Outcome (1) and (3): How to analyze recursive algorithms?
Will teach an algortihm design technique.

Recurrence, recursive function, divide-and-conquer approach

Review:
A function is recursive if it is defined in terms of itself.

Let n be positive integers

e.g. $f(n)=\begin{cases} 1 & n=1 \\ f(n-1)+n & n>1 \end{cases}$

$$\begin{aligned} f(4) &= f(3) + 4 \\ &= f(2) + 3 + 4 \\ &= f(1) + 2 + 7 \\ &= 1 + 9 \\ &= 10 \end{aligned}$$

Two components of a recursion

Base case: value of f(n) is known

Recursive part: by following this recursion, the value of f(n) can be calculated by making progress towards the base case

(non-recursive formula)
Q: How to find a "closed form" for f(n)?
(how to solve it?)

One simple approach: The Iteration Method

$$\text{e.g. } f(n)=\begin{cases} 1 & n=1 \\ f(n-1)+n & n>1 \end{cases}$$

$$\begin{aligned} f(n) &= f(n-1) + n \\ &= f(n-2) + (n-1) + n \\ &= \ldots.. \\ &= f(1) + 2 + 3 + \ldots + n \\ &= n(n+1)/2 \end{aligned}$$

Remarks:

1) Then, formally prove it by MI.

2) Sometimes we only need to know the solution in asymptotic notation.

MI proof (sketch)

Base: when $n = 1$, $f(n) = 1$ by definition.
$n(n+1)/2 = 1$, so $f(n) = n(n+1)/2$ is true.

Induction: for $n = k$,

$$\begin{aligned} f(k) &= f(k-1)+k && \text{by definition} \\ &= (k-1)k/2 + k && \text{by hypothesis} \\ &= k(k+1)/2. \end{aligned}$$

e.g. $f(n) = \begin{cases} 1 & n=1 \\ f(n-1)+n & n>1 \end{cases}$

$$\begin{aligned} f(n) &= f(n-1) + n \\ &= f(n-2) + (n-1) + n \\ &= \ldots.. \\ &= n(n+1)/2 \end{aligned}$$

If we only need to show that $f(n) = O(n^2)$, we only need to prove $f(n) \leq cn^2$ for some constant c for $n \geq 1$ using MI.

Step 1: try to get the value of c (do it in rough paper)

For the induction to go through,

$f(n) = f(n-1) + n \leq c(n-1)^2 + n = cn^2 - (2cn - c - n)$.

Since we want $f(n) \leq cn^2$, so we set $2cn - c - n \geq 0$

$\Rightarrow c \geq n/(2n-1)$ where $n/(2n-1) \leq 1$ for $n \geq 1$

So, we can set $c = 1$.

Step 2: formal proof

We will show that $f(n) \leq n^2$ for all $n \geq 1$.

Base:....

Induction: ....

Take-home exercise:

Prove $f(n) = O(n)$: $f(n) = 3$ for $n = 1$, otherwise $f(n) = f(n-1) + 10$.

To analyze a recursive algorithm,

(1) we model the running time of the algorithm as a recursive function, then

(2) solve the recursive function to get the answer in asymptotic notation.

```
Alg(n) {
    if (n = 1) return 1;
    temp = Alg(n-1);
    for (i = 1 to n)
        temp = temp + 1;
    return temp;
}
```

Let f(n) be the running time of Alg(n).

$$\text{e.g. } f(n)=\begin{cases} c_1 & n=1 \\ f(n-1)+c_2 n & n>1 \end{cases}$$

where c_1 and c_2 are constants

The time complexity of Alg(n) is $O(n^2)$.

Solving recurrences (obtain the asymptotic bounds on the solution)

[MIT, ch4.3] substitution method

[MIT, ch4.4] recursion-tree method

[MIT, ch4.5] master method

Or you can get related material from web or any related reference books

Can you learn it yourself or let me know later if you want a tutorial [will NOT be a main topic in the exam]?

Divide-and-conquer approach usually gives a recursive algorithm and the time complexity of the algorithm is usually formulated as a recursive function. Solving the recursive function will give the time complexity of the algorithm.

- ➢Divide problem into subproblems
- ➢Conquer subproblems recursively
- ➢Combine subproblem solutions

Simple example: calculate n!

Let f(n) = n!,
then f(1) = 1 and f(n) = n*f(n-1)

```
fact(n){
    if (n = 1) return 1;
    else return n*fact(n-1);
}
```

Let T(n) be the running time of fact(n)

$$\text{Then, } T(n) = \begin{cases} c1 & \text{if } n=1 \\ T(n-1)+c2 & \text{if } n>1 \end{cases}$$

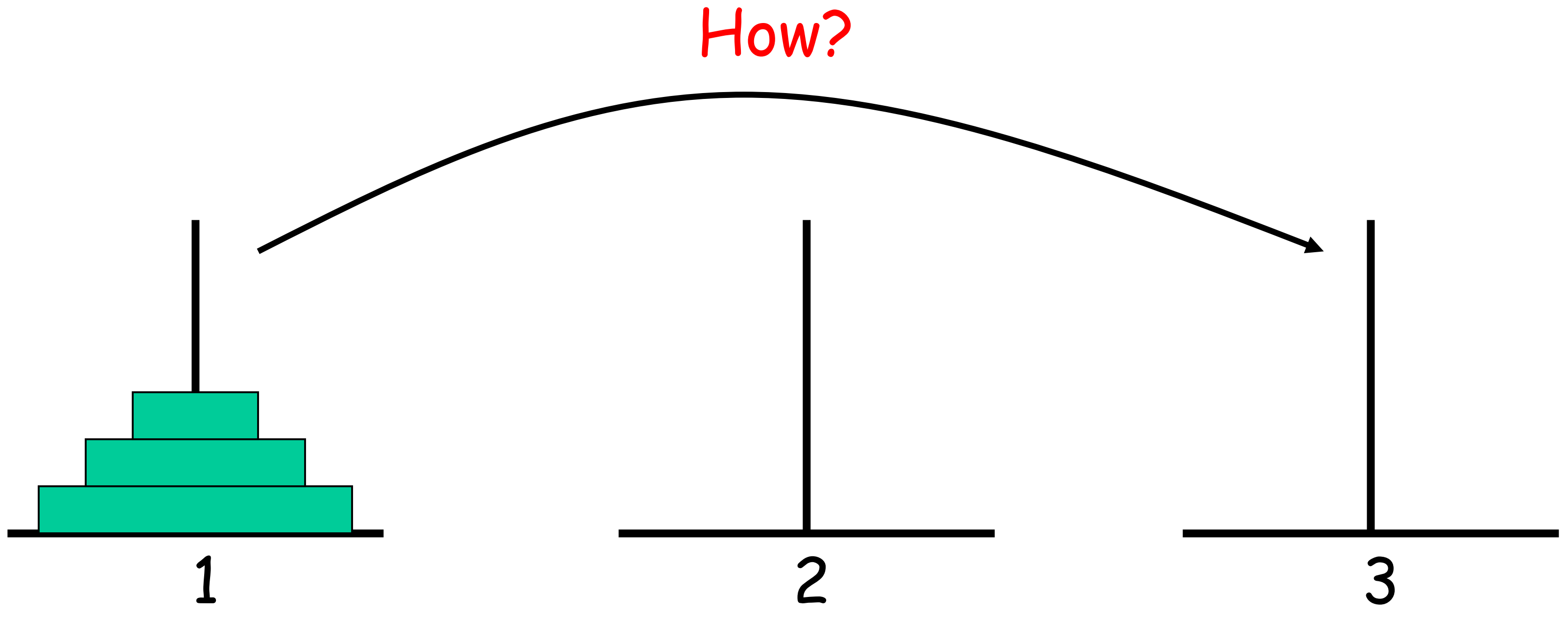
where c1,c2 are constants

$$\begin{aligned} T(n) &= T(n-1)+c2 \\ &= T(n-2)+c2+c2 \\ &= \ldots.. \\ &= T(1) + (n-1)c2 = c1 + (n-1)c2 \\ &= O(n) \end{aligned}$$

Review: Tower of Hanoi

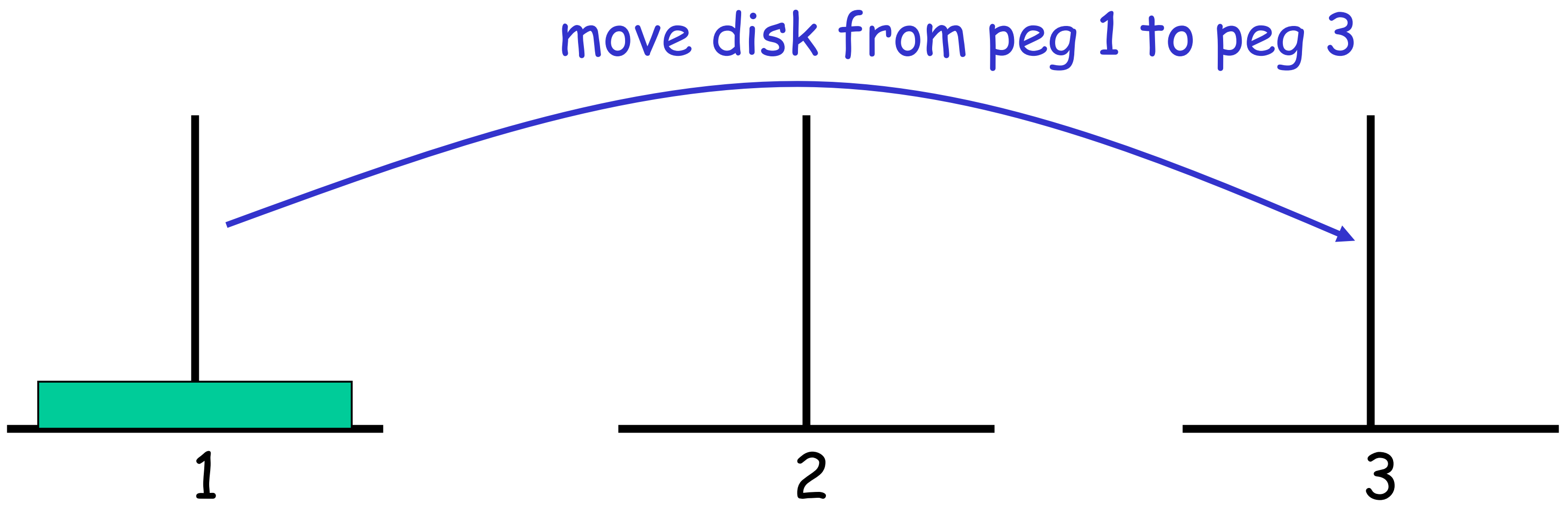
Given 3 pegs and n disks of different sizes placed in an increasing order of size on one peg, transfer the disks from the original peg to another peg such that:

- only one disk can be moved in each step
- in each peg, the disks must be in increasing order of size

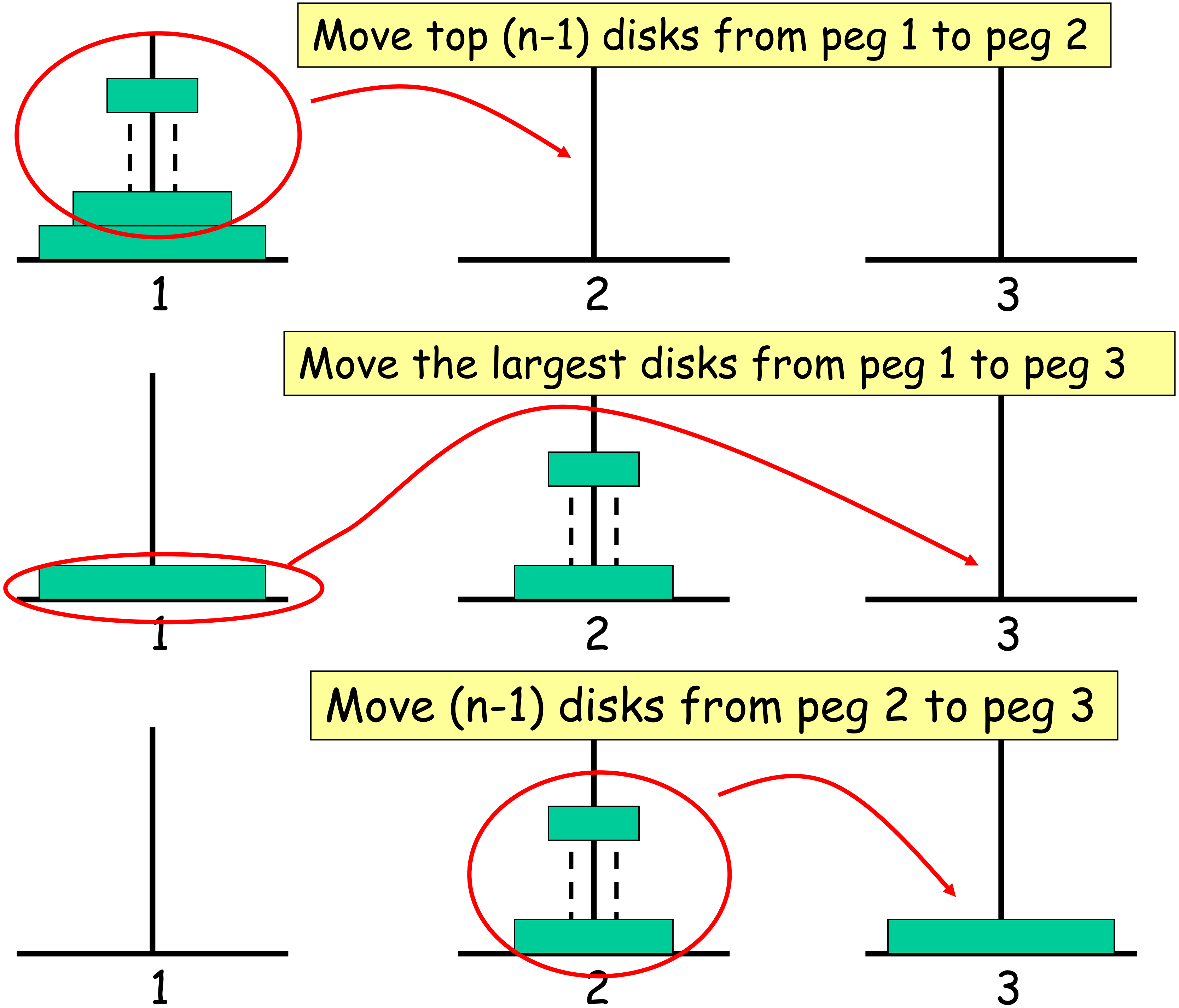


Solve it recursively:

Base case: if there is only one disk.



Assuming that we know how to move n-1 disks, can we solve it for n disks?


Move top (n-1) disks from peg 1 to peg 2

Move the largest disks from peg 1 to peg 3

Move (n-1) disks from peg 2 to peg 3

```
TofH(A,B,C,n)  // source peg: A, destination peg: C
{
    if (n=1)
        move disk from A to C
    else {
        TofH(A,C,B,n-1);
        move disk from A to C;
        TofH(B,A,C,n-1);
    }
}
```

To call the program:
TofH(1, 2,3, 3) // 3 disks

What is the number of moves?

Let f(n) be the number of moves

$$f(n) = \begin{cases} 1 & n=1 \\ 2f(n-1)+1 & n>1 \end{cases}$$

$$\begin{aligned} f(n) &= 2f(n-1) + 1 \\ &= 2(2f(n-2)+1) + 1 = 2^2f(n-2) + 2 + 1 \\ &= 2^2(2f(n-3)+1) + 2 + 1 = 2^3f(n-3) + 2^2 + 2 + 1 \\ &= \ldots. \\ &= 2^{n-1}f(1) + 2^{n-2} + \ldots + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + \ldots + 2 + 1 \\ &= 2^n - 1 \end{aligned}$$

How about time complexity of the algorithm?

Review: Divide-and-conquer approach

(1) In order to obtain a solution for input size n, we assume the solution(s) for smaller n (subproblem) is available, then construct the final solution from these solutions.

E.g. computing n! (assuming (n-1)! is available)
E.g. solving the Tower of Hanoi with n disks
(assuming we know how to do (n-1) disks)

(2) Don't forget the base case:
E.g. n! = 1 when n = 1
E.g. Solving the Tower of Hanoi problem with 1 disk

(3) Derive a "recursive" algorithm from this idea.

(4) Capture the running time of the algorithm by a recursive function, then solve the recurrence for the time complexity of the algorithm.

Pay attention to the subproblem definition =>

A student tries to solve the Tower of Hanoi problem recursively as follows: Assume that we know how to move n/2 disks (n: even for simplicity)

```
TofH2(A,B,C,n)  // source peg: A, destination peg: C
{
    if (n=1)
        move disk from A to C
    else {
        TofH2(A,C,B,n/2); // move 1st half from A to B
        TofH2(A,B,C,n/2); // move 2nd half from A to C
        TofH2(B,A,C,n/2); // move 1st half from B to C
        }
}
```

Is it correct?

** Definition: TofH(A,B,C,n): move n disks from A to C with B empty or having disks with the correct order & all larger than the given n disks.

Q: to solve this problem, is $2^n - 1$ the minimum number of moves?

Example: Find the largest number from a given set of n numbers using divide-and-conquer approach

Divide the problem into subproblems
1) Solve base cases for subproblems
2) Assume that solutions of subproblems are available, combine them to find the solution for the original problem

There may be more than one way to define the subproblems.

Recall that for the problem of Tower of Hanoi:

problem: moving n disks
subproblem: moving (n-1) disks

We can do something similar here:

problem: finding largest in n numbers

subproblem: finding largest in first (n-1) numbers

Base case: if there is only one number, return that number!

Recurrence: if the subproblem is solved, say x is the solution, how can we find the largest number among all n numbers?

```
Check if (x > A[n])     // A stores the numbers
        return x;
else
        return(A[n]);
```

```
largest1(A[1..n]) {
    if (n = 1) return A[1];
    else
        x = largest1(A[1..n-1]);
        if (x > A[n]) return x;
        else return A[n];
}
```

Time complexity:

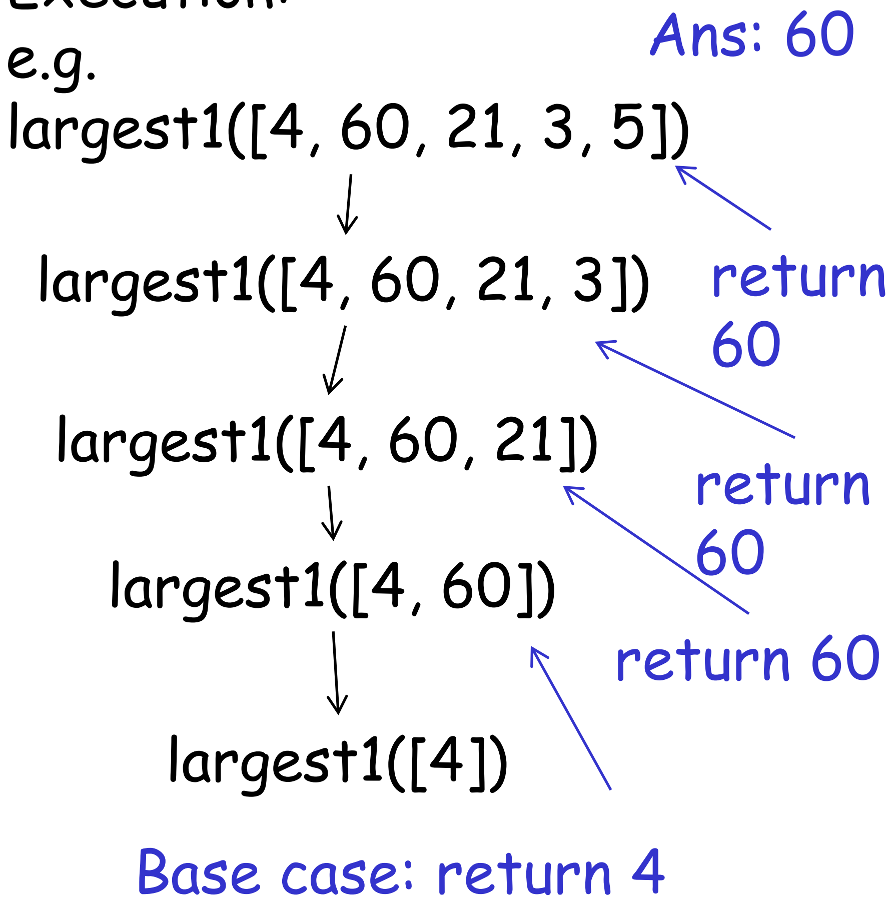
$T(n) = c1$ if $n = 1$;

$T(n) = T(n-1) + c2$ for $n > 1$.

=> $T(n) = O(n)$.

Execution:

e.g.



Example: Find the largest number from a given set of n numbers

Another way to define subproblems:
Divide the numbers into two groups of size n/2 each.

Base case: If the group has only one number, return it.
Recurrence: Assume that the two subproblems have been solved and x1, x2 are returned, what we do to get the final answer?

```
Check if (x1 > x2)
        return x1;
else
        return x2;
```

```
largest2(A[1..n])        // for simplicity, assume n is a power of 2
{
    If (n = 1)
            return A[1];
    else {
            x1 = largest2(A[1..n/2]);
            x2 = largest2(A[n/2+1..n]);
            if (x1 > x2)
                    return x1;
            else
                    return x2;
        }
}
```

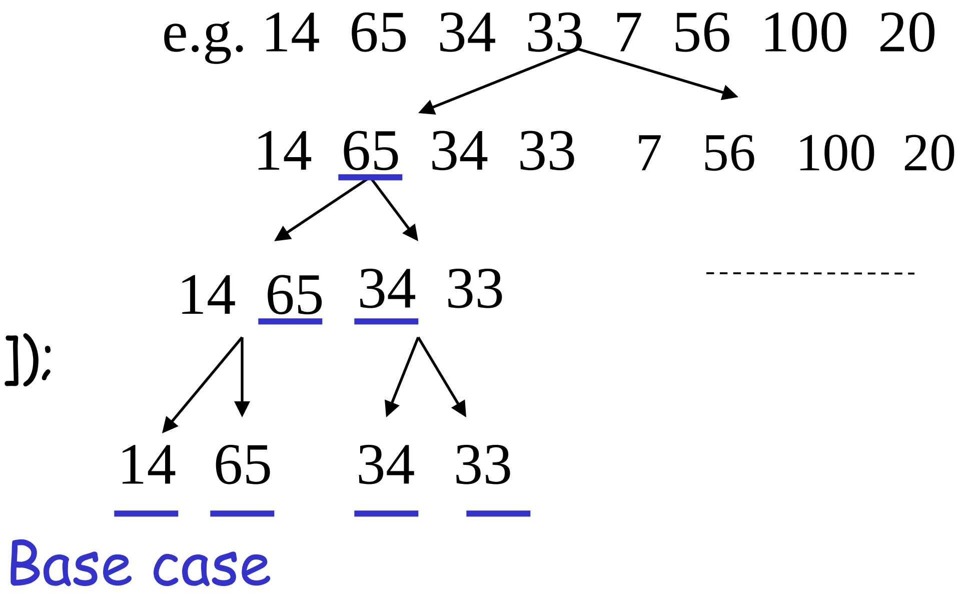
Let $T(n)$ be the time complexity of largest2.

$T(1) = c1$

$T(n) = 2T(n/2) + c2$ where c1, c2: positive constants

$\Rightarrow T(n) = O(n)$

```
largest2(A[1..n])
{
   If (n = 1)
         return A[1];
   else {
     x1 = largest2(A[1..n/2]);
     x2 = largest2(A[n/2+1..n]);
     if (x1 > x2)
         return x1;
     else
         return x2;
         }
}
```



Remark: This problem, of course, can be solved using a more straightforward approach, so it is only for illustration purpose

Note: the following straightforward approach will give an algorithm with the same time complexity.

```
largest(A[1..n])
{
        maximum = A[1];
        for i = 2 to n do
                if (A[i] > maximum)
                        maximum = A[i];
        return maximum;
}
```

Remarks:

Theorem :

If T(n) = a T(n/b) + f(n) and f(n) = $\Theta(n^p)$, then

$$T(n) = \begin{cases} \Theta(n^p) & \text{if } a < b^p \\ \Theta(n^p \log n) & \text{if } a = b^p \\ \Theta(n^{\log_b a}) & \text{if } a > b^p \end{cases}$$ where $a \geq 1$, $b \geq 2$, $p \geq 0$

How about $n \neq b^k$ (e.g. b = 2) for any integer k?

Usually the time complexity remains the same, but the algorithm will be complicated to handle extra cases. Interested students can explore more on this aspect.

Note: recursion may not always be a good solution

The Fibonacci numbers

Definition:
$F_1 = 1$
$F_2 = 1$
$F_i = F_{i-1} + F_{i-2}$ for $i > 2$

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Problem: Generate the nth Fibonacci number

Algorithm 1: Direct implementation of recursive definition

```
FIB1(n) {
   if (n <=2) then
      return 1
   else
      return (FIB1(n-1) + FIB1(n-2))
}
```

Time complexity?

$$T(n) = \begin{cases} c1 & \text{if } n \le 2 \\ T(n-1)+T(n-2) + c2 & \text{if } n > 2 \end{cases}$$

where c1,c2 are constants

Claim: $T(n) > 2^{(n-2)/2}$

Proof:
$$\begin{aligned}
T(n) &> T(n-1) + T(n-2) \\
&> T(n-2) + T(n-3) + T(n-2) \\
&= 2T(n-2) + T(n-3) \\
&> 2T(n-2) \\
&> 2(2T(n-4)) = 4T(n-4) \\
&> 4(2T(n-6)) = 8T(n-6) \\
&> \ldots. \\
&> 2^k\, T(n-2k)
\end{aligned}$$

If n is even, $n-2k = 2$. So, $T(n) > 2^{(n-2)/2}$

If n is odd, $n-2k = 1$. So, $T(n) > 2^{(n-1)/2}$

Therefore, T(n) is exponential in n and FIB1 runs in exponential time.

Note: F_n can be shown to be $\cong \frac{1}{\sqrt{5}} 1.618^n$ $T(n) = \Theta(1.618^n)$

Can we do better?

In the recursive version, we need to recompute FIB1(n) values for many n's.

Algorithm 2: Calculate F_n by a loop

```
FIB2(n) {
   if (n <=2) then
         return 1
   else
         p = 1
         q = 1
         for i = 3 to n do
               r = p + q
               p = q
               q = r
         return r
}
```

Time Complexity?

$\Theta(n)$

Q: Do you think we can still do better?

Revisit: Computing Fibonacci Numbers

We first consider the following problem.
Given a 2x2 matrix A, design an algorithm to compute A^n for any given $n \geq 1$.

<u>1st Attempt</u>: A straight-forward implementation

```
Power1(A, n) {     // n ≥ 1; A: 2 × 2 matrix
    if (n = 1)
            return A;
    else
            return 2-Multiply(A, Power1(A, n-1));  // details of 2-
}                                                  // Multiply omitted
```

Complexity:
2-Multiple takes constant time.
Power1(A, n) takes O(n) time.

2nd Attempt:

[Hint: if n is always a power of 2, can you design a better (faster) algorithm?]

```
Power2(A, n) {     // n = 2^k for a non-neg int k; A: 2 × 2 matrix
    if (n = 1)
        return A;
    else
        B = Power2(A, n/2);
        return 2-Multiply(B, B);
}
```

Complexity:

Let T(n) be the running time for Power2(A, n).

$$T(n) = \begin{cases} c1 & \text{if } n = 1 \\ T(n/2) + c2 & \text{if } n > 1 \end{cases}$$

where c1,c2 are constants

$T(n) = O(\log n)$.

How about for general n?

e.g. $n = 10$

$A^{10} = A^5 \times A^5$

To compute A^5: $A^5 = A \times A^4$

To compute A^4: $A^4 = A^2 \times A^2$

And $A^2 = A \times A$.

If n is odd,

$$A^n = A \times (A^{n-1/2})^2$$

else

$$A^n = A^{n/2} \times A^{n/2}$$

```
Power3(A, n) {      // $n \geq 1$; A: $2 \times 2$ matrix
    if (n = 1)
        return A;
    else
        if (n is odd)
            B = Power3(A, (n-1)/2);
            return(2-Multiply(A, 2-Multiply(B, B));
        else
            B = Power3(A, n/2);
            return(2-Multiply(B, B));
}
```

Complexity: $O(\log n)$

How this relates to our problem (computing the Fibonacci #)? [MIT, Ex31-3, p.981]

Hint: Consider the following matrix and its powers.

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

See if you can design a faster algorithm for computing the n-th Fibonacci #.

One more example: The maximum subarray problem

Given an array A[1..n] of integers, find a nonempty, contiguous subarray of A whose values have the largest sum.

e.g. $$-4 \quad \underbrace{6 \quad -3 \quad -1 \quad 6 \quad 1}_{\text{A[2..6] gives the largest sum (9)}} \quad -2$$

1st attempt (brute-force approach)
Idea: check all possible subarrays

```
Max_subarray1(A, n){
        max = -∞
        for i = 1 to n
                for j = i to n
                        sum = 0;
                        for k = i to j
                                sum = sum + A[k];
                        if sum > max
                                max = sum;
                                ans = (i, j, max);
        return ans;
```

Time complexity: $O(n^3)$

Can we do better?

This brute-force algorithm can be further improved due to many redundant addition

i/j	1	2	3	4	…
1	1				
2	2	1			
3	3	2	1		
4	4	3	2	1	….
…..	…..	…..	…..	…..	….

redundant

When i = 1; j = 1, we compute sum = A[1]
When i = 1; j = 2, we compute sum = A[1] + A[2]
When i = 1; j = 3, we compute sum = $\boxed{A[1] + A[2]}$ + A[3]

When i = 1; j = 4, we compute sum = $\boxed{A[1] + A[2] + A[3]}$ + A[4]

Do you know how to improve it to $O(n^2)$ time?

2nd attempt (brute-force approach with improvement)
Idea: check all possible subarrays, faster computation of sum

```
Max_subarray2(A, n){
    max = $-\infty$
    for i = 1 to n
        sum = 0;
        for j = i to n
            ~~sum = 0;~~
            ~~for k = i to j~~
            sum = sum + A[j];
            if sum > max
                max = sum;
                ans = (i, j, max);
    return ans;
```

Time complexity:
$O(n^2)$

Can we do better?

3rd attempt: How about divide-and-conquer approach?

middle

You know how to find the maximum subarray in the 1st half
You know how to find the maximum subarray in the 2nd half
What else you need to find?

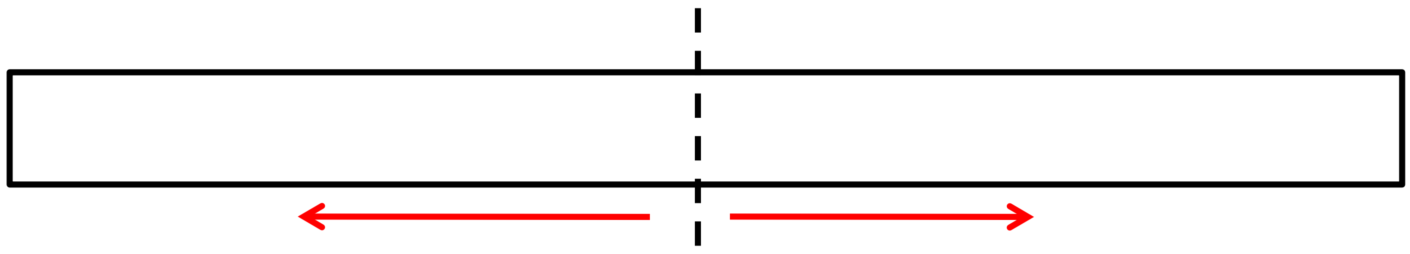
e.g. 3 -1 3 - 5 4 ¦ 5 -8 2 0 4

Recursion for 1st half returns (A[1..3], max = 5)
Recursion for 2nd half returns (A[8..10], max = 6)
But the correct answer is (A[5..6], max = 9)

We need to find also the maximum subarray across the mid point!

Finding the max subarray across the middle:

Idea:



middle

Search for A[i, n/2] for $1 \leq i \leq n/2$ with maximum sum.

```
max = -∞; sum = 0;
for i from n/2 downto 1
        sum = sum + A[i];
if sum > max
        max = sum
        ans = i
return (i, max)
```

Search for A[, n/2+1, j] for $n/2+1 \leq j \leq n$ with maximum sum.

```
max = -∞; sum = 0;
for j from n/2+1 to n
        sum = sum + A[i];
if sum > max
        max = sum
        ans = j
return (j, max)
```

Combine these two results
It takes O(n) time.

```
Max_subarray3(A, p, q) {
        (i1, j1, max1) = Max_subarray3(A, p, p+q/2);
        (i2, j2, max2) = Max_subarray3(A, p+q/2+1, q);
        (i3, j3, max3) = Max_subarray_middle(A, p, q);
        if max 1 > max 2
                if max 1 > max 3
                        return (i1, j1, max1)
                else
                        return(i3, j3, max3)
        else
                if max2 > max3
                        return(i2, j2, max2)
                else
                        return(i3, j3, max3)
}
```

O(n) time

Time complexity: $T(n) = 2T(n/2) + cn$ for $n > 1$ ($n = 2^k$)
=> $T(n) = O(n \log n)$

Can we do better?

Exercises

1) Given an array of numbers, design a recursive algorithm to find the average of all numbers in the array and analyze the time complexity of your algorithm.

2) Design a recursive algorithm for reversing the entries in an array and analyze the time complexity of your algorithm. You are only allowed to use constant amount of extra storage.