Competitive Programming Week 5

Divide & conquer

COMPETITIVE
PROGRAMMING
CLUB

Membership sign up:

JUNIOR EXEC APPLICATIONS OPEN

Go to link below or scan QR code to apply.
The deadline is OCTOBER 31st at midnight.

https://forms.gle/8dkGw1Zezfm4BVnRA

SCAN ME

Alberta Collegiate Programming Contest (ACPC):
November 25th

More info to come

Week 4 Review

COMPETITIVE
PROGRAMMING
CLUB

Planting Trees

Problem: Given an array A where A[i] is how long tree i takes to grow, determine the earliest day where all trees will be fully grown if one tree can be planted per day

Greedy Heuristic: Plant the slowest growing remaining tree on the earliest remaining day

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool comp(int a, int b) { return a > b; }

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    int n;
    cin >> n;
    vector<int> v(n);

    for (int i = 0; i < n; i++) cin >> v[i];
    sort(v.begin(), v.end(), comp);

    int max = 0;
    for (int i = 0; i < n; i++) {
        int done = i + v[i];
        if (done > max) max = done;
    }

    cout << max + 2 << endl;

    return 0;
}
```

Hot Springs

Problem: Given a array A where A[i] represents the temperature of the ith hotspring, determine an arrangement of A where the difference of neighbouring elements is non-decreasing

Greedy Heuristic: Pick the coldest and hottest remaining hotsprings

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <stack>

using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    int n;
    cin >> n;
    vector<int> v(n);
    stack<int> res;
    for (int i = 0; i < n; i++) {
        int h;
        cin >> h;
        v[i] = h;
    }
    sort(v.begin(), v.end());

    for (int i = 0; i < n / 2; i++) {
        res.push(v[i]);
        res.push(v[n - 1 - i]);
    }
    if (n % 2 == 1) res.push(v[n / 2]);

    while (res.size() > 1) {
        cout << res.top() << ' ';
        res.pop();
    }
    cout << res.top() << endl;;

    return 0;
}
```

Interval Scheduling

Problem: Given a set of intervals determine how many can be scheduled without overlap

Greedy Heuristic: From the remaining intervals, pick the one with the earliest end time

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

typedef struct {
    long start;
    long end;
} Interval;

bool comp(Interval a, Interval b) {
    if (a.end == b.end) return a.start > b.start;
    return a.end < b.end;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    int n;
    cin >> n;

    vector<Interval> v(n);

    for (int i = 0; i < n; i++)
        cin >> v[i].start >> v[i].end;

    sort(v.begin(), v.end(), comp);

    int end = 0, c = 0;
    for (Interval i : v) {
        if (i.start >= end) {
            end = i.end;
            c++;
        }
    }

    cout << c << endl;

    return 0;
}
```

Birds

Problem: Given a length of cable, minimum distance between birds, and set of existing birds, determine how many additional birds could sit on the cable where birds cannot sit closer than 6 units from the ends.

Greedy Heuristic: Order the birds + the end points (offset to convert them to birds)

Post-processing step: Treat each pair of entries (i and i + 1) as an interval and determine how many birds could fit in it

```cpp
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    long l, d, n;
    cin >> l >> d >> n;

    vector<long> birds(n + 2);
    for (int i = 0; i < n; i++) cin >> birds[i];

    birds[n] = 6L - d;
    birds[n + 1] = l - 6L + d;

    sort(birds.begin(), birds.end());

    long c = 0;
    for (int i = 0; i < n + 1; i++)
        c += (birds[i + 1] - birds[i]) / d - 1L;

    cout << c << endl;

    return 0;
}
```

Divide and Conquer

Divide in Conquer is another approach to algorithm design focusing on taking a problem and **dividing** said problem into smaller, easier to solve subproblems (**conquering**).

Divide and conquer algorithms take advantage recursive structures in problems and the relationship between the depth of trees and the number of nodes in order to achieve logarithmic runtimes

Divide and Conquer

Generally there are 3 steps in a divide and conquer algorithm:
1. Divide the problem/input into smaller problems/inputs
2. Recursively solve the smaller problems
3. Merge the results into a solution to the original problem

This creates runtimes in the form of:

$$T(n) = \overbrace{\underbrace{a}_{\text{\# of subproblems}} \cdot T(n / \underbrace{b}_{\text{input division}})}^{\text{Conquer}} + \overbrace{\theta(f(n))}^{\text{Divide + Merge}}$$

The Master Theorem

$T(n) = a \cdot T(n / b) + \theta(f(n))$

The Master Theorem tells us that the open form of the runtime can be solved asymptotically by the following rules:

If $n^{log\_b(a)}$ is better asymptotically than f(n) (ie. $n^{log\_b(a)} \in o(f(n))$) then $T(n) \in O(f(n))$

If $n^{log\_b(a)}$ is asymptotically equivalent to f(n) (ie. $n^{log\_b}(a) \in \theta(f(n))$) then $\in T(n)\ O(f(n) \cdot \log(n))$

If $n^{log\_b(a)}$ is worse asymptotically than f(n) (ie. $n^{log\_b(a)} \in \omega(f(n))$) then $T(n) \in O(n^{log\_b(a)})$

Example

Problem:

Given an array of n integers A find the largest integer in A

Solution 1 (non-D&C):

```
def maxElement(A, n):
    int m := A[0]
    for i from 1 to n - 1:
        if A[i] > m then:
            m := A[i]
    return max
```

Time Complexity: O(n)

Solution 2 (D&C):

```
def maxElement(A, start, end):
    if start = end then:
        return A[0]
    int left := maxElement(A, start, (start + end / 2))
    int right := maxElement(A, (start + end) / 2,
    end)
    return max(left, right)
```

Time Complexity: $T(n) = 2T(n/2) + \theta(1) \in$ **O(n)**

Binary Search

Goal: Find the index of a value *val* in a sorted array *A* with length *n*

Algorithm:

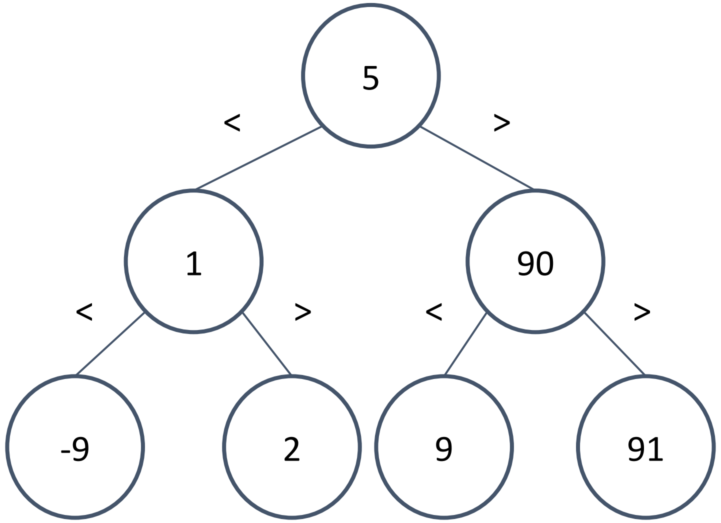
Recursive version:

```
def BinarySearch(A, start, end, val)
    if start = end
        if A[start] = val
            return start
        error
    if A[start + end / 2] > val
        return BinarySearch(A, start, (start + end) / 2, val)
    return BinarySearch(A, (start + end) / 2, end, val)
```

Iterative version:

```
def BinarySearch(A, n, val)
    min = 0
    max = n
    mid = (min + max) / 2
    while (A[mid] != val)
        if (A[mid] > val)
            max = mid
        else
            min = mid
        mid = (min + max) / 2
    return mid
```

-9	1	2	5	9	90	91



Merge Sort

Goal: Sort an array A

Insights:

1. An array of length 1 or 0 is already sorted
2. Merging 2 sorted arrays is "easy" (walk through the arrays comparing the current elements and put the smallest in another array)

1	7	8	10

2	3	4	9

Merge Sort

Goal: Sort an array A

Insights:

1. An array of length 1 or 0 is already sorted
2. Merging 2 sorted arrays is "easy" (walk through the arrays comparing the current elements and put the smallest in another array)

7	8	10

2	3	4	9

1							

Merge Sort

Goal: Sort an array A
Insights:

1. An array of length 1 or 0 is already sorted
2. Merging 2 sorted arrays is "easy" (walk through the arrays comparing the current elements and put the smallest in another array)

7	8	10

3	4	9

1	2						

Merge Sort

Goal: Sort an array A

Insights:

1. An array of length 1 or 0 is already sorted
2. Merging 2 sorted arrays is “easy” (walk through the arrays comparing the current elements and put the smallest in another array)

7	8	10

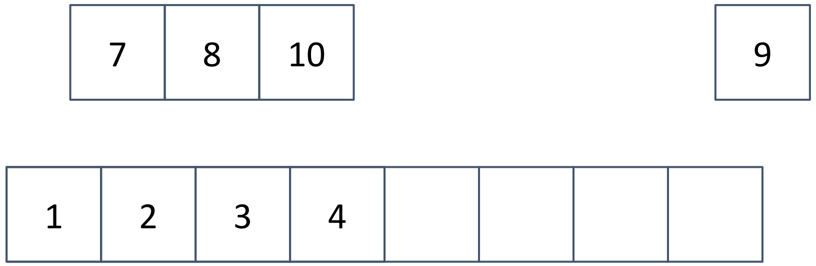
4	9

1	2	3					

Merge Sort

Goal: Sort an array A
Insights:

1. An array of length 1 or 0 is already sorted
2. Merging 2 sorted arrays is “easy” (walk through the arrays comparing the current elements and put the smallest in another array)

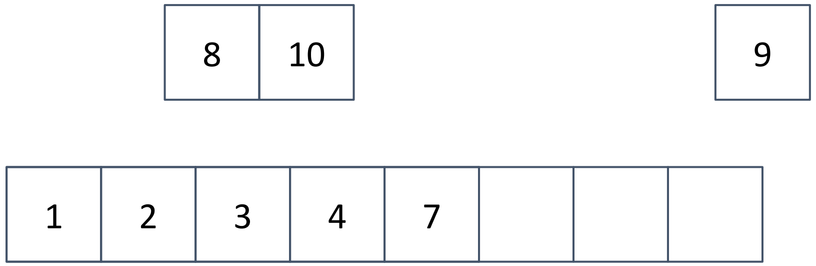


Merge Sort

Goal: Sort an array A

Insights:

1. An array of length 1 or 0 is already sorted
2. Merging 2 sorted arrays is "easy" (walk through the arrays comparing the current elements and put the smallest in another array)

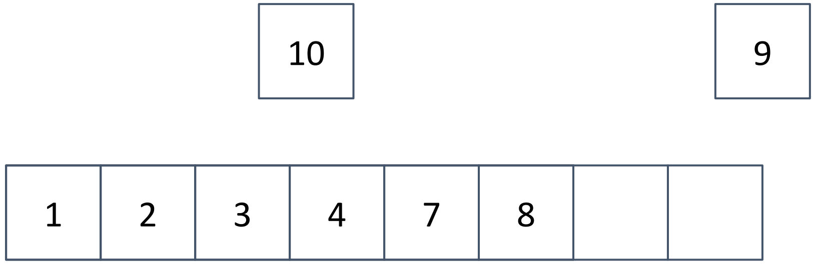


Merge Sort

Goal: Sort an array A

Insights:

1. An array of length 1 or 0 is already sorted
2. Merging 2 sorted arrays is "easy" (walk through the arrays comparing the current elements and put the smallest in another array)

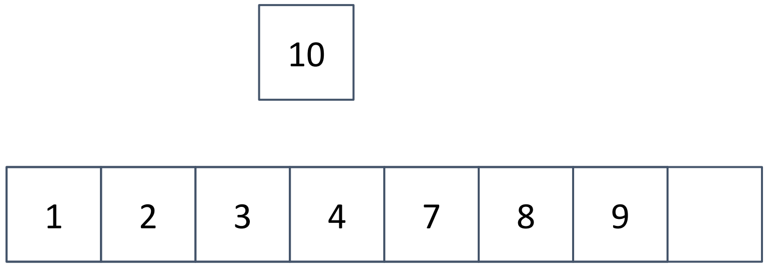


Merge Sort

Goal: Sort an array A

Insights:

1. An array of length 1 or 0 is already sorted
2. Merging 2 sorted arrays is “easy” (walk through the arrays comparing the current elements and put the smallest in another array)



Merge Sort

Goal: Sort an array A

Insights:

1. An array of length 1 or 0 is already sorted
2. Merging 2 sorted arrays is "easy" (walk through the arrays comparing the current elements and put the smallest in another array)

1	2	3	4	7	8	9	10

Merge Sort

Goal: Sort an array A

Insights:

1. An array of length 1 or 0 is already sorted
2. Merging 2 sorted arrays is "easy" (walk through the arrays comparing the current elements and put the smallest in another array)

Algorithm:

1. Split the array in half
2. Recursively sort the arrays
3. Merge them

$T(n) = 2T(n/2) + \theta(n)$

$T(n) \in O(n\log(n))$

Merge Sort

Goal: Sort an array A

Insights:

1. An array of length 1 or 0 is already sorted
2. Merging 2 sorted arrays is "easy" (walk through the arrays comparing the current elements and put the smallest in another array)

Algorithm:

1. Split the array in half
2. Recursively sort the arrays
3. Merge them

$T(n) = 2T(n/2) + \theta(n)$

$T(n) \in O(n\log(n))$

```
def MergeSort(A, start, end)
    if start = end:
        return [A[start]]
    L := MergeSort(A, start, (start + end / 2))
    R := MergeSort(A, (start + end / 2), end)
    Res := []
    l := 0
    r := 0
    for i from 0 to end - start:
        if L[l] < R[r]
            Res[i] := L[l]
            l++
        else
            Res[i] := R[r]
            r++
    return res
```

This Week's Contest:

https://open.kattis.com/contests/ggt8pr

(or look up "CPC Fall 2023 Practice Contest Week 5" in the Kattis contest list)

Feel free to ask questions until 7pm, and then throughout the week on Discord!

COMPETITIVE
PROGRAMMING
CLUB