CISS350: Data Structures and Advanced Algorithms
Assignment 3

Objectives:

1. View problems recursively
2. Write recursive functions

The skeleton code given for each question must be used. The trace printing in the code cannot be removed. (Just as in CISS245, the skeleton code is of course not complete and might have errors. The point is to give you something to begin your work.)

Q1. Write a recursive function that implements a power function recursively. You must use the following skeleton code which contains tracing of your function. The recursive power function below computes a^n where a has a `double` value `n` has an integer values that is ≥ 0. (You need not worry about the case when `n` is < 0.)

```
#include <iostream>

double pow_rec(double a, int n)
{
    if (n == -99999) // modify base condition
    {
        double ret = -99999; // set ret to correct value
        std::cout << "pow_rec(" << a << ", " << n << ") base case ... return "
                  << ret << '\n';
        return ret;
    }
    else
    {
        std::cout << "pow_rec(" << a << ", " << n << ") recursive case ...\n";
        double ret = -99999; // set ret to correct value
        std::cout << "pow_rec(" << a << ", " << n << ") recursive case ... "
                  << "return " << ret << '\n';
        return ret;
    }
}

int main()
{
    double a;
    int n;
    std::cin >> a >> n;
    double p = pow_rec(a, n);
    std::cout << p << '\n';
    return 0;
}
```

Hint: If a is a real number say 3.1 and n is a positive say 5. You want to think of 3.1^5 in terms of a smaller subproblem. For instance is there a relationship between 3.1^5 and 3.1^4?

Test 1

```
2 0
pow_rec(2, 0) base case ... return 1
1
```

Test 2

```
2 4
pow_rec(2, 4) recursive case ...
pow_rec(2, 3) recursive case ...
pow_rec(2, 2) recursive case ...
pow_rec(2, 1) recursive case ...
pow_rec(2, 0) base case ... return 1
pow_rec(2, 1) recursive case ... return 2
pow_rec(2, 2) recursive case ... return 4
pow_rec(2, 3) recursive case ... return 8
pow_rec(2, 4) recursive case ... return 16
16
```

Make sure you create your own test cases as well.

Q2. Write a recursive max function that implements the maximum value of an array. You must use the following skeleton code which contains tracing of your function. You may assume the array has at least one value (i.e., the size is ≥ 1.)

The function `max_rec(x, start, end)` computes the maximum value of `x[start]`, `x[start + 1]`, `x[start + 2]`, ..., `x[end - 1]`.

You must use a recursive relation between `max_rec(x, start, end)` and `max_rec(x, start, end - 1)`

```
#include <iostream>

int max_rec(int x[], int start, int end)
{
    if (start == end - 1)
    {
        int ret = -99999; // set ret to correct value
        std::cout << "max_rec(x, " << start << ", " << end
                  << ") base case ... "
                  << "return " << ret << '\n';
        return ret;
    }
    else
    {
        std::cout << "max_rec(x, " << start << ", " << end
                  << ") recursive case ...\n";

        int ret = -99999; // set ret to correct value





        std::cout << "max_rec(x, " << start << ", " << end
                  << ") recursive case ... "
                  << "return " << ret << '\n';
        return ret;
    }
}

int main()
{
    int n;
    std::cin >> n;
    int * x = new int[n];
    for (int i = 0; i < n; ++i)
    {
        std::cin >> x[i];
    }
    int max = max_rec(x, 0, n);
    std::cout << max << '\n';
    delete [] x;
    return 0;
}
```

Test 1

```
5
1 6 9 2 4
max_rec(x, 0, 5) recursive case ...
max_rec(x, 0, 4) recursive case ...
max_rec(x, 0, 3) recursive case ...
max_rec(x, 0, 2) recursive case ...
max_rec(x, 0, 1) base case ...return 1
max_rec(x, 0, 2) recursive case ...return 6
max_rec(x, 0, 3) recursive case ...return 9
max_rec(x, 0, 4) recursive case ...return 9
max_rec(x, 0, 5) recursive case ...return 9
9
```

(Clearly the inputs are the size of the array and the array.)

Test 2

```
5
9 4 1 2 6
max_rec(x, 0, 5) recursive case ...
max_rec(x, 0, 4) recursive case ...
max_rec(x, 0, 3) recursive case ...
max_rec(x, 0, 2) recursive case ...
max_rec(x, 0, 1) base case ...return 9
max_rec(x, 0, 2) recursive case ...return 9
max_rec(x, 0, 3) recursive case ...return 9
max_rec(x, 0, 4) recursive case ...return 9
max_rec(x, 0, 5) recursive case ...return 9
9
```

Test 4

```
1
10
max_rec(x, 0, 1) base case ...return 10
10
```

Make sure you create your own test cases as well.

Q3. Write a recursive linear search function that implements a linear search recursively. You must use the following skeleton code which contains tracing of your function. The recursive linear search function below returns the first index `i` where `x[i]` is `target` in the array `x`. Otherwise `-1` is returned.

You must use a recursive relation between `linearsearch_rec(x, start, end, target)` and `linearsearch_rec(x, start + 1, end, target)` when `n` is `start` is $<$ `end - 1`. Basically,
`linearsearch_rec(x, start, end, target)` performs a linear search on
`x[start]`, `x[start + 1]`, `x[start + 2]`, ..., `x[end - 1]`,

```
#include <iostream>

int linearsearch_rec(int x[], int start, int end, int target)
{
    if (start == end)
    {
        int ret = -99999; // set ret to correct value
        std::cout << "linearsearch_rec(x, " << start << ", " << end << ", "
                  << target << ") base case ... "
                  << "return " << ret << '\n';
        return ret;
    }
    else
    {
        std::cout << "linearsearch_rec(x, " << start << ", " << end << ", "
                  << target << ") recursive case ...\n";

        int ret = -99999; // set ret to correct value





        std::cout << "linearsearch_rec(x, " << start << ", " << end << ", "
                  << target << ") recursive case ... "
                  << "return " << ret << '\n';
        return ret;
    }
}

int main()
{
    int n, target;
    std::cin >> n;
    int * x = new int[n];
    for (int i = 0; i < n; ++i)
    {
        std::cin >> x[i];
    }
    std::cin >> target;
    int index = linearsearch_rec(x, 0, n, target);
    std::cout << index << '\n';
    delete [] x;
    return 0;
}
```

Test 1

```
5
1 3 2 5 4
2
linearsearch_rec(x, 0, 5, 2) recursive case ...
linearsearch_rec(x, 1, 5, 2) recursive case ...
linearsearch_rec(x, 2, 5, 2) recursive case ...
linearsearch_rec(x, 2, 5, 2) recursive case ...return 2
linearsearch_rec(x, 1, 5, 2) recursive case ...return 2
linearsearch_rec(x, 0, 5, 2) recursive case ...return 2
2
```

(Clearly the inputs are the size of the array, the array, and the target.)

Test 2

```
5
1 3 2 5 4
4
linearsearch_rec(x, 0, 5, 4) recursive case ...
linearsearch_rec(x, 1, 5, 4) recursive case ...
linearsearch_rec(x, 2, 5, 4) recursive case ...
linearsearch_rec(x, 3, 5, 4) recursive case ...
linearsearch_rec(x, 4, 5, 4) recursive case ...
linearsearch_rec(x, 4, 5, 4) recursive case ...return 4
linearsearch_rec(x, 3, 5, 4) recursive case ...return 4
linearsearch_rec(x, 2, 5, 4) recursive case ...return 4
linearsearch_rec(x, 1, 5, 4) recursive case ...return 4
linearsearch_rec(x, 0, 5, 4) recursive case ...return 4
4
```

Test 4

```
5
3 1 4 5 2
3
linearsearch_rec(x, 0, 5, 3) recursive case ...
linearsearch_rec(x, 0, 5, 3) recursive case ...return 0
0
```

Test 5

```
5
5 4 3 2 1
0
linearsearch_rec(x, 0, 5, 0) recursive case ...
linearsearch_rec(x, 1, 5, 0) recursive case ...
linearsearch_rec(x, 2, 5, 0) recursive case ...
linearsearch_rec(x, 3, 5, 0) recursive case ...
linearsearch_rec(x, 4, 5, 0) recursive case ...
linearsearch_rec(x, 5, 5, 0) base case ...return -1
linearsearch_rec(x, 4, 5, 0) recursive case ...return -1
linearsearch_rec(x, 3, 5, 0) recursive case ...return -1
linearsearch_rec(x, 2, 5, 0) recursive case ...return -1
linearsearch_rec(x, 1, 5, 0) recursive case ...return -1
linearsearch_rec(x, 0, 5, 0) recursive case ...return -1
-1
```

Make sure you create your own test cases as well.

Q4. Write a recursive bubblesort function that implements bubblesort recursively. Specifically, `bubblesort_rec(x, start, end)` performs bubblesort on
`x[start]`, `x[start + 1]`, `x[start + 2]`, ..., `x[end - 1]`,

You must use the following skeleton code which contains tracing of your function.

You must use a recursive relation between `bubblesort_rec(x, start, end)` and `bubblesort_rec(x, start, end - 1)`.

```
#include <iostream>
#include <string>

void print(int x[], int end)
{
    std::string delim = "";
    std::cout << '{';
    for (int i = 0; i < end; ++i)
    {
        std::cout << delim << x[i];
        delim = ", ";
    }
    std::cout << '}';
}


void bubblesort_rec(int x[], int start, int end)
{
    if (start >= end - 1)
    {
        std::cout << "bubblesort_rec(";
        print(x, end);
        std::cout << ", " << start << ", " << end
                  << ") base case ... return\n";
        return;
    }
    else
    {
        std::cout << "bubblesort_rec(";
        print(x, end);
        std::cout << ", " << start << ", " << end
                  << ") recursive case ...\n";
        std::cout << "bubblesort_rec(";
        print(x, end);
        std::cout << ", " << start << ", " << end
                  << ") recursive case ... return\n";
        return;
    }
}

int main()
{
    int n;
    std::cin >> n;
    int * x = new int[n];
    for (int i = 0; i < n; ++i)
    {
        std::cin >> x[i];
    }
    bubblesort_rec(x, 0, n);
    print(x, n);
    std::cout << '\n';
    delete [] x;
    return 0;
}
```

Test 1

```
5
5 1 4 2 3
bubblesort_rec({5, 1, 4, 2, 3}, 0, 5) recursive case ...
bubblesort_rec({1, 4, 2, 3}, 0, 4) recursive case ...
bubblesort_rec({1, 2, 3}, 0, 3) recursive case ...
bubblesort_rec({1, 2}, 0, 2) recursive case ...
bubblesort_rec({1}, 0, 1) base case ... return
bubblesort_rec({1, 2}, 0, 2) recursive case ... return
bubblesort_rec({1, 2, 3}, 0, 3) recursive case ... return
bubblesort_rec({1, 2, 3, 4}, 0, 4) recursive case ... return
bubblesort_rec({1, 2, 3, 4, 5}, 0, 5) recursive case ... return
{1, 2, 3, 4, 5}
```

(Clearly the inputs are the size of the array and the array.)

Test 2

```
1
5
bubblesort_rec(5, 0, 1) base case ... return
5
```

Make sure you create your own test cases as well.

Spoilers on next page.

Spoilers

The base case is when there are either no values or only one value to sort. There is one value to sort if `start` is `end - 1`. There are no values to sort if `start` > `end - 1`. Suppose recursively, `bubblesort_rec(x, start, end)` calls `bubblesort_rec(x, start, end - 1)`. In that case, on return from `bubblesort_rec(x, start, end - 1)`, the values `x[start]`, `x[start + 1]`, `x[start + 2]`, ..., `x[end - 2]` are sorted.

For instance support `start` is 0 and `end` is 5. `bubblesort_rec(x, 0, 5)` is supposed to sort `x[0]`, `x[1]`, `x[2]`, `x[3]`, `x[4]`. But since `bubblesort_rec(x, 0, 5)` calls `bubblesort_rec(x, 0, 4)` that means that `bubblesort_rec(x, 0, 4)` will sort `x[0]`, `x[1]`, `x[2]`, `x[3]`. What must `bubblesort_rec(x, 0, 5)` do to `x[0]`, `x[1]`, `x[2]`, ... `x[4]`. before calling `bubblesort_rec(x, 0, 4)`?

Divide-and-conquer

Make sure you review my notes on recursion (#28).

In all the previous recursions, there's a common theme:

1. Power computation: If I write `pow(x, n)` for x to the power of n, then the recursion is

   `pow(x, n)`: uses `pow(x, n - 1)` (and `x`)

2. Linear search: Let me write `x[0..n-1]` to denote the array of values `x[0],...,x[n-1]` (and ignore implementation specifics whether using array, `std::vector`, pointer to array, etc. If I write `linearsearch(x[0..n-1], target)` for the linear search of `target` in `x[0..n-1]`, then the recursion is

   `linearsearch(x[0..n-1], target)`: uses `linearsearch(x[1..n-1], target)` (and `x[0]`)

3. Bubblesort: If I write `bubblesort(x[0..n-1])` for the bubblesort of `x[0..n-1]`, then the recursion is

   `bubblesort of x[0..n-1]`: uses `bubblesort(x[0..n-2])`

If n denotes the size of a problem (in the case of the power computation it's the exponent and in the case of array type problems it's the size of the array), then in all the above the philosophy of recursion is this: If $P(n)$ is a problem, then to solve $P(n)$ by recursion you might want to try

to solve $P(n)$: solve $P(n-1)$ (as well as some other computations)

For instance

to compute x^n: compute x^{n-1} (and multiply it with x)

(Such recursions are called linear recursions of degree 1.)

A divide-and-conquer (DAC) recursion is "usually" different. The philosophy of DAC is like this:

to solve $P(n)$: solve one or two $P(n/2)$ problems (and do some computations)

The $n/2$ is not exactly $n/2$, but an integer value roughly $n/2$.

Note however that some people consider *both* methods,

- $P(n)$ in terms of $P(n-1)$ and
- $P(n)$ in terms of $P(n/2)$

as DAC. In other words for some people, as long as a problem is solved in terms of "smaller" problem(s), it's a DAC solution, whether it is smaller by 1 or smaller by a factor of 1/2.