CSC 212: Data Structures and Abstractions

12: Recursion

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

THINK BIG WE DO℠

Notes

- Assignment 4
  - ✓ increase total points to 120
  - ✓ solving less problems can still achieve 100 points
- Instructor office hours
  - ✓ help on assignment 4 (Wed 26th, 4-5p)
- Midterm 2
  - ✓ new date (April 3rd)

Recursion

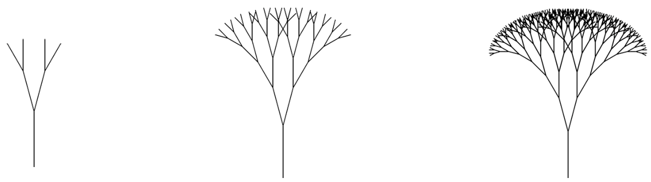
- Definition
  - ✓ method of solving problems that involves breaking a problem down into smaller and smaller subproblems (of the same structure) until you get to a small enough problem that it can be solved trivially
- Recursive functions
  - ✓ technically, a recursive function is one that calls itself
  - ✓ must have at least a base case and a recursive case
  - ✓ **base case**: a condition that will eventually be met that will stop the recursion
  - ✓ **recursive case**: a condition that will eventually be met that will continue the recursion

Basic form

```
function() {

    if (base case) {

        return trivial solution

    } else {

        break task into subtasks
        solve each task recursively
        merge solutions if necessary
        return solution

    }
}
```

Why recursion?

- Can we live without it?
  - ✓ yes, for every recursive function, there is an iterative solution
- However …
  - ✓ some formulas are explicitly recursive
  - ✓ some problems exhibit a natural recursive solution



https://courses.cs.washington.edu/courses/cse120/17sp/labs/11/tree.html

Practice

- Write a recursive function to add all elements in a vector

```cpp
int sum_array(std::vector<int>& A, int n) {
    // base case
    if (n == 1) {
        return A[0];
    }

    // solve sub-task
    int partial_sum = sum_array(A, n-1);

    // return sum
    return A[n-1] + partial_sum;
}
```


Recursion call tree

- Definition
  - ✓ a tree that represents the recursive calls of a function
- Properties
  - ✓ each node in the tree represents a call to the function
  - ✓ the root of the tree represents the initial call
  - ✓ the children of a node represent the recursive calls made by that function call
  - ✓ the leaves of the tree represent the base cases
  - ✓ the height of the tree represents the depth of the recursion
  - ✓ the number of nodes in the tree represents the total number of recursive calls made

Draw the recursion call tree

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

int sum_array(std::vector<int>& A, int n) {
    // base case
    if (n == 1) {
        return A[0];
    }

    // solve sub-task
    int partial_sum = sum_array(A, n-1);

    // return sum
    return A[n-1] + partial_sum;
}

int main() {
    std::vector<int> A = {1, 2, 3, 4, 5};
    int sum = sum_array(A, A.size());
    std::cout << "Sum of array: " << sum << std::endl;
    return 0;
}
```

Draw the recursion call tree

$$b^n = \underbrace{b \cdot b \cdot \ldots \cdot b}_{n \text{ times}}$$

```
double power(double b, int n) {
    // base case
    if (n == 0) {
        return 1;
    }
    // recursive call
    return b * power(b, n-1);
}

int main() {
    std::cout << "3^8: " << power(3, 8) << std::endl;
    return 0;
}
```


Faster?

- Draw the recursion call tree

$$b^n = \underbrace{b \cdot \ldots \cdot b}_{n/2 \text{ times}} \cdot \underbrace{b \cdot \ldots \cdot b}_{\sim n/2 \text{ times}}$$

```
double power_2(double b, int n) {
    if (n == 0) {
        return 1;
    }
    double half = power(b, n/2);
    if (n % 2 == 0) {
        return half * half;
    } else {
        return b * half * half;
    }
}

int main() {
    std::cout << "3^8: " << power_2(3, 8) << std::endl;
    return 0;
}
```


Binary search

Binary search

- Search on a sorted sequence
  - find the position of a target value within a **sorted array**
  - binary search is an efficient algorithm for this problem
- Binary search algorithm
  - compare the target value to the middle element of the array
  - if they are not equal
    - the half in which the target cannot lie is eliminated and the search continues on the remaining half
  - if the target value is equal to the middle element
    - the search terminates successfully
- Recursive approach
  - base case: the array is empty
  - recursive case: the array is not empty, apply recursion to the left or right half of the array

Binary search

0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	5	10	15	20	22	30	35	40	43	48	51

`lo` `hi`

k = 48?

Binary search

0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	5	10	15	20	22	30	35	40	43	48	51

`lo` `mid` `hi`

k = 48?

Binary search

0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	5	10	15	20	22	30	35	40	43	48	51

`lo` `hi`

k = 48?

Binary search

0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	5	10	15	20	22	30	35	40	43	48	51

`lo` `mid` `hi`

k = 48?

Binary search

0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	5	10	15	20	22	30	35	40	43	48	51

lo mid hi

k = 48?

Show me the code

```
int bsearch(std::vector<int>& A, int lo, int hi, int k) {
    // base case
    if (hi < lo) {
        return -1;
    }
    // calculate midpoint index
    int mid = lo + ((hi-lo)/2);
    // key found?
    if (A[mid] == k)
        return mid;
    // key in upper subarray?
    if (A[mid] < k)
        return bsearch(A, mid+1, hi, k);
    // key is in lower subarray?
    return bsearch(A, lo, mid-1, k);
}

int main() {
    std::vector<int> A = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    std::cout << bsearch(A, 0, A.size()-1, 9) << std::endl;
    return 0;
}
```


Draw the recursion call tree

- What is the complexity?
  - ✓ best case? worst case? average case?

```
int bsearch(std::vector<int>& A, int lo, int hi, int k) {
    // base case
    if (hi < lo) {
        return -1;
    }
    // calculate midpoint index
    int mid = lo + ((hi-lo)/2);
    // key found?
    if (A[mid] == k)
        return mid;
    // key in upper subarray?
    if (A[mid] < k)
        return bsearch(A, mid+1, hi, k);
    // key is in lower subarray?
    return bsearch(A, lo, mid-1, k);
}

int main() {
    std::vector<int> A = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    std::cout << bsearch(A, 0, A.size()-1, 9) << std::endl;
    return 0;
}
```


Linear vs logarithmic time

n	time (days)	~ log(n)
1	0.000	0
10	0.000	3
100	0.000	7
1000	0.000	10
10000	0.000	13
100000	0.000	17
1000000	0.000	20
10000000	0.000	23
100000000	0.000	27
1000000000	0.000	30
10000000000	0.000	33
100000000000	0.000	37
1000000000000	0.000	40
10000000000000	0.000	43
100000000000000	0.003	47
1000000000000000	0.028	50
10000000000000000	0.281	53
100000000000000000	2.809	56
1000000000000000000	28.086	60
10000000000000000000	280.863	63
100000000000000000000	2808.628	66

Intel Core i9-9900K	412,090 MIPS at 4.7 GHz