But this not works when we have recursive calls.

Now, suppose we have a function which implement a divide and conuqer algorithm like above pseudo code. We can write our time complexity as $T(n) = g(n) + a \times T(\frac{n}{b}) + h(n)$. + $g(n)$ shows the time complexity of dividing problem to sub-problems. + A problem will be divided into a sub-problems. + Each sub-problem takes $\frac{n}{b}$ data. + $h(n)$ shows the time complexity of aggregating results.

Let $f(n) = g(n) + h(n)$, we will have

$$T(n) = a \times T(\frac{n}{b}) + f(n)$$

If we expand the equation, we'll get

$$T(n) = a^2 \times T(\frac{n}{b^2}) + f(n) + a \times f(\frac{n}{b})$$

$\log_b n$ rounds later, we have

$$T(n) = \sum_{i=0}^{\log_b n} a^i \times f(\frac{n}{b^i}) = \sum_{i=0}^{\log_b n} a^i \times f(b^{(\log_b n)-i})$$

$$= f(n) + a \times f(\frac{n}{b}) + \cdots + a^{(\log_b n)-1} \times f(b) + a^{\log_b n} \times f(1)$$

By above equation, we found the complexity is decided by recurrence cost $a^{\log_b n}$ and other cost $f(n)$. $a^{\log_b n} = n^{\log_b a}$. We named the exponent part $c_{crit} = \log_b a$ *critical exponent*. *Critical exponent* is very helpful for analysis the complexity.

- When $f(n) = O(n^c)$ for any $c < c_{crit}$, $T(n) = \Theta(n^{c_{crit}})$
- When $f(n) = \Omega(n^c)$ for any $c > cc_{crit}$, $T(n) = \Theta(f(n)) = \Omega(n^c)$
- Otherwise, $f(n) = \Theta(n^{c_{crit}}(\log n)^k), k >= 0$, $T(n) = \Theta(n^{c_{crit}}(\log n)^{k+1})$

Question: When satisfied otherwise, try to proof there are no $c > c_{crit}$ can let $f(n) = \Omega(n^c)$

Advantage of divide and conquer algorithms

Because sub-problems are indepenant to each others, we can solve the sub-problems simultaneously. Our true time usage will faster than $T(n) = aT(\frac{n}{b}) + f(n)$.

Assume that we have infinite computation power, we can solve all a sub-problems in the same time, so our time usage will near to $T(n) = T(\frac{n}{b}) + f(n)$.

Assume that we can run most k routines in the same time, that means we only need, our time usage will near to $T(n) = \lceil \frac{a}{k} \rceil T(\frac{n}{b}) + f(n)$

Map-reduce is a typical divide and conquer computation model hiwch was designed to processing big amount of data in Hadoop platform.

Typical divid and conquer exmaple: merge sort (in increasing order)

- Trivial state: If you have only one or no elements, then the array is sorted.
- Divid: split unsorted array to k parts, let their size as flat as possible.
- Aggregate: compare all head item of each sub-problem responses, extract the minimum one and append it to sorted array

Question: Try to analyse the time complexity of merge sort with $k = 5$ by master theorem

Question: Suppose given elements are not unique, please modify the algorithm to merge same values and count the frequent.

Input: [3, 2, 3, 5, 2, 3, 4, 5, 6, 22]

Output: [{2, 2}, {3, 3}, {4, 1}, {5, 2}, {6, 1}, {22,1}]