CSC 212: Data Structures and Abstractions

13: Recursion

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2025

THINK BIG WE DO℠

Recursion call tree

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

Practice

- Draw the recursion call tree
  - express computational cost as the total number of additions

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


Practice

- Draw the recursion call tree
  - express computational cost as the total number of multiplications

```
double power(double b, int n) {
    if (n == 0) {
        return 1;
    }
   return b * power(b, n-1);
}

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

Practice

- Draw the recursion call tree
  - ✓ express computational cost as the total number of multiplications

```
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;
}
```


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

Draw the recursion call tree

```
// 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;
}
```


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)
  - if the array is empty, the target is not present
  - 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

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, 5) << 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) {
    if (hi < lo) {
        return -1;
    }
    int mid = lo + ((hi-lo)/2);
    if (A[mid] == k) return mid;
    if (A[mid] < k) return bsearch(A, mid+1, hi, k);
    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
  - ✓ the time complexity is proportional 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:

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

where k is the number of divisions required to reduce the problem size to 1 or less

  - ✓ 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?

$$\frac{16}{2} = 8 \rightarrow \frac{8}{2} = 4 \rightarrow \frac{4}{2} = 2 \rightarrow \frac{2}{2} = 1$$