CSC 212
Data Structures & Algorithms

Fall 2022 | Jonathan Schrader

Recursions

Housekeeping

Lab 5

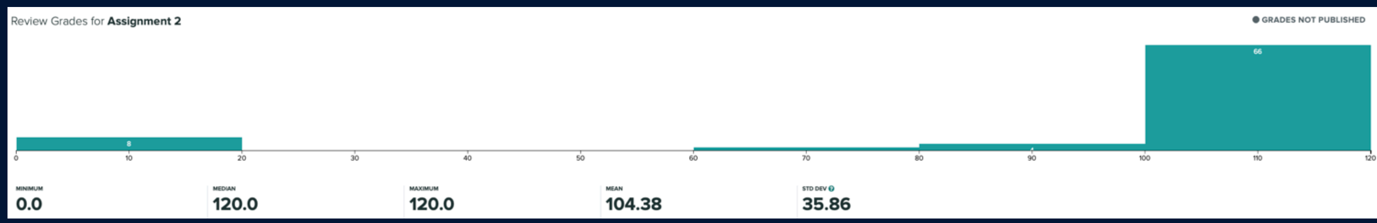
- Lab = (Lab + Recursion(4))

Review / Start Assignment 3

Career Fair Tomorrow

- 11a - 2p

Assignment 2 Outcomes



Input Size	Submissions	>= 70%	% Passing
87	80	66	82.5

Recursion

Recursion

The process of solving a problem by reducing it to smaller versions of itself

```
int someFunction() {
  if (base_case) {
    return // calculate trival solution
  } else {
    // break task into subtasks
    // solve each task recursively
    // merge solutions if necessary
    return someFunction();
  }
}
```

1. Every recursive definition must have one (or more) base cases.
2. The general case must eventually be reduced to a base case.
3. The base case stops the recursion.

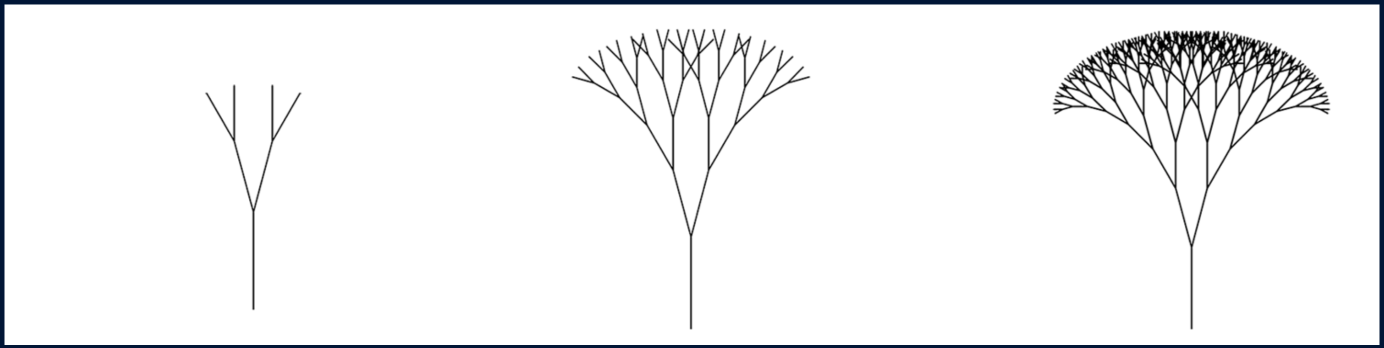
Why recursion?

Can we live without it?

- yes, you can write “any program” with arrays, loops, and conditionals

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

Recursive Call Tree : Sum Array

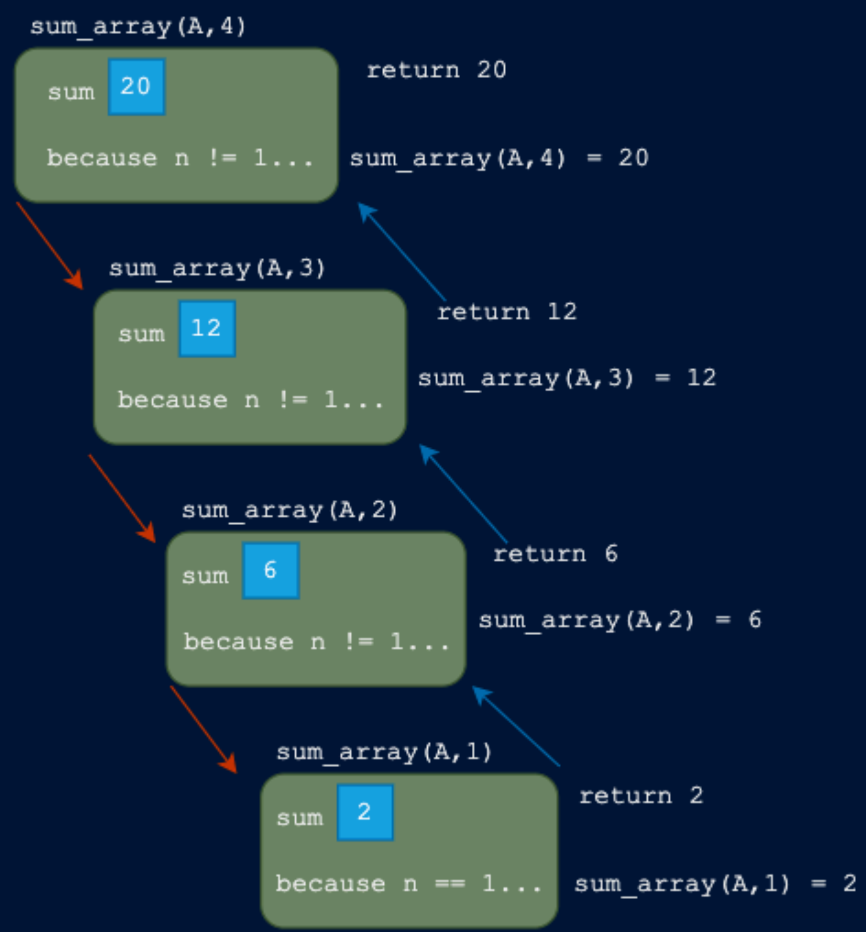
```c
int sum_array(int *A, int n) {
  //basecase
  if (n == 1)
    return A[0];

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

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

base case:

$$sum = A[0] \qquad if\, n = 1$$

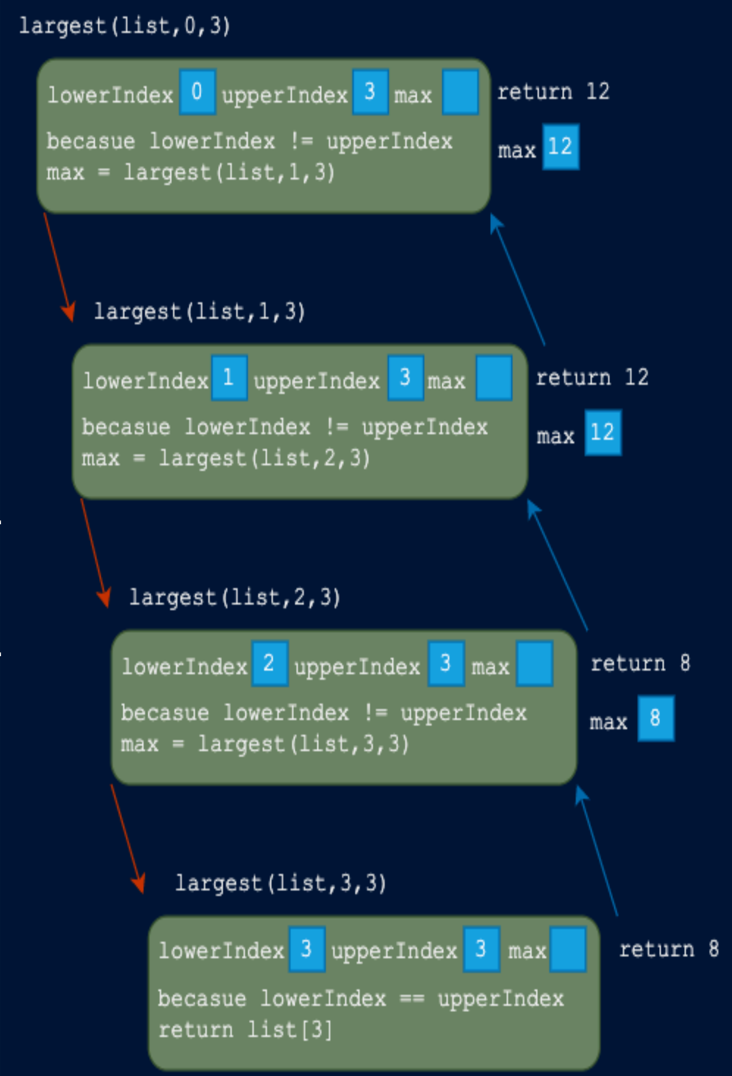
recursive calls:

$$A(n-1) + sum \qquad if\, n > 1$$



Recursive Call Tree : Largest

```
int largest (const int list[], int lowerIndex, int upperIndex) {
  int max;
  if (lowerIndex == upperIndex) //size of the sublist is one
    return list[lowerIndex];
  else {
    max = largest(list, lowerIndex + 1, upperIndex);
    if (list[lowerIndex] >= max)
      return list[lowerIndex];
    else
      return max;
  }
}
```



base case: size of `list` is 1

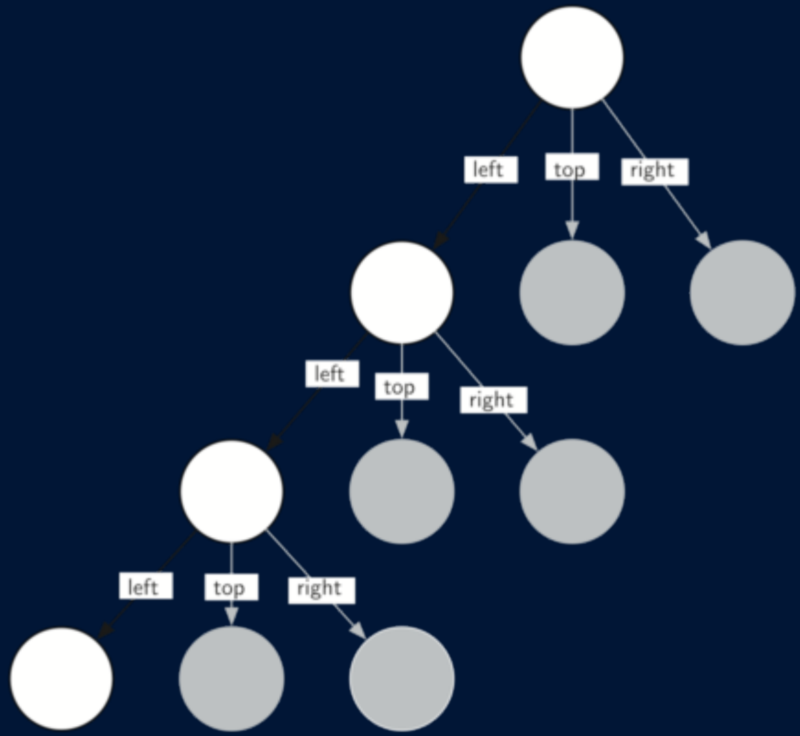
---

recursive calls: size of list is greater than 1

```
To find the largest element in list[a]...list[b]

a. Find the largest element in `list[a + 1]...list[b]`
     and call it max
b. Compare the elements `list[a]` and `max`
   `if (list[a] >= max)`
     the largest element in list[a]...list[b] is list[a]
   otherwise
     the largest element in list[a]...list[b] is max
```

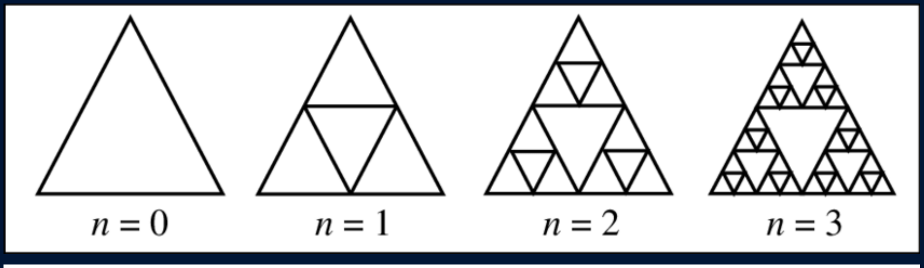
Recursive Call Tree : Serpinski

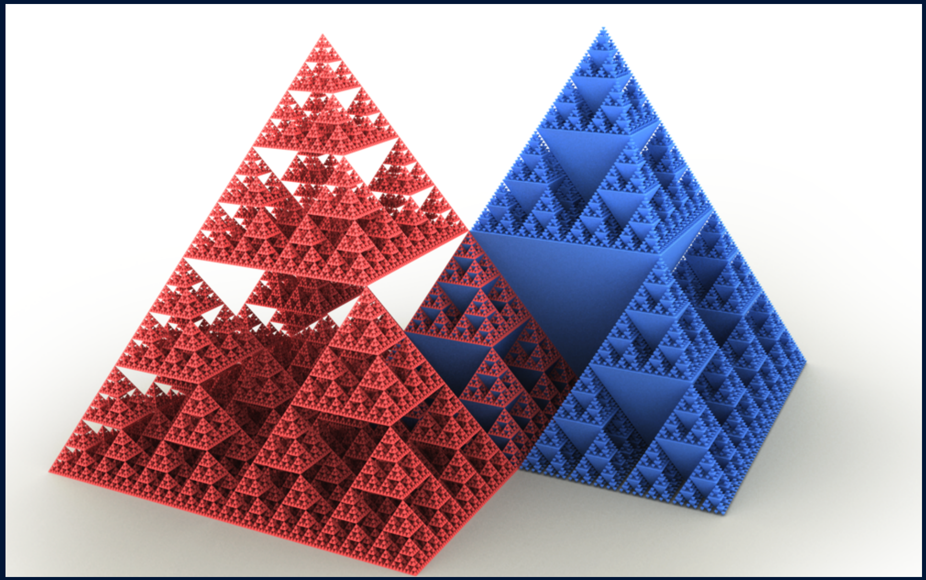
```
void printSierpinski(int n)
{
    for (int y = n - 1; y >= 0; y--) {
        for (int i = 0; i < y; i++)
            cout << " ";

        for (int x = 0; x + y < n; x++) {
      if(x & y)
        cout << " " << " ";
      else
        cout << "* ";
        }
        cout << endl;
    }
}
```



https://www.geeksforgeeks.org/sierpinski-triangle/

Serpinski: Triangle & Pyramid

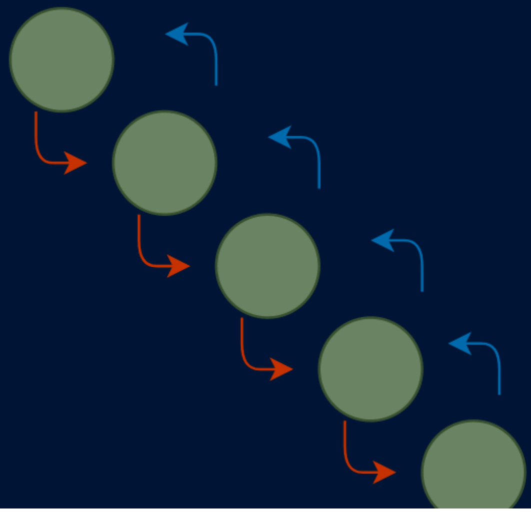


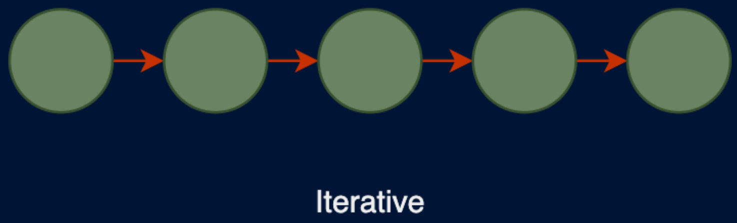


Recursion v. Iteration

```
double power (double x, int n) {
  //basecase
  if (n == 0)
    return 1;

  //recursive call
  return x * power(x, n - 1);
}
```

```
double power (double x, int n) {
  if (n == 0)
    return 1;

  double half = power(x, n / 2);
  if (n % 2 == 0)
    return half * half;
  else
    return x * half * half;
}
```





Binary Search

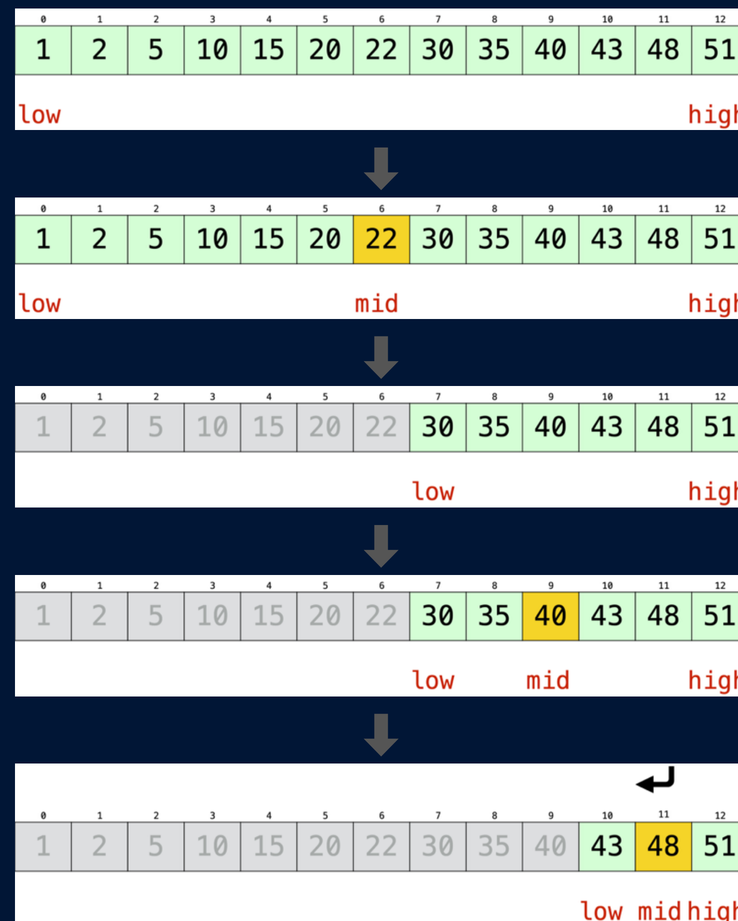
Binary Search

```c
int bsearch(int *A, int lo, int hi, int k) {
  //base case
  if (hi < lo)
    return NOT_FOUND;

  // calculate mid point 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);
}
```

$k = 48$

found at index 11

https://www.geeksforgeeks.org/binary-search/



Recursion Tree Call : Binary Search

```c
int bsearch(int *A, int lo, int hi, int k) {
  //base case
  if (hi < lo)
    return NOT_FOUND;

  // calculate mid point 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);
}
```

```
bsearch(A,0,12,48)
  lo 0   mid 6   hi 12
  because A[mid] < k
    return bsearch(A,7,12,48)

bsearch(A,7,12,48)
  lo 7   mid 9   hi 12
  because A[mid] < k
    return bsearch(A,10,12,48)

bsearch(A,10,12,48)
  lo 10   mid 11   hi 12
  because A[mid] == k
    return 11
```

Unimodal arrays

An array is (*strongly*) *unimodal* if it can be split into an increasing part followed by a decreasing part

1	2	5	16	20	18	17	16	15	12	10	8	5

How to efficiently find the max?

1	2	5	16	20	18	17	16	15	12	10	8	5

Find the max (strongly unimodal)

```c
int max_s_unimodal (int *A, int low, int hi) {
  if (low == hi)
    return A[low];

  int mid = (low + hi) / 2;

  if (A[mid] < A[mid+1])
    return max_s_unimodal(A, mid + 1, hi);
  else
    return max_s_unimodal(A, low, mid);
}
```

1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5
1	2	5	8	15	20	22	20	15	12	10	8	5

```
max_s_unimodal(A, 0, 12)
```

low 0 hi 12

because 22 !< 20
max_s_unimodal(A, 0, 6)

```
max_s_unimodal(A, 0, 6)
```

Unimodal arrays

An array is (*weakly*) *unimodal* if it can be split into a nondecreasing part followed by a nonincreasing part

1	2	5	5	15	20	22	22	35	38	13	8	5

How to efficiently find the max?

1	2	5	5	15	20	22	22	35	38	13	8	5

Find the max (weakly unimodal)

```
int max_w_unimodal (int *A, int low, int hi) {
  if (low == hi)
    return A[low];

  int   mid =   (low + hi) / 2;

  if (A[mid] < A[mid + 1])
    return max_w_unimodal(A, mid + 1, hi);
  else if (A[mid] > A[mid + 1])
    return max_w_unimodal(A, low, mid);
  else {
    int left = max_w_unimodal(A, mid+1, hi);
    int right = max_w_unimodal(A, low, mid);
    return std::max(left, right);
  }
}
```

1	2	5	5	15	20	22	22	35	38	13	8	5
1	2	5	5	15	20	22	22	35	38	13	8	5
1	2	5	5	15	20	22	22	35	38	13	8	5