Lab 03. Divide And Conquer cont. - Binary Search

Sorting is a process that rearange a set of data in some meaningful order. e.g, in increasing order.

Definition: sorted in increasing order

> For a given list $A = \{a_1, a_2, \dots, a_n\}$, we say A is sorted in increasing order if $\forall i, 1 < i \leq n, a_{i-1} \leq a_i$

Sorted data may make our work musch easier than arbitary data. For a given list $A = \{a_1, a_2, \dots, a_n\}$, finding the maximum element in A at least cost $O(n)$ time. Since we need to check the value of each elements at least once to guarentee the answer, our time complexity always not better than $O(n)$. We may design a linear time element as below:

```
template <typename T>
T& max(const T data[], int n) {
    int ret = 0;
    for (int i = 1; i < n ; ++i) {
        if (data[ret] < data[i])
            ret = i;
    }
    return data[ret];
}
```

If our data is in increasing order, the only thing we need to do is just return the last element (cost $O(1)$).

```
template <typename T>
T& max(const T sorted_data[], int n) {
    return sorted_data[n-1];
}
```

But, consider about the preprocessing cost, sort maybe not a good idea. In above example, find the maximum number only cost $O(n)$, sort them cost $O(n\log n)$.

Question: Try to proof that sorting algorithms based on comparison never faster than $O(n\log n)$.

> Hint: For n elements, your data may have $P_n^n = n!$ different permutations. Each comparison have only 2 kinds of result, true or false.

Binary search

Let us do a more complex variation of above problem. For a given list $A = \{a_1, a_2, \dots, a_n\}$ and a key k, asking for finding the elements $a \in A$ that $a = k$.

In naive way, we can design an $O(n)$ program as below:

```
template <typename T>
int find(const T data[], int n, const T key) {
    for (int i = 0; i < n ; ++i) {
        if (data[i] == key)
            return i;
    }
    return n; // return terminator to represent not found
}
```

And again, if we working on a sorted list, we may have a much faster solution.

Suppose we choosing an arbitary element a_k from an incrasing order list A. We know that any element $a_i, i < k$ will not bigger than a_k, and any element $a_i, i > k$ will not less than a_k.

Thus, we can design a search progrm as followed:

```
template <typename T>

int find(const T sorted_data[], int n, const T key) {
    if (n <= 0)
        return 0;
    int k = n/2; // choose arbitary element a_k
    if (sorted_data[k] == key)
        return k;
    else if (sorted_data[k] > key) { // target i < k
        int tmp = find(sorted_data, k, key);
        return (tmp == k)? tmp: n; // if not found, return terminator
    } else { // target i > k => start from k+1
        int tmp = find(sorted_data + k + 1, n - k - 1, key);
        return tmp + k + 1;
    }
}
```

Above algorithm named **binary search** which always choose mid element as period and divide data to half and half. Binary search cost $O(\log_2 n)$ time, which is much faster than $O(n)$. If we consider the preprocessing cost ($O(n\log n)$ for sorting), binary search will beat naive algorithm when we query more than $\log n$ times on the same data set. Empirically, the larger the amount of data, the more often is is reused. That's why binary search is the first choice of most of database management system.

Question: Write down the recursive time function and analyze it by master theorem.

Question: Rewrite above algorithm by while loop.

Binary search can not only searching element, but also find solution from continuous function.

> Intermediate value theorem:
>
> For a continuous function $f: [a, b] \to \mathbb{R}$, any $u \in \mathbb{R}$ that $(f(a) - u)(f(b) - u) \leq 0$ exists $c \in [a, b]$ that $f(c) = u$.

We can solve unkonwn continuous function by intermediate value theorem and binary search as below:

```
#include <cmath>

double solve(double (*f)(double), double a, double b, double val, double epi)
{
    if ((f(a) - val) * (f(b) - val) > 0)
        return NAN;
    double mid = (a + b) * 0.5;
    if (abs(f(mid) - val) <= epi)
        return mid;
    else if ((f(a) - val) * (f(mid) - val) <= 0)
        return solve(f, a, mid, val, epi);
    else
        return solve(f, mid, b, val, epi);
}
```

Question: what's the time complexity of above algorihtm (in worst case)?

Hint 1: strongly related to precision (epi)

Hint 2: don't forget the time complexity of f

k-ary search

For traditional search problem, we can divide data to more than 2 partitions if we want.

Followed pseudo code shows how to work with k partitions

```
#include <vector>
template <typename T>

int find(const std::vector<T> &data, const T key, int k, int lb, int ub) {
    if (ub - lb < k) {
        for (int i = lb; data[i] <= key && i < ub; ++i)
            if (data[i] == key)
                return i;
        return data.size();
    }
    int periods[k - 1];
    for (int i = 1; i < k; ++i)
        periods[i-1] = lb * i + ub * (k-i) / k;
    if (key < data[periods[0]])
        return find(data, key, k, lb, periods[0]);
    if (key == data[periods[0]])
        return periods[0];
    for (int i = 1; i < k-1; ++i) {
        if (key < data[periods[i]])
            return find(data, key, k, periods[i-1]+1, periods[i]);
        if (key == data[periods[i]])
```

```
            return periods[i];
    }
    return find(data, key, k, periods[k-2]+1, ub);
}

int find(const std::vector<T> &data, const T key, int k) {
    return find(data, key, k, 0, data.size());
}
```

Question: What is the best k for searching key in n elements?