CSC 212: Data Structures and Abstractions

13: Recursion

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2025

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

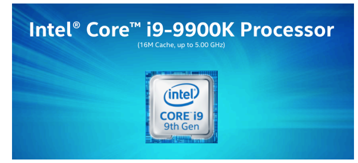
- Draw the recursion call tree
  - ✓ express computational cost as the total number of multiplications

```
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, 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

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:

$$n \leq 2^k$$
$$\log_2 n \leq \log_2 2^k$$
$$\log_2 n \leq k$$

the smallest integer k satisfying this is $\lceil \log_2 n \rceil$

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

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

Recurrence relations

Recurrences

- Recurrence relation (a.k.a. recurrence)
  - equation that expresses the terms of a sequence (beyond the initial conditions) as a function of one or more preceding terms
  - bases cases must be specified to uniquely define the sequence
  - e.g., $T(n) = T(n-1) + 1$
- Recurrences and algorithm analysis
  - recurrences are used to analyze the time complexity of recursive algorithms
    - the time complexity $T(n)$ of a recursive algorithm can be expressed by a recurrence relation
    - solving a recurrence means finding a close-form formula for $T(n)$
    - an exact closed-form solution may not exist or may be difficult to derive
  - for most recurrences, an asymptotic solution of the form $\Theta\left(f(n)\right)$ is typically sufficient

Techniques for solving recurrences

- **recursion tree**
  - draw a recursion tree and sum the costs at each level (not a formal proof)
- **substitution**
  - guess the form of the solution and prove it works by induction
- **master theorem**
  - a shortcut for solving recurrences of the form $T(n) = aT(n/b) + f(n)$
- **iteration**
  - iteratively expand (unroll) the recurrence until a pattern emerges, then express the general case and apply the base case
  - not trivial in all cases but it is helpful to build an intuition
  - induction may be necessary to prove correctness

Practice

$$T(0) = 0$$

$$T(n) = 1 + T(n-1)$$

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


Binary search

Binary search

- Search algorithm for sorted sequences
  - **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 (recursive formulation)
  - **base case:** if the input sequence is empty, the target is not present, return false
  - compare the target value to the middle element of the input sequence
  - if the target equals the middle element, return true and/or the index of the element found
  - **recursive cases:** if the target differs from the middle element:
    - if target value is less than the middle element, recursively search the left half
    - if target value is greater than the middle element, recursively search the right 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

```
bool bsearch(std::vector<int>& A, size_t lo, size_t hi, int k) {
    if (hi < lo) {
        return false;
    }

    size_t mid = lo + ((hi-lo)/2); // safer than (lo+hi) / 2

    if (A[mid] == k) {
        return true;
    } else if (A[mid] < k) {
        return bsearch(A, mid+1, hi, k);
    } else {
        if (mid == 0) return false; // prevent size_t wraparound
        return bsearch(A, lo, mid-1, k);
    }
}
```


Draw the recursion call tree

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

```
bool bsearch(std::vector<int>& A, size_t lo, size_t hi, int k) {
    if (hi < lo) {
        return false;
    }
    size_t mid = lo + ((hi-lo)/2); // safer than (lo+hi) / 2
    if (A[mid] == k) {
        return true;
    } else if (A[mid] < k) {
        return bsearch(A, mid+1, hi, k);
    } else {
        if (mid == 0) return false; // prevent size_t wraparound
        return bsearch(A, lo, mid-1, k);
    }
}
```

Analysis of binary search

$$T(0) = 0$$

$$T(n) = c + T(n/2)$$

```
bool bsearch(std::vector<int>& A, size_t lo, size_t hi, int k) {
    if (hi < lo) {
        return false;
    }
    size_t mid = lo + ((hi-lo)/2); // safer than (lo+hi) / 2
    if (A[mid] == k) {
        return true;
    } else if (A[mid] < k) {
        return bsearch(A, mid+1, hi, k);
    } else {
        if (mid == 0) return false; // prevent size_t wraparound
        return bsearch(A, lo, mid-1, k);
    }
}
```

count the number of comparisons

MergeSort

Sorting

- Given a sequence of n elements that can be compared according to a total order relation
  - ✓ we want to rearrange them in monotonic order (non-decreasing or non-increasing)
- Formally, the output of any sorting algorithm must satisfy two conditions:
  - ✓ the output is in monotonic order (each element is no smaller/larger than the previous element, according to the required order)
  - ✓ the output is a permutation (a reordering that retains all of the original elements) of the input

`central problem in computer science`

Insertion sort

- Algorithm (non-decreasing order)
  - ✓ start at index 1, loop through the array
  - ✓ for each element
    - compare with the element to its left
    - if smaller, swap them and move left
    - repeat until element is not smaller or you reach the start

Time complexity depends on the input

- Worst-case: $\Theta(n^2)$
  - input is reverse sorted
- Best-case? $\Theta(n)$
  - input is already or almost sorted
- Average-case? $\Theta(n^2)$
  - expect every element to move $O(n/2)$ times on average

```
void insertion_sort(std::vector<int>& A) {
    for (size_t i = 1 ; i < A.size() ; ++i) {
        for (size_t j = i; j > 0 ; --j) {
            if (A[j] < A[j-1]) {
                std::swap(A[j], A[j-1]);
            } else {
                break;
            }
        }
    }
}
```

Merge sort

- **Divide** the array into two halves
  - ✓ calculate the midpoint to split the array
- **Conquer** each half recursively
  - ✓ call merge sort on each half (solve 2 smaller problems)
- **Combine** the solutions
  - ✓ after both recursive calls finish, **merge** the two sorted halves into one sorted array

Divide and Conquer Methods
A problem-solving approach that breaks a problem into smaller subproblems, solves them independently, and then combines their solutions

Examples: binary search, merge sort, quick sort

Merge sort

0	1	2	3	4	5	6	7
4	2	1	10	6	20	7	30

Merge sort algorithm

```
if (hi <= lo) return

mid = lo + (hi - lo) / 2

mergesort(A, lo, mid)
mergesort(A, mid+1, hi)

merge(A, lo, mid, hi)
```

```
void merge_sort(std::vector<int>& A, size_t lo, size_t hi) {
    if (hi <= lo) return;
    size_t mid = lo + ((hi-lo) / 2);
    merge_sort(A, lo, mid);
    merge_sort(A, mid+1, hi);
    merge(A, lo, mid, hi);
}
```


Merging two sorted arrays

1	2	4	10

i →

6	7	20	30

j →

A temporary array is necessary to guarantee a linear time operation

Show me the code

```
void merge(std::vector<int>& A, size_t lo, size_t mid, size_t hi) {
    std::vector<int> B(hi - lo + 1);
    size_t i = lo, j = mid + 1, k = 0;

    while (i <= mid && j <= hi) {
        if (A[i] <= A[j]) {
            B[k++] = A[i++];
        } else {
            B[k++] = A[j++];
        }
    }
    while (i <= mid) B[k++] = A[i++];
    while (j <= hi) B[k++] = A[j++];

    for (k = 0 ; k < B.size() ; ++k) {
        A[lo + k] = B[k];
    }
}
```


Draw the recursion call tree

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

```
void merge_sort(std::vector<int>& A, size_t lo, size_t hi) {
    if (hi <= lo) return;
    size_t mid = lo + ((hi-lo) / 2);
    merge_sort(A, lo, mid);
    merge_sort(A, mid+1, hi);
    merge(A, lo, mid, hi);
}
```


Analysis of merge sort

$$T(0) = 0$$
$$T(1) = 0$$
$$T(n) = 2T(n/2) + \Theta(n)$$

```
void merge_sort(std::vector<int>& A, size_t lo, size_t hi) {
    if (hi <= lo) return;
    size_t mid = lo + ((hi-lo) / 2);
    merge_sort(A, lo, mid);
    merge_sort(A, mid+1, hi);
    merge(A, lo, mid, hi);
}
```

count the number of comparisons

Final comments

- Major disadvantage
  - ✓ a sorting algorithm is in-place if it uses O(log n) extra memory
  - ✓ merge sort is not **in-place**
- Improvements
  - ✓ use insertion sort for small arrays
  - ✓ stop recursion if subarray already sorted