Algorithm Design Assignments 2
Dr Zamani Nit

Alireza Jafartash

983112029

Repository:

https://github.com/alirezaopmc/My-Toxic-Repo/tree/master/school-shits/da-term4/assignment2

```cpp
/*
Write an algorithm that prints out all the subsets of three elements
of n elements. The elements of this set are stored in a list that is
the input to the algorithm.
*/

#include <iostream>
#include <vector>

using namespace std;

void threeSubsets(vector<int> &arr);

int main() {
    vector<int> arr = {1, -4, 9, 6, 3};
    threeSubsets(arr);
}

void threeSubsets(vector<int> &arr) {
    int n = arr.size();

    for(int i = 0; i < n; i++) {
        for(int j = i+1; j < n; j++) {
            for(int k = j+1; k < n; k++) {
                printf("[%d, %d, %d]\n", arr[i], arr[j], arr[k]);
            }
        }
    }
}
```

```cpp
/*
Write an algorithm that finds both the smallest and the largest numbers
in a  list of n numbers. Try to find a method that does at most about
1.5 comparisions of array items.
*/

#include <iostream>
#include <vector>
#include <math.h>
// #include <pair>

using namespace std;

pair<int, int> findMinMax(vector<int> &arr, int l, int r);

int main() {
    vector<int> arr = {1, 4, 2, -1, 2, 12};
    auto min_max = findMinMax(arr, 0, 5);
    printf("Min = %d\n", min_max.first);
    printf("Max= %d\n", min_max.second);
}

pair<int, int> findMinMax(vector<int> &arr, int l, int r) {
    if (l == r) return {arr[l], arr[r]};

    if (l == r-1) return { min(arr[l], arr[r]), max(arr[l], arr[r]) };

    int mid = (l + r) / 2;

    auto left = findMinMax(arr, l, mid);
    auto right = findMinMax(arr, mid, r);

    return { min(left.first, right.first), max(left.second, right.second) };
}
```

```cpp
/*
Write an algorithm that determins whether or not an almost complete
binary tree us heap.
*/

class Node {
public:
    Node *left, *right;
    int value;
};

bool isHeap(Node *node);

int main() {
    // Only algorithm is available
}

bool isHeap(Node *node) {
    if (node == nullptr) return true;

    if (
        node->value < node->left->value ||
        node->value > node->right->value
    ) return false;

    return isHeap(node->left) && isHeap(node->right);
}
```

```
/*
Define basic operations for your algorithm in exercise
1-7, and study the performance of these algorithms. If
a given algorithm has an every-case time complexity,
determine it. Otherwise, determine the worst-case time
complexity.
*/

/*
???
*/
```

```cpp
/*
Write a linear-time algorithm that sorts n distinct
integers ranging from 1 to 500, inclusive.
*/

#include <iostream>
#include <vector>

using namespace std;

void linearSort(vector<int> &arr, int N);

int main() {
    int N = 500;
    vector<int> arr = {1, 500, 250, 500, 7, 1, 7, 1};

    linearSort(arr, N);

    for(int x : arr) {
        cout << x << " ";
    }
}

void linearSort(vector<int> &arr, int N) {
    vector<int> count(N, 0);
    for(int x : arr) {
        count[x-1]++;
    }

    int index = 0;
    for(int i = 0; i < N; i++) {
        for(int cnt = 0; cnt < count[i]; cnt++) {
            arr[index++] = i+1;
        }
    }
}
```

```
/*
There are two algorithms called Algl and Alg2
for a problem of size n. Alg 1 runs in n^2 microseconds
and A1g2 runs in 100n log n microseconds. Alg 1 can
be implemented using 4 hours of programmer time and
needs 2 minutes of CPU time. On the other hand, Alg2
requires 15 hours of programmer time and 6 minutes of
CPU time. If programmers are paid 20 dollars per hour
and CPU time costs 50 dollars per minute, how many
times must a problem instance of size 500 be solved
using Alg2 in order to justify its development cost?
*/

/*
Alg1:
    n^2 = 500^2 = 2500 microseconds
    4h development
    2 mins

    Income for one instance: 4 * 20 = 80

Alg2:
    100n log n = 100000 log 5 microseconds
    15h development
    6 mins

    Income for one instance: 15 * 20 = 300
    Then we have:
        300 = 50 * t => t = 6 minutes
        6 minutes = instances * 100k log 5
        => instances = 6 * 60 * 10^6 / 10^5 log 5
            = 3600 / log 5 ~ 5150
```

```
/*
Group the follwing functions by complexity category
*/

/*
linear:
8n+12

polynomial:
5n^2+7n, n^(5/2)

n lg(n):
l ln(n), lg(n!)

exponential:
4^n, e^n

square root:
sqrt(n)

n^n:
n^n, n^n + ln(N)

factorial:
n!

others:
5^lg(n), (lg n) ^ 2, 2^n!, lg(n)!
*/
```

```
/*
Discuss the reflexive, symmetric and transitive
properties for asymptotic comparisons.
O
BigO
Omega
Theta
o
*/
```

```
/*
Suppose you have a computer that requires 1 minute
to solve problem instances of size n = 1000. What
instance size can be run in 1 minute if you buy a
new computer that runs 1000 times faster than the
older one, assuming the following time complexities
T(n) for our algorithm?
(a) T(n) = Theta(n)
(b) T(n) = Theta(n^3)
(c) T(n) = Theta(10^n)
*/

/*
1000 times, size of input does not affect
*/
```

```
/*
Show the correctness of the following statements.
(a) lg n = O(n)
(b) n = O(n lg n)
(c) n lg n = O(n^2)
(d) 2^n = Omega(5^ln n)
(e) lg^3 n = o(n^0.5)


(a) lg n <= n
(b) n <= n lg n
(c) lg n <= n => n lg n <= n^2
(d) 5 < 8 = 2 ^ 3 => 5^ln n < 2^(3ln n)
    => n >= 3ln n => 2^n = Omega(5^ln n)
(e) ?
*/
```

```
/*
What is the time complexity T(n) of the nested loops
below?

for (i = 1; i <= n; i++) {
    j = n
    while(j >= 1) {
        ??? => Theta(1)

        j = [j / 2]
    }
}
*/

/*
T(n) = (1 + ln n) + (1 + ln n-1) + ... + (1 + ln 1)
    = n + ln n!
*/
```

```
/*
What is the time complexity T(N) of the nested loops
below?

i = n
while (i >= 1) {
    j = i
    while (j <= n) {
        ??? => Theta(1)

        j *= 2
    }

    i = [i / 2]
}
*/

/*
T(N) = 1 + 2 + ... + ln n = (ln n)(ln n + 1) / 2
*/
```

```cpp
/*
Give a Theta(n lg n) algorithm that computes the
remainder when x^n is divided by p.
*/

#include <iostream>
#include <vector>

using namespace std;

const int MAXN = 100000;

int moduleExponential(int x, int n, int p);

int main() {
    int x, n, p;

    cin >> x >> n >> p;
    cout << moduleExponential(x, n, p);
}

int moduleExponential(int x, int n, int p) {
    if (n == 0) return 1;
    if (n == 1) return (x >= p ? x % p : x);

    int k = moduleExponential(x, n/2, p);
    int r = (n & 1 ? x : 1);

    int result = k * k * r;
    return (result >= p ? result % p : result);
}
```

```
/*
Show that the function f(n) = |n^2 sin n|
is in neither O(n) nor Omega(n).
*/

/*
For enough large n, we have to prove
n^2 |sin n| > n <=> n |sin n| > 1
    <=> |sin n| > 1 / n
    RHS is almost equal to zero
    but LHS is surely more than RHS
    suppose it's smaller than 1 / n
    add one to the n then |sin n|
    will grow a bit.
*/
```

```
/*
Justify the correctness of the following
statements assuming that f(n) and g(n)
are asymptotically positive functions.

(a) f(n) + g(n) = O(max(f(n), g(n)))
(b) f^2(n) = Omega(f(n))
(c) f(n) + o(f(n)) = Theta(f(n)), where
    means any function g(n) = o(f(n))

(a) f(n) + g(n
    = min(f(n), g(n)) + max(f(n), g(n)))
    <= 2 * max(f(n), g(n))
    <=> min(f(n), g(n)) <= max(f(n), g(n))

(b) f(n) > 1 (positive)
    f^2(n) > f(n) <=> f(n) > 1

(c) ?
*/
```

```cpp
/*
Give an algorithm for the following problem.
Given a list of n distinct positive
integers, partition the list into two
sublists, each of size n/2, such that the
difference between the sums of the integers
in the two sublists is minimized. Determine
the time complexity of your algorithm.
You may assume that n is a multiple of 2.
*/

/*
Simple naive brute-force algorithm that checks
all subsets.
*/

/*
Time complexity O(ns) where:
    n: the number of elements
    s: sum of all elements
*/

#include <iostream>
#include <vector>
#include <math.h>

using namespace std;

void partitionWithMinDifference(vector<int> &arr, vector<bool> &record, int i, in
t &sum, int total);

int main() {
    vector<int> arr = {
        9, 2, 11, 12, 45, 1, 3
    };
    int n = arr.size();

    int total = 0;
    for(int x : arr) total += x;

    int sum = 0;
    vector<bool> partition(n+1);
    partitionWithMinDifference(arr, partition, 0, sum, total);

    for(int i = 0; i < n; i++) {
```

```
        if (partition[i]) {
            cout << arr[i] << " ";
        }
    }
    cout << endl;

    for(int i = 0; i < n; i++) {
        if (!partition[i]) {
            cout << arr[i] << " ";
        }
    }
    cout << endl;
}

void partitionWithMinDifference(vector<int> &arr, vector<bool> &record, int i, in
t &sum, int total) {
    int n = arr.size();

    if (i == n) {
        return;
    }

    vector<bool> with = record, without = record;
    with[i] = true;
    int withSum = sum + arr[i], withoutSum = sum;

    partitionWithMinDifference(arr, with, i+1, withSum, total);
    partitionWithMinDifference(arr, without, i+1, withoutSum, total);

    if (abs(total - 2 * withSum) < abs(total - 2 * withoutSum)) {
        record = with;
        sum += arr[i];
    } else {
        record = without;
    }
}
```

```
/*
Algorithm 1.7 (nth Fibonacci Term, Iterative) is clearly
linear in n, but is it a linear-time algorithm?
In Section 1.3.1 we defined the input size as the size
of the input. In the case of the nth Fibonacci term,
n is the input, and the number of bits it takes to
encode n could be used as the input size. Using this
measure the size of 64 is lg 64 = 6, and the size of
1024 is lg 1024 = 10. Show that Algorithm 1.7 is
exponential-time in terms of its input size. Show further
that any algorithm for computing the nth Fibonacci term
must be an exponential-time algorithm because the size
of the output is exponential in the input size. See
Section 9.2 for a related discussion of the input size.
*/

/*
Same as next problem
*/
```

```
/*
Determine the time complexity of Algorithm 1.6 (nth Fibonacci Term,
Recursive) in terms of its input size.
*/

/*
f(n) = f(n-1) + f(n-2)
T(n) = T(n-1) + T(n-2) + c

T(n-1) > T(n-2)
    => T(n) >= 2T(n-2) + c
    => T(n) >= (2^(n/2) - 1) * c
    &
    => T(n) <= 2T(n-1) + c
    => T(n) <= (2^n-1) * c
*/
```