CSC 212: Data Structures and Abstractions

12: Recursion

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

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

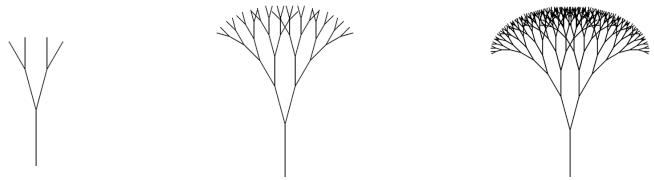
- Definition
  - method of solving problems that involves breaking a problem down into smaller and smaller subproblems (of the same structure) until you get to a small enough problem that can be solved trivially
- Recursive functions
  - technically, a recursive function is one that calls itself
  - 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

```
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 are equivalent to the total number of recursive calls made

Draw the recursion call tree

```
#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

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

```cpp
#include <iostream>

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;
}
```


Linear vs logarithmic time

n	time (# days)	~ log(n) (# operations)
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

How much faster?

- Draw the recursion call tree
  - what is the computational complexity?

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

```cpp
#include <iostream>

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

int main() {
    std::cout << "3^8: " << power_2(3, 16) << std::endl;
    return 0;
}
```


Draw the recursion call tree

```cpp
#include <cinttypes>
#include <iostream>

// fibonacci sequence (recursive)
// 0 1 1 2 3 5 8 13 21 34 55 89 144 ...
uint64_t fibR(uint16_t n) {
    if (n < 2) {
        return n;
    } else {
        return fibR(n-1) + fibR(n-2);
    }
}

int main() {
    std::cout << fibR(100) << std::endl;
    return 0;
}
```

Practice

- Write a recursive function to search for an element on a singly linked list (return false/true)
  - the function gets a pointer to an arbitrary node in the list
- Modify the function to return the index of the element relative to the initial node p

Binary search

Binary search

- Search on a sorted sequence
  - binary search is an **efficient** algorithm for locating a **target value** within a **sorted array**
  - the ordered nature of the data is exploited to achieve logarithmic time complexity
- Algorithm (using recursion)
  - compare the target value to the middle element of the sorted array
  - if the target equals the middle element, the search terminates successfully
  - if the target differs from the middle element:
    - eliminate the half of the array in which the target cannot reside
    - continue recursively on the remaining half
- Recursive approach
  - **base case**: the array is empty, target not present
  - **recursive case**: the array is not empty, target found or recursion is applied

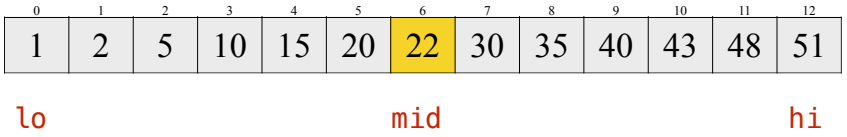
Binary search

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`

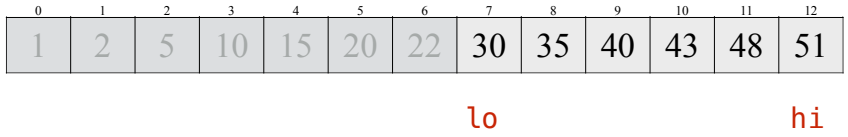
k = 48?

Binary search



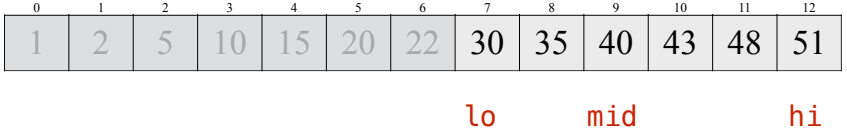
k = 48?

Binary search



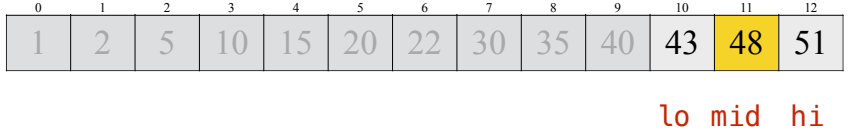
k = 48?

Binary search



k = 48?

Binary search



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;
}
```


Logarithmic complexity

- Mathematical basis for $O(\log n)$ complexity
  - cost corresponds to the number of times n must be divided by 2 until reaching 1
  - each division by 2, represents a reduction of the problem space by half
  - this process can be mathematically formulated as:

$$\frac{n}{2^k} \leq 1$$

  - solving for k:

$$k \geq \log_2 n$$

- Example
  - consider $n = 16$, the number of times 16 must be divided by 2 until reaching 1 is?

$$\frac{16}{2^1} \rightarrow \frac{16}{2^2} \rightarrow \frac{16}{2^3} \rightarrow \frac{16}{2^4}$$