TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Data structures and Algorithms

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Course outline

Chapter 1. Fundamentals

Chapter 2. Basic data structures

Chapter 3. Tree

Chapter 4. Sorting

Chapter 5. Searching

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chapter 1. Fundamentals

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Contents

Contents

Example: The maximum subarray problem

- Given an array of n numbers:

  $a_1, a_2, \ldots, a_n$

  The contiguous subarray $a_i, a_{i+1}, \ldots, a_j$ with $1 \le i \le j \le n$ is a subarray of the given array and $\sum_{k=i}^{j} a_k$ *is called as the value of this subarray*

  The task is to find the maximum value of all possible subarrays, *in other words, find the maximum* $\sum_{k=i}^{j} a_k$. The subarray with the maximum value is called as the maximum subarray.

Example: Given the array -2, **11, -4, 13**, -5, 2 then the maximum subarray is 11, -4, 13 with the value = 11+ (-4)+13 =20

➔ This problem can be solved using several different algorithmic techniques, including brute force, divide and conquer, dynamic programming, etc.

1. Introductory example: the max subarray problem

1.1.1. Brute force

1.1.2. Brute force with better implement

1.1.3 Recursive algorithm

1.1.4. Dynamic programming

1. Introductory example: the max subarray problem

1.1.1. Brute force

1.1.2. Brute force with better implement

1.1.3 Recursive algorithm

1.1.4. Dynamic programming

1.1.1. Brute force algorithm to solve max subarray problem

- The first simple algorithm that one could think about is: browse all possible sub-arrays:

$$a_i, a_{i+1}, \ldots, a_j \quad \text{với} \quad 1 \le i \le j \le n,$$

then calculate the value of each sub-array in order to find the maximum value.

- The number of all possible sub-arrays:

$$C(n, 1) + C(n, 2) = n^2/2 + n/2$$

Brute force algorithm: browse all possible sub-array

Index i	0	1	2	3	4	5
a[i]	-2	11	-4	13	-5	2

```
i = 0: (-2), (-2, 11), (-2,11, -4), (-2,11,-4,13), (-2,11,-
   4,13,-5), (-2,11,-4,13,-5,2)
i = 1: (11), (11, -4), (11, -4, 13), (11, -4, 13, -5), (11, -4,
   13, -5, 2)
i = 2: (-4), (-4, 13), (-4, 13, -5), (-4,13,-5,2)
i = 3: (13), (13,-5), (13, -5,2)
i = 4: (-5), (-5, 2)
i = 5: (2)
```

```
int maxSum = 0;
for (int i=0; i<n; i++) {
   for (int j=i; j<n; j++) {
      int sum = 0;
      for (int k=i; k<=j; k++)
          sum += a[k];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

Brute force algorithm: browse all possible sub-array

- **Analyzing time complexity**: we count the number of additions that the algorithm need to perform, it means we count the statement

```
sum += a[k]
```

must perform how many times.

The number of additions:

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i+1)=\sum_{i=0}^{n-1}(1+2+\ldots+(n-i))=\sum_{i=0}^{n-1}\frac{(n-i)(n-i+1)}{2}$$

$$=\frac{1}{2}\sum_{k=1}^{n}k(k+1)=\frac{1}{2}\left[\sum_{k=1}^{n}k^2+\sum_{k=1}^{n}k\right]=\frac{1}{2}\left[\frac{n(n+1)(2n+1)}{6}+\frac{n(n+1)}{2}\right]$$

$$=\frac{n^3}{6}+\frac{n^2}{2}+\frac{n}{3}$$

```
int maxSum = 0;
for (int i=0; i<n; i++) {
   for (int j=i; j<n; j++) {
      int sum = 0;
      for (int k=i; k<=j; k++)
          sum += a[k];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

1. Introductory example: the max subarray problem

1.1.1. Brute force

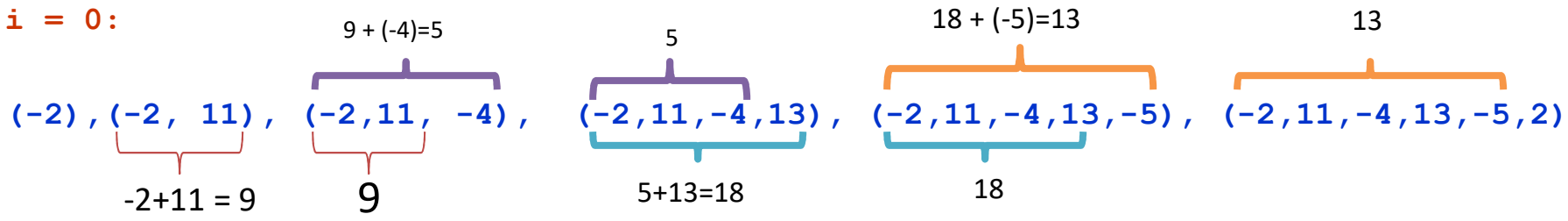
1.1.2. Brute force with better implement

1.1.3 Recursive algorithm

1.1.4. Dynamic programming

1.1.2. A better implementation
Brute force algorithm: browse all possible sub-array

Index i	0	1	2	3	4	5
a[i]	-2	11	-4	13	-5	2



We could get the sum of elements from i to j by just using one addition:

$$\sum_{k=i}^{j} a[k] = a[j] + \sum_{k=i}^{j-1} a[k]$$

The sum of elements from i to j — The sum of elements from i to j-1

1.1.2. A better implementation
Brute force algorithm: browse all possible sub-array

Index i	0	1	2	3	4	5
a[i]	-2	11	-4	13	-5	2

```
i = 0: (-2), (-2, 11), (-2,11, -4), (-2,11,-4,13), (-2,11,-4,13,-5), (-2,11,-
   4,13,-5,2)
i = 1: (11), (11, -4), (11, -4, 13), (11, -4, 13, -5), (11, -4, 13, -5, 2)
i = 2: (-4), (-4, 13), (-4, 13, -5), (-4,13,-5,2)
i = 3: (13), (13,-5), (13, -5,2)
i = 4: (-5), (-5, 2)
i = 5: (2)
```

We could get the sum of elements from i to j by just using one addition:

$$\sum_{k=i}^{j} a[k] = a[j] + \sum_{k=i}^{j-1} a[k]$$

The sum of elements from i to j — The sum of elements from i to j-1

```
int maxSum = 0;
for (int i=0; i<n; i++) {
   for (int j=i; j<n; j++) {
      int sum = 0;
      for (int k=i; k<=j; k++)
          sum += a[k];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

```
int maxSum = a[0];
for (int i=0; i<n; i++) {
   int sum = 0;
   for (int j=i; j<n; j++) {
      sum += a[j];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

1.1.2. A better implementation
Brute force algorithm: browse all possible sub-array

- **A better implementation:**

We could get the sum of elements from i to j by just using one addition:

$$\underbrace{\sum_{k=i}^{j} a[k]}_{\text{The sum of elements from } i \text{ to } j} = a[j] + \underbrace{\sum_{k=i}^{j-1} a[k]}_{\text{The sum of elements from } i \text{ to } j\text{-1}}$$

```
int maxSum = 0;
for (int i=0; i<n; i++) {
   for (int j=i; j<n; j++) {
      int sum = 0;
      for (int k=i; k<=j; k++)
          sum += a[k];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

```
int maxSum = a[0];
for (int i=0; i<n; i++) {
   int sum = 0;
   for (int j=i; j<n; j++) {
      sum += a[j];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

1.1.2. A better implementation
Brute force algorithm: browse all possible sub-array

- **Analyzing time complexity**: we again count the number of additions that the algorithm need to perform, it means we count the statement

```
Sum += a[j]
```

must perform how many times.

The number of additions:

$$\sum_{i=0}^{n-1}(n-i) = n + (n-1) + \ldots + 1 = \frac{n^2}{2} + \frac{n}{2}$$

This number is exactly the number of all possible sub-arrays ➔ it seems this implementation is good as we examine each subarray exactly once.

```
int maxSum = a[0];
for (int i=0; i<n; i++) {
   int sum = 0;
   for (int j=i; j<n; j++) {
      sum += a[j];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

1. Introductory example: the max subarray problem

1.1.1. Brute force

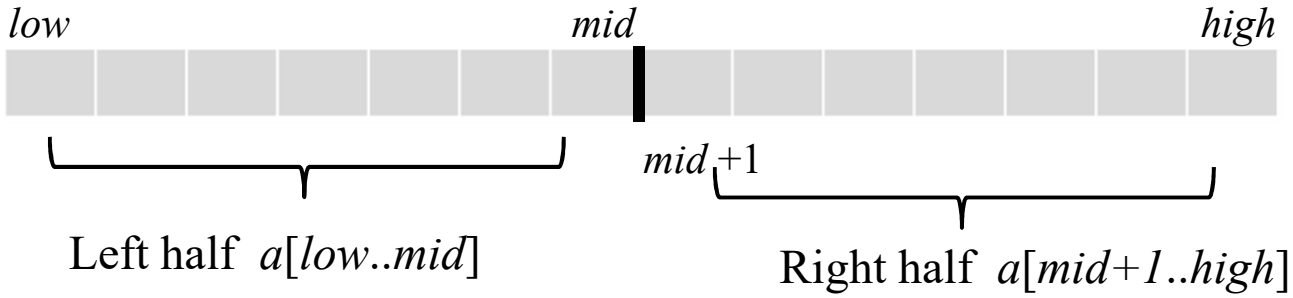
1.1.2. Brute force with better implement

1.1.3 Recursive algorithm

1.1.4. Dynamic programming

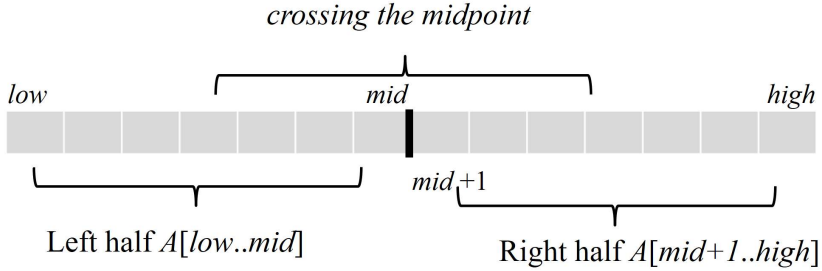
1.1.3. Recursive algorithm to solve max subarray problem

- We could even build better algorithm by using divide-and-conquer technique. This technique consists of the following steps:
  - Divide the problem into a number of sub-problems that are smaller instances of the same problem.
  - Conquer the sub-problems by solving them recursively.
    - ***Base case: If the sub-problems are small enough, just solve them by brute force.***
  - Combine the sub-problem solutions to give a solution to the original problem.
- To solve the maximum sub-array problem:
  - Use the middle element to divide the array into 2 sub-arrays (left half and right half) with halving length



1.1.3. Recursive algorithm to solve max subarray problem

- To get the solution, we see there are 3 possible locations of the maximum subarray $A[i..j]$ of $A[low..high]$, where $mid = \lfloor (low+high)/2 \rfloor$:
  - entirely in the left half $A[low..mid]$ ($low \le i \le j \le mid$)
  - entirely in the right half $A[mid+1..high]$ ($mid < i \le j \le high$)
  - crossing the midpoint ($low \le i \le mid < j \le high$) {the maximum subarray starts at the left half and ends at the right half}

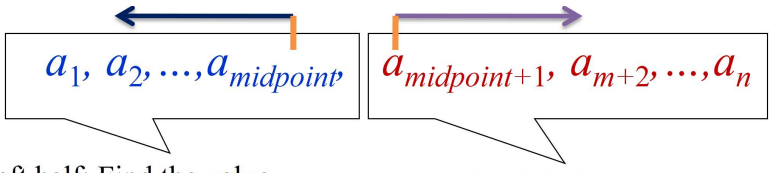


- Therefore, if denote the value of the maximum subarray at the left half w_L, at the right half w_R, and crossing the midpoint w_M, then the value need to determine is **$\max(w_L, w_R, w_M)$**

```
MaxSub(a, low, high);
{
  if (low = high) return a[low] //base case: only 1 element
  else
  {   mid = (low+high)/2;
      wL = MaxSub(a, low, mid);
      wR = MaxSub(a, mid+1, high);
      wM = MaxCrossMidPoint(a, low, mid, high);
      return max(wL, wR, wM);
  }
}
```

1.1.3. Recursive algorithm to solve max subarray problem

- Finding the value of max subarray on the left half (w_L) and right half (w_R) could be done recursively:
  - Base case: left/right half consists of only one element
- Finding the value $\mathbf{w_M}$ of the max subarray starting at the left half and ending at the right half as following:
  - On the left half: find the value w_{ML} of the max subarray ending at the midpoint
  - On the right half: find the value w_{MR} of the max subarray starting at the midpoint+1
  - Then $w_M = w_{ML} + w_{MR}$.



On the left half: Find the value W_{ML} of the max subarray ending at element $a_{midpoint}$

On the right half: Find the value W_{MR} of the max subarray starting at the element $a_{midpoint+1}$

```
MaxSub(a, low, high);
{
  if (low = high) return a[low] //base case: only 1 element
  else
  {   mid = (low+high)/2;
      wL = MaxSub(a, low, mid);
      wR = MaxSub(a, mid+1, high);
      wM = MaxCrossMidPoint(a, low, mid, high);
      return max(wL, wR, wM);
  }
}
```

Example Finding the value w_M of the max subarray starting at the left half and ending at the right half

mid =5

On the left half:

	1	2	3	4	5	6	7	8	9	10
a	13	-3	-25	20	-3	-16	-23	18	20	-7

a [5 .. 5] = -3
a [4 .. 5] = 17 ⇐ (max-left = 4)
a [3 .. 5] = -8
a [2 .. 5] = -11
a [1 .. 5] = 2

mid =5

On the right half:

	1	2	3	4	5	6	7	8	9	10
a	13	-3	-25	20	-3	-16	-23	18	20	-7

a[6 .. 6] = -16
a[6 .. 7] = -39
a[6 .. 8] = -21
a[6 .. 9] = (max-right = 9) ⇒ -1
a[6..10] = -8

⇒ maximum subarray crossing *mid* is $a[4..9] = 17 + (-1) = 16$

Finding the value w_M of the max subarray starting at the left half and ending at the right half

- On the left half: find the value w_{ML} of the max subarray ending at the midpoint
- On the right half: find the value w_{MR} of the max subarray starting at the midpoint+1

```
MaxLeft(a, low, mid);
{
  maxSum = -∞; sum = 0;
  for (int k=mid; k>=low; k--) {
       sum = sum+a[k];
       maxSum = max(sum, maxSum);
   }
    return maxSum;
}
```

```
MaxRight(a, mid, high);
{
     maxSum = -∞; sum = 0;
     for (int k=mid; k<=high; k++){
       sum = sum+a[k];
       maxSum = max(sum, maxSum);
   }
   return maxSum;
}
```

- Then $w_M = w_{ML} + w_{MR}$.

```
MaxSub(a, low, high);
{
  if (low = high) return a[low] //base case: only 1 element
  else
  {   mid = (low+high)/2;
      wL = MaxSub(a, low, mid);
      wR = MaxSub(a, mid+1, high);
      wM = MaxCrossMidPoint(a, low, mid, high);
      return max(wL, wR, wM);
  }
}
```

⇒ `wM = MaxLeft(a,low,mid) + MaxRight(a,mid,high);`

1.1.3. Recursive algorithm to solve max subarray problem

Analyzing time complexity:

- Procedure `MaxLeft` and `MaxRight` requires

  $n/2 + n/2 = n$ additions

- Define T(n) the number of additions that the procedure `maxSub(a, 1, n)` need to perform, then we get the following recursion relation:

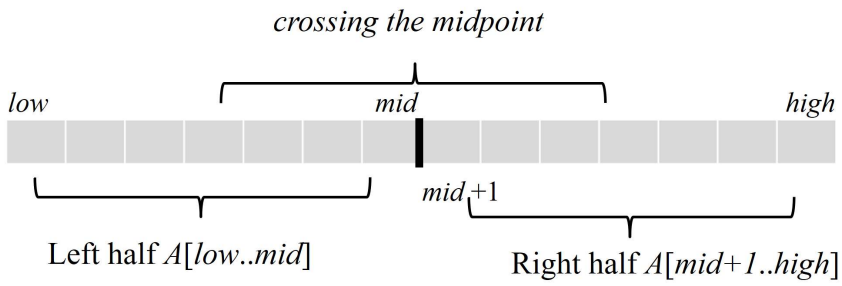
$$T(n) = \begin{cases} 0 & n = 1 \\ T(\frac{n}{2}) + T(\frac{n}{2}) + n = 2T(\frac{n}{2}) + n & n > 1 \end{cases}$$

Solving this recursive relation, we get $T(n) = n \log n$

```
MaxLeft(a, low, mid);
{
  maxSum = -∞; sum = 0;
  for (int k=mid; k>=low; k--) {
       sum = sum+a[k];
       maxSum = max(sum, maxSum);
   }
    return maxSum;
}

MaxRight(a, mid, high);
{
     maxSum = -∞; sum = 0;
    for (int k=mid; k<=high; k++){
      sum = sum+a[k];
      maxSum = max(sum, maxSum);
   }
   return maxSum;
}
```

```
MaxSub(a, low, high);
{
  if (low = high) return a[low] //base case: only 1 element
   else
   {    mid = (low+high)/2;
        wL = MaxSub(a, low, mid);
        wR = MaxSub(a, mid+1, high);
        wM = MaxLeft(a, low, mid) + MaxRight(a, mid, high);
        return max(wL, wR, wM);
   }
}
```



1.1.3. Recursive algorithm to solve max subarray problem

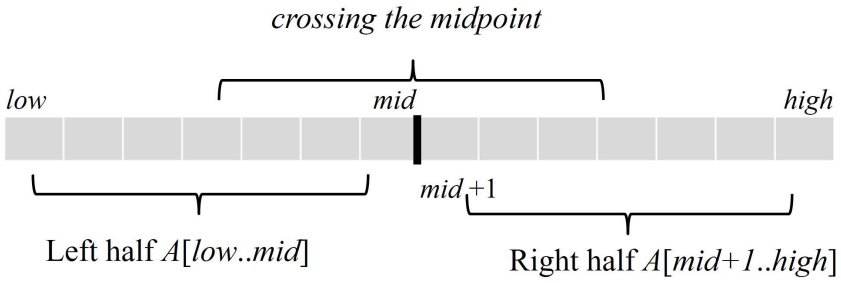
Example: MaxSub(a, 1, 10)

1	2	3	4	5	6	7	8	9	10
13	-3	-25	20	-3	-16	-23	18	20	-7

```
MaxLeft(a, low, mid);
{
  maxSum = -∞; sum = 0;
  for (int k=mid; k>=low; k--) {
       sum = sum+a[k];
       maxSum = max(sum, maxSum);
   }
    return maxSum;
}

MaxRight(a, mid, high);
{
     maxSum = -∞; sum = 0;
    for (int k=mid; k<=high; k++){
       sum = sum+a[k];
       maxSum = max(sum, maxSum);
   }
   return maxSum;
}
```

```
MaxSub(a, low, high);
{
  if (low = high) return a[low] //base case: only 1 element
   else
   {    mid = (low+high)/2;
        wL = MaxSub(a, low, mid);
        wR = MaxSub(a, mid+1, high);
        wM = MaxLeft(a, low, mid) + MaxRight(a, mid, high);
        return max(wL, wR, wM);
   }
}
```



Max subarray problem: compare the time complexity between algorithms

The number of additions that the algorithm need to perform:

1.1.1. Brute force $\frac{n^3}{6}+\frac{n^2}{2}+\frac{n}{3}$

1.1.2. Brute force with better implement $\frac{n^2}{2}+\frac{n}{2}$

1.1.3 Recursive algorithm: $n \log n$

➔ For the same problem (max subarray), we propose 3 algorithms that requires different number of addition operations, and therefore, they will require different computation time.

The following tables show the computation time of these 3 algorithms with the assumption: the computer could do 10^8 addition operation per second

Complexity	n=10	Time (sec)	n=100	Time (sec)	$n=10^4$	Time	$n=10^6$	Time
n^3	10^3	10^{-5}	10^6	10^{-2} sec	**10^{12}**	2.7 hours	10^{18}	115 days
n^2	100	10^{-6}	10000	10^{-4} sec	**10^8**	1 sec	10^{12}	2.7 hours
n log n	33.2	$3.3*10^{-8}$	664	$6.6*10^{-6}$sec	**$1.33*10^5$**	10^{-3} sec	$1.99*10^7$	$2*10^{-1}$ sec
e^n	$2.2*10^4$	$1*10^{-4}$	$2.69*10^{43}$	$>10^{26}$ centuries	**$8.81*10^{4342}$**	$>10^{4327}$ centuries		

Max subarray problem: compare the time complexity between algorithms

Complexity	n=10	Time (sec)	n=100	Time (sec)	$n=10^4$	Time	$n=10^6$	Time
n^3	10^3	10^{-5}	10^6	10^{-2} sec	**10^{12}**	2.7 hours	10^{18}	115 days
n^2	100	10^{-6}	10000	10^{-4} sec	**10^8**	1 sec	10^{12}	2.7 hours
n log n	33.2	$3.3*10^{-8}$	664	$6.6*10^{-6}$sec	**$1.33*10^5$**	10^{-3} sec	$1.99*10^7$	$2*10^{-1}$ sec
e^n	$2.2*10^4$	$1*10^{-4}$	$2.69*10^{43}$	$>10^{26}$ centuries	**$8.81*10^{4342}$**	$>10^{4327}$ centuries		

- With small n, the calculation time is negligible.
- The problem becomes more serious when $n > 10^6$. At that time, only the third algorithm is applicable in real time.
- Can we do better?

Yes! It is possible to propose an algorithm that requires only n additions!

1. Introductory example: the max subarray problem

1.1.1. Brute force

$$\frac{n^3}{6}+\frac{n^2}{2}+\frac{n}{3}$$

1.1.2. Brute force with better implement

$$\frac{n^2}{2}+\frac{n}{2}$$

1.1.3 Recursive algorithm

$n \log n$

1.1.4. Dynamic programming

n

1.1.4. Dynamic programming to solve max subarray problem

The primary steps of dynamic programming:

1. Divide: Partition the given problem into subproblems

(Subproblem: have the same structure as the given problem but with smaller size)

2. Note the solution: store the solutions of subproblems in a table

3. Construct the final solution: from the solutions of smaller size problems, try to find the way to construct the solutions of the larger size problems until get the solution of **the given problem (the subproblem with largest size)**

1.1.4. Dynamic programming to solve max subarray problem

The primary steps of dynamic programming:

1. Divide:

- Define s_i the value of max subarray of the array $a_1, a_2, ..., a_i$, $i = 1, 2, ..., n$.
- Clearly, s_n is the solution.

3. Construct the final solution:

- $s_1 = a_1$
- Assume $i > 1$ and we already know the value of s_k with $k = 1, 2, ..., i\text{-}1$. Now we need to calculate the value of s_i which is the value of max subarray of the array:

  $a_1, a_2, ..., a_{i-1}, a_i$.
- We see that: the max subarray of this array $a_1, a_2, ..., a_{i-1}, a_i$ could either include the element a_i or not include the element a_i ➔ therefore, the max subarray of the array $a_1, a_2, ..., a_{i-1}, a_i$ could only be one of these 2 arrays:
  - The max subarray of the array $a_1, a_2, ..., a_{i-1}$
  - The max subarray of the array $a_1, a_2, ..., a_i$ *ending at* a_i.

➔ Thus, we have $s_i = max\ \{s_{i-1}, e_i\}$, $i = 2, ..., n$.

where e_i is the value of the max subarray $a_1, a_2, ..., a_i$ ending at a_i.

To calculate e_i, *we could use the recursive relation*:

- $e_1 = a_1$;
- $e_i = \max \{a_i, e_{i-1} + a_i\}$, $i = 2, ..., n$.

1.1.4. Dynamic programming to solve max subarray problem

The primary steps of dynamic programming:

1. Divide:

- Define s_i the value of max subarray of the array $a_1, a_2, ..., a_i$, $i = 1, 2, ..., n$.
- Clearly, s_n is the solution.

3. Construct the final solution:

- $s_1 = a_1$
- Assume $i > 1$ and we already know the value of s_k with $k = 1, 2, ..., i\text{-}1$. Now we need to calculate the value of s_i which is the value of max subarray of the array:

$$a_1, a_2, ..., a_{i-1}, a_i .$$

- We see that: the max subarray of this array $a_1, a_2, ..., a_{i-1}, a_i$ could either include the element a_i or not include the element a_i ➔ therefore, the max subarray of the array $a_1, a_2, ..., a_{i-1}, a_i$ could only be one of these 2 arrays:
  - The max subarray of the array $a_1, a_2, ..., a_{i-1}$
  - The max subarray of the array $a_1, a_2, ..., a_i$ *ending at* a_i.

➔ Thus, we have $s_i = max\ \{s_{i-1}, e_i\}$, $i = 2, ..., n$.

where e_i is the value of the max subarray $a_1, a_2, ..., a_i$ ending at a_i.

To calculate e_i, *we could use the recursive relation*:

- $e_1 = a_1$;
- $e_i = \max \{a_i, e_{i-1} + a_i\}$, $i = 2, ..., n$.

```
MaxSub(a)
{
    smax  = a[1];        (* smax – the value of max subarray  *)
    ei    = a[1];        (* ei    – the value of max subarray ending at a[i] *)
    for i = 2 to n {
          ei = max { a[i], ei + a[i] }
          smax = max { smax, ei }
    }
}
```

1.1.4. Dynamic programming to solve max subarray problem

```
MaxSub(a)
{
    smax = a[1];          (* smax – the value of max subarray  *)
    ei   = a[1];          (* ei   – the value of max subarray ending at a[i] *)
    imax = 1;             (* imax – the index of the last element of the max sub array *)
    for i = 2 to n {
        u = ei + a[i];
        v = a[i];
        if (u > v) ei = u
         else ei = v;
        if (ei > smax) {
            smax := ei;
            imax := i;
        }
    }
}
```

```
MaxSub(a)
{
    smax = a[1];          (* smax – the value of max subarray  *)
    ei   = a[1];          (* ei   – the value of max subarray ending at a[i] *)
    for i = 2 to n {
        ei = max { a[i], ei + a[i] }
        smax = max { smax, ei }
    }
}
```

Analyzing time complexity:

the number of addition operations need to be performed in the algorithm

= the number of times the statement u = ei + a[i]; need to be executed

= n

Comparison of 4 algorithms

- The following table shows the estimated running time of the four proposed algorithms above (assuming the computer could perform 10^8 addition operations per second).

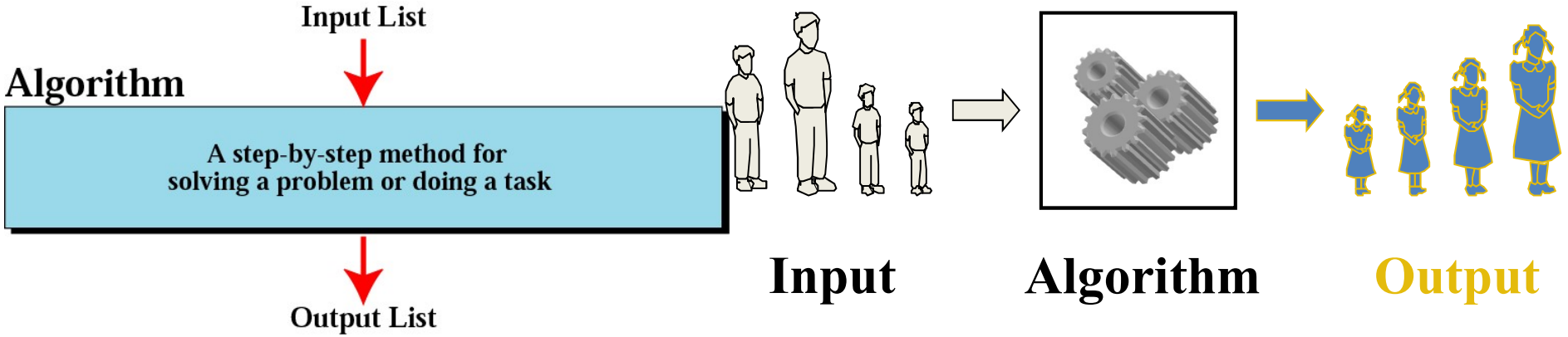
Algorithm	Complexity	$n=10^4$	time	$n=10^6$	time
Brute force	n^3	10^{12}	2.7 hours	10^{18}	115 days
Brute force with better implementation	n^2	10^8	1 sec	10^{12}	2.7 hours
Recursive	n log n	$1.33*10^5$	10^{-3} sec	$1.99*10^7$	$2*10^{-1}$ sec
Dynamic programming	n	10^4	10^{-4} sec	10^6	$2*10^{-2}$ sec

This example shows how the development of effective algorithms could significantly reduce the cost of running time.

Contents

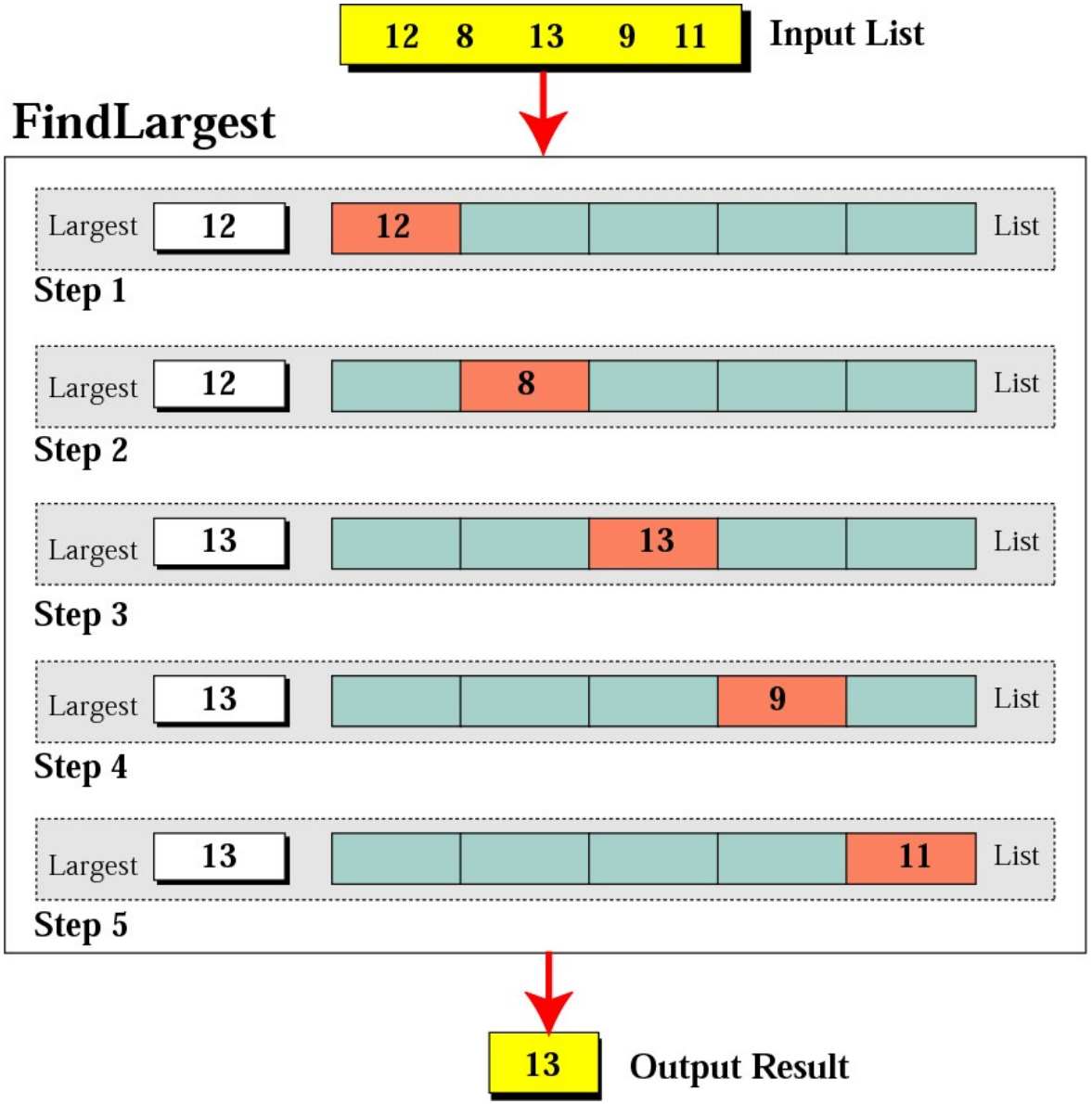
Algorithm

- The word *algorithm* comes from the name of a Persian mathematician Abu Ja'far Mohammed ibn-i Musa al Khowarizmi.
- In computer science, this word refers to a special method consisting of a sequence of unambiguous instructions useable by a computer for solution of a problem.
- Informal definition of an algorithm in a computer:

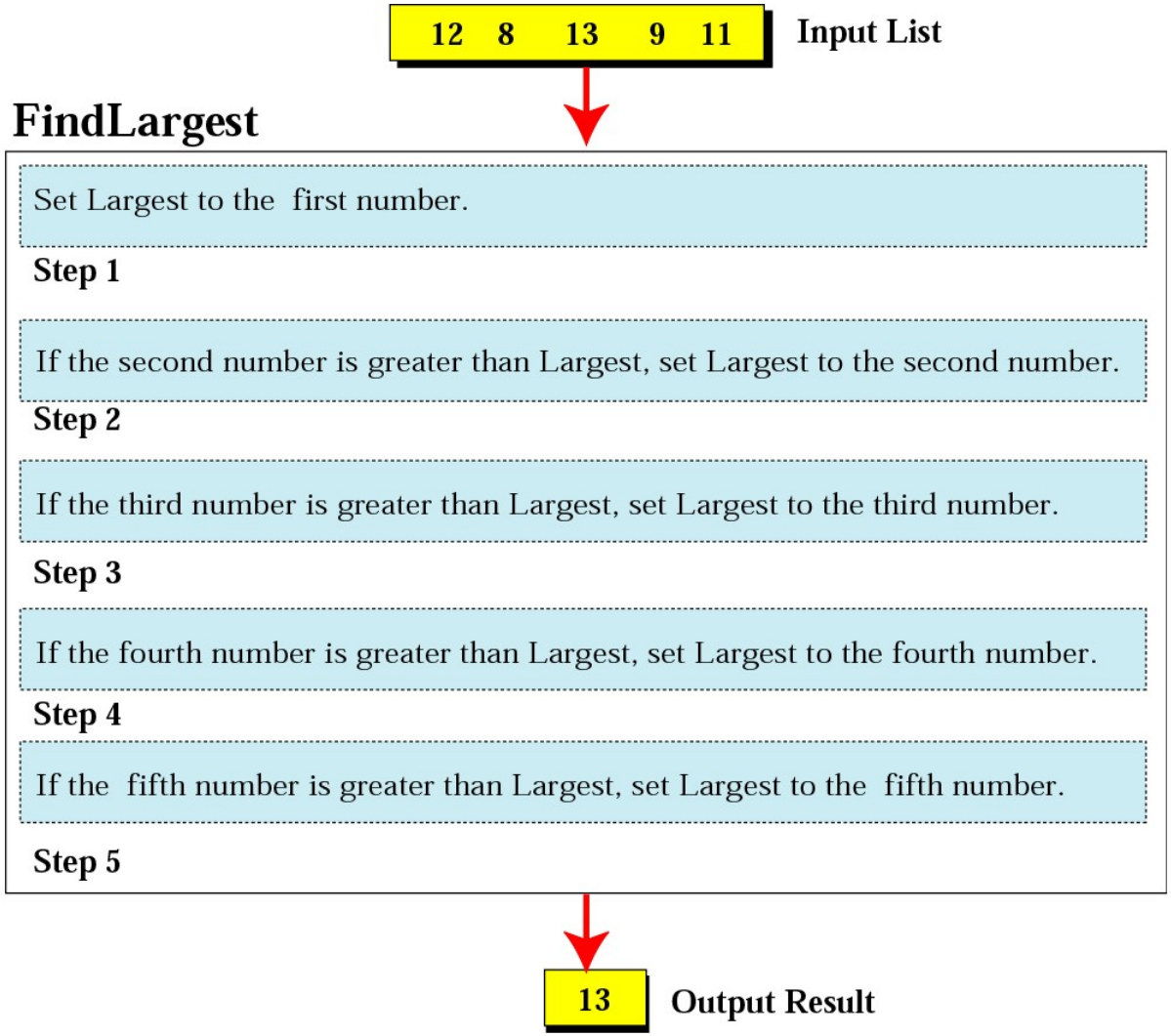


- Example: The problem of finding the largest integer among a number of positive integers
  - Input: the array of n positive integers $a_1, a_2, \ldots, a_n$
  - Output: the largest
  - Example: Input 12 8 13 9 11 ➔ Output: 12
  - Question: Design the algorithm to solve this problem

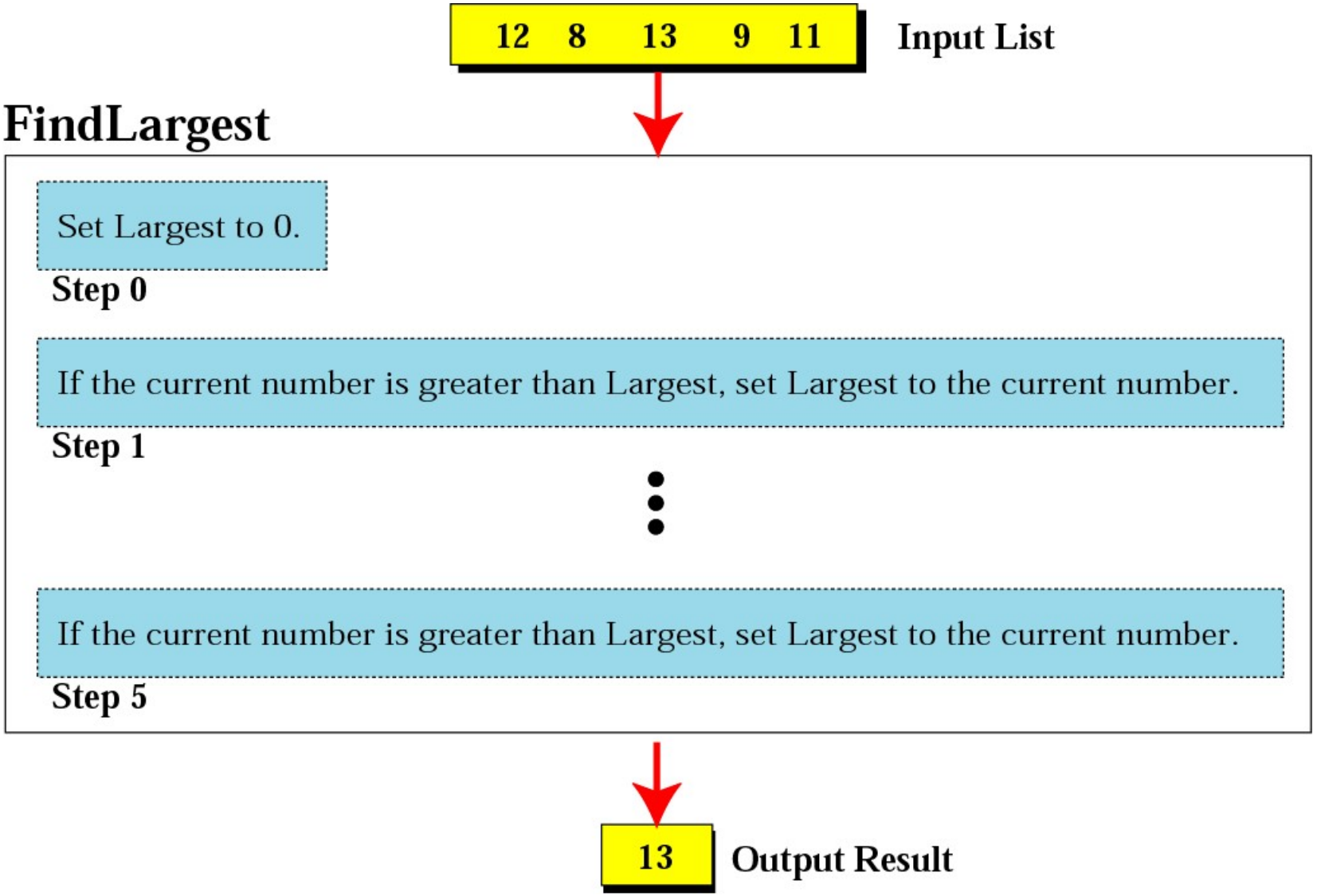
Example: Finding the largest integer among five integers



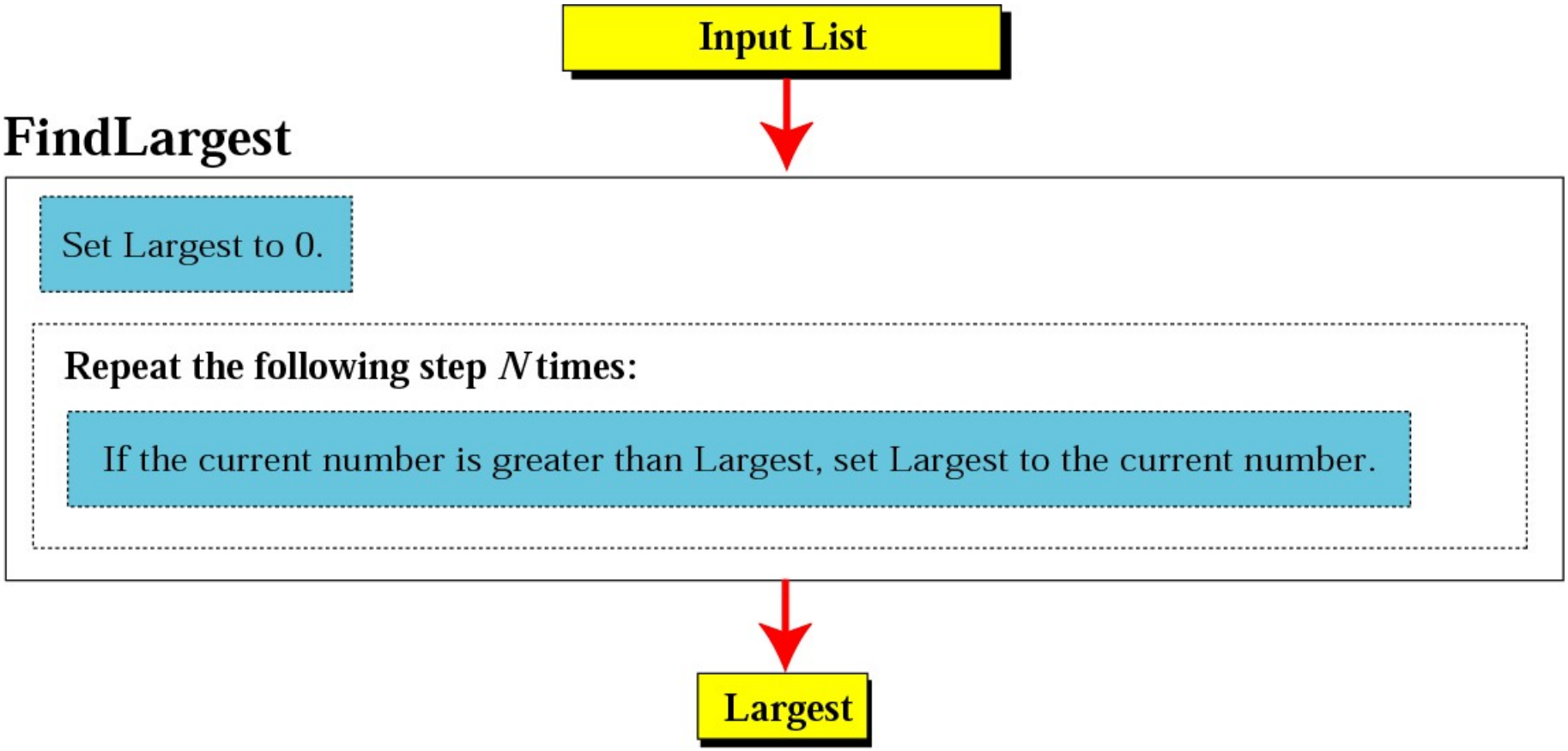
Defining actions in FindLargest algorithm



FindLargest refined



Generalization of FindLargest



Algorithm

- All algorithms must satisfy the following criteria:

  (1) **Input**. The algorithm receives data from a certain set.

  (2) **Output**. For each set of input data, the algorithm gives the solution to the problem.

  (3) **Precision**. Each instruction is clear and unambiguous.

  (4) **Finiteness**. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite (possibly very large) number of steps.

  (5) **Uniqueness**. The intermediate results of each step of the algorithm are uniquely determined and depend only on the input and the result of the previous steps.

  (6) **Generality.** The algorithm could be applied to solve any problem with a given form

Example: Translating a Problem into an Algorithm

- Problem: Devise a program that sorts a set of $n >= 1$ integers
  - Input: A set of n integers: a[1], a[2],…, a[n]
  - Output: A set of n integers in ascending order: a[1] $\leq a[2] \leq .. \leq a[n]$
- Step I - Concept
  - From those integers that are currently unsorted, find the smallest and place it next in the sorted list
- Step II - Algorithm

  for (i = 1; i <= n; i++){
    - Examine a[i] to a[n] to find the smallest one, and suppose that the smallest integer is a[index_*min*] where $i \leq$ index_min $\leq n$;
    - Swap a[i] and a[index_*min*];

  }
- Step III - Coding

```
void sort(int *a, int n)
{
   for (i= 1; i < n; i++)
   {
       int index_min= i;
       for (int k= i+1; k<= n; k++)
            if (a[k ] < a[index_min]) index_min= k;
        int temp=a[i]; a[i]=a[index_min]; a[index_min]=temp;
   }
}
```

Example: An array of 6 integers

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
9	6	-7	8	10	3

Comparing Algorithms

- Given 2 or more algorithms to solve the same problem, how do we select the best one?
- Some criteria for selecting an algorithm:
  1) Is it easy to implement, understand, modify?
  2) How long does it take to run it to completion? **TIME**
  3) How much of computer memory does it use? **SPACE**

In this lecture we are interested in the second and third criteria:

- **Time complexity**: The amount of time that an algorithm needs to run to completion
- **Space complexity**: The amount of memory an algorithm needs to run

We will occasionally look at space complexity, but we are mostly interested in time complexity in this course. Thus in this course the better algorithm is the one which runs faster (has smaller time complexity)

How to Calculate Running time

- Most algorithms transform input objects into output objects

- The running time of an algorithm typically grows with the input size
  - Idea: analyze running time as a function of input size
  - Even on inputs of the same size, running time can be very different
    - Example: algorithm that finds the first prime number in an array by scanning it left to right
      - Array 1: 3 9 8 12 15 20
      - Array 2: 9 8 3 12 15 20
      - Array 3: 9 8 12 15 20 3

➔ Idea: analyze running time in the

- best case
- worst case
- average case

Kind of analyses

Best-case:

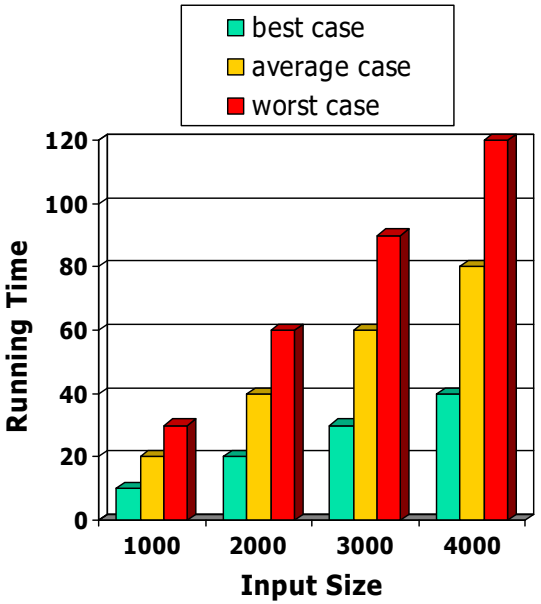
- Cheat with a slow algorithm that works fast on some input.

Average-case: (sometimes)

- T(n) = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs
- Very useful but often difficult to determine

Worst-case: (usually)

- T(n) = maximum time of algorithm on any input of size n.
- Easier to analyze



To evaluate the running time: 2 ways:

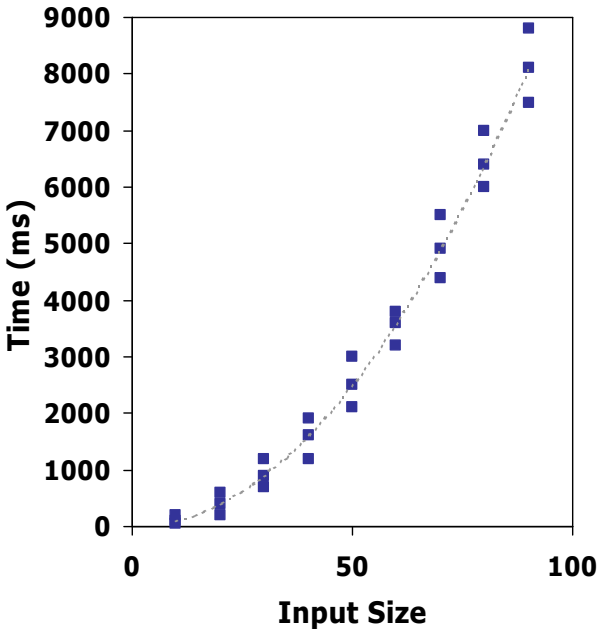
- Experimental evaluation of running time
- Theoretical analysis of running time

Experimental Evaluation of Running Time

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like clock( ) to get an accurate measure of the actual running time

```
clock_t startTime = clock();
doSomeOperation();
clock_t endTime = clock();
clock_t clockTicksTaken = endTime - startTime;
double timeInSeconds = clockTicksTaken / (double) CLOCKS_PER_SEC;
```

- Plot the results



Limitations of Experiments when evaluating the running time of an algorithm

- Experimental evaluation of running time is very useful but
  - It is necessary to implement the algorithm, which may be difficult
  - Results may not be indicative of the running time on other inputs not included in the experiment
  - In order to compare two algorithms, the same hardware and software environments must be used

➔ We need: **Theoretical Analysis of Running Time**

Theoretical Analysis of Running Time

- Uses a pseudo-code description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, ***n***
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment (Changing the hardware/software environment affects the running time by a constant factor, but does not alter the growth rate of the running time)

Contents

1.3. Asymptotic notation

Θ, Ω, O, **o**, ω

» What these symbols do are:

- give us a notation for talking about how fast a function goes to infinity, which is just what we want to know when we study the running times of algorithms.
- defined for functions over the natural numbers
- used to compare the order of growth of 2 functions

Example: $f(n) = \Theta(n^2)$: Describes how $f(n)$ grows in comparison to n^2.

» Instead of working out a complicated formula for the exact running time, we can just say that the running time is for example $\Theta(n^2)$ [read as theta of n^2]: that is, the running time is proportional to n^2 plus lower order terms. For most purposes, that's just what we want to know.

Θ - Theta notation

- For a given function $g(n)$, we denote by $\boldsymbol{\Theta}(g(n))$ the set of functions

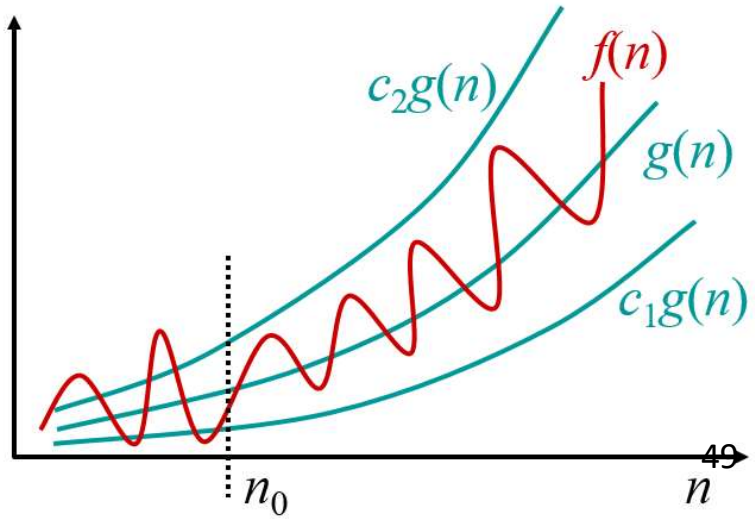
$$\Theta(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c_1, c_2, \text{and } n_0 \text{ s.t.} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.

- A function $f(n)$ belongs to the set $\boldsymbol{\Theta}(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sand- wiched" between $c_1 g(n)$ and $c_2 g(n)$ for sufficiently large n
  - $f(n) = \boldsymbol{\Theta}(g(n))$ means that there exists some constant c_1 and c_2 s.t.

$$c_1 g(n) \le f(n) \le c_2 g(n) \text{ for large enough } n.$$

- When we say that one function is theta of another, we mean that neither function goes to infinity faster than the other.



$f(n) = \Theta(g(n)) \iff \exists c_1, c_2, n_0 > 0 : \forall n \geq n_0, \; c_1 g(n) \leq f(n) \leq c_2 g(n)$

Example 1: Show that $10n^2 - 3n = \mathbf{\Theta}(n^2)$

- With which values of the constants n_0, c_1, c_2 then the inequality in the definition of the theta notation is correct:

$$c_1 n^2 \leq f(n) = 10n^2 - 3n \leq c_2 n^2 \; \forall n \geq n_0$$

- Suggestion: Make c_1 a little smaller than the leading (the highest) coefficient, and c_2 a little bigger.

➔ Select: $c_1 = 1$, $c_2 = 11$, $n_0 = 1$ then we have

$$n^2 \leq 10n^2 - 3n \leq 11n^2, \; with \; n \geq 1.$$

➔ $\forall n \geq 1$: $10n^2 - 3n = \mathbf{\Theta}(n^2)$

- Note: For polynomial functions: To compare the growth rate, it is necessary to look at the term with the highest coefficient

$f(n) = \Theta(g(n))$ $\iff$ $\exists c_1, c_2, n_0 > 0 : \forall n \geq n_0, \; c_1 g(n) \leq f(n) \leq c_2 g(n)$

Example 2: Show that $f(n) = \frac{1}{2}n^2 - 3n = \boldsymbol{\Theta}(n^2)$

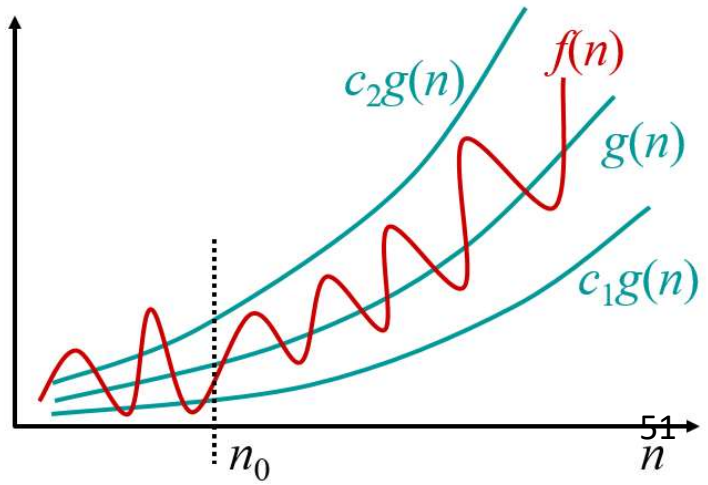
We must find n_0, c_1 and c_2 such that

$$c_1 n^2 \leq f(n) = \frac{1}{2}n^2 - 3n \leq c_2 n^2 \; \forall n \geq n_0$$

Select $c_1 = ¼$, $c_2 = 1$, and $n_0 = 7$ we have:

$$\frac{1}{4}n^2 \leq f(n) = \frac{1}{2}n^2 - 3n \leq n^2 \quad \forall\, n \geq 7$$

➔$\forall\, n \geq 7$: $\frac{1}{2}n^2 - 3n = \boldsymbol{\Theta}(n^2)$

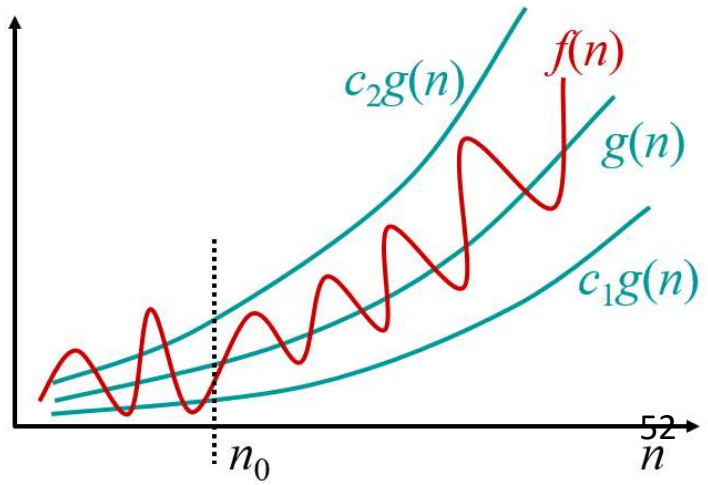


$f(n) = \Theta(g(n))$ $\iff$ $\exists c_1, c_2, n_0 > 0 \ : \ \forall n \geq n_0, \ c_1 g(n) \leq f(n) \leq c_2 g(n)$

Example 3: Show that $f(n) = 23n^3 - 10\, n^2 \log_2 n + 7n + 6 = \boldsymbol{\Theta}(n^3)$

We must find n_0, c_1 and c_2 such that

$$c_1 n^3 \leq f(n) = 23n^3 - 10\, n^2 \log_2 n + 7n + 6 \leq c_2 n^3 \forall n \geq n_0$$



$f(n) = \Theta(g(n))$ $\Longleftrightarrow$ $\exists c_1, c_2, n_0 > 0 \ : \ \forall n \geq n_0, \ c_1 g(n) \leq f(n) \leq c_2 g(n)$

Example 3: Show that $f(n) = 23n^3 - 10\, n^2 \log_2 n + 7n + 6 = \Theta(n^3)$

$f(n) = 23n^3 - 10\, n^2 \log_2 n + 7n + 6$

➔ $f(n) = [23 - (10 \log_2 n)/n + 7/n^2 + 6/n^3]n^3$

Then:

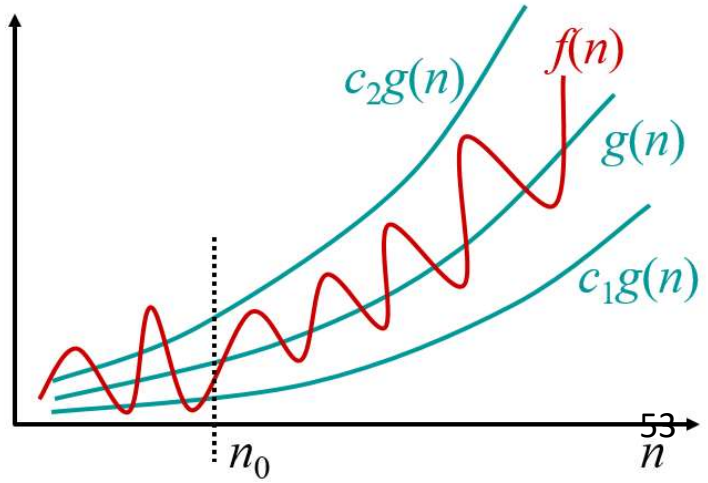
- $\forall n \geq 10 : \ f(n) \leq (23 + 0 + 7/100 + 6/1000)n^3$
  $= (23 + 0 + 0.07 + 0.006)n^3$
  $= 23.076\, n^3 < 24\, n^3$
- $\forall n \geq 10 : \ f(n) \geq (23 - \log_2 10 + 0 + 0)n^3 > (23 - \log_2 16)n^3 = 19n^3$

➔ We have:

$\forall n \geq 10: \ 19\, n^3 \leq f(n) \leq 24\, n^3$

$(n_0 = 10, \ c_1 = 19, \ c_2 = 24, \ g(n) = n^3)$

Therefore: $f(n) = \Theta(n^3)$



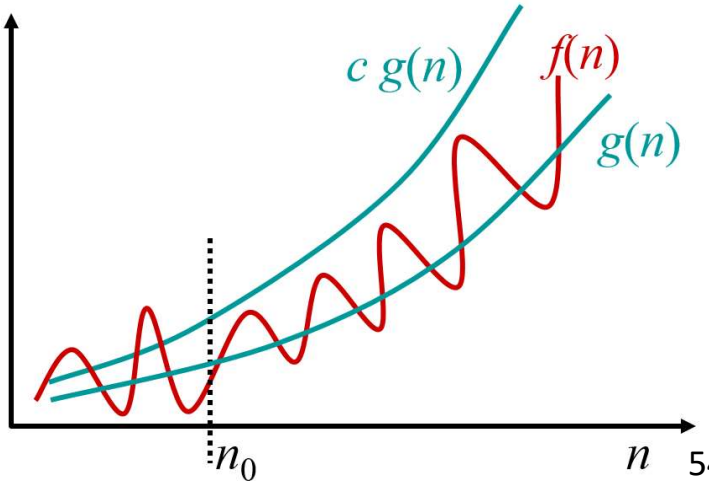
O - big Oh notation

For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

- We say: $g(n)$ is asymptotic upper bound of the function $f(n)$, to within a constant factor, and write $f(n) = O(g(n))$.
- $f(n) = O(g(n))$ means that there exists some constant c such that $f(n)$ is always $\le cg(n)$ for large enough n.
- $O(g(n))$ is the set of functions that go to infinity no faster than $g(n)$.



Graphic Illustration

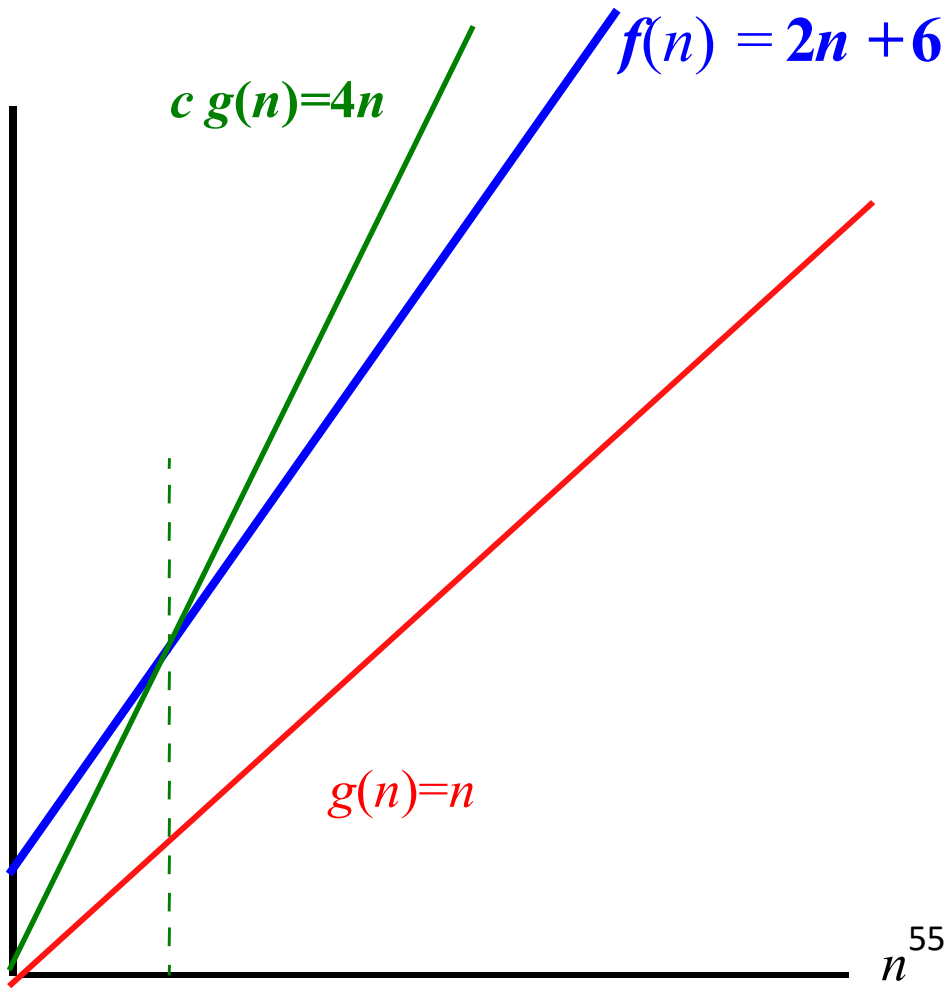
$O(g(n))$ = {$f(n)$: ∃ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)$ }

- $f(n) = 2n+6$
- Conf. def:
  - Need to find a function $g(n)$ and constants c and n_0 such as $f(n) < cg(n)$ when $n > n_0$

➔ $g(n) = n$, $c = 4$ and $n_0 = 3$

→ $f(n)$ is $O(n)$

The order of $f(n)$ is n



Big-Oh Examples

$O(g(n))$ = $\{f(n)$: $\exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)\}$

- Example 1: Show that $2n + 10 = O(n)$

➔ $f(n) = 2n+10$, $g(n) = n$

  - Need constants c and n_0 such that $2n + 10 \leq cn$ for $n \geq n_0$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$

- Example 2: Show that $7n$-2 is $O(n)$

➔ $f(n) = 7n$-2, $g(n) = n$

  - Need constants c and n_0 such that $7n - 2 \leq cn$ for $n \geq n_0$
  - $(7 - c\,)\, n \leq 2$
  - $n \leq 2/(7 - c)$
  - Pick $c = 7$ and $n_0 = 1$

Note

- The values of positive constants n_0 and c **are not unique** when proof the asymptotic formulas
- Example: show that $100n + 5 = O(n^2)$
  - $100n + 5 \leq 100n + n = 101n \leq 101n^2 \qquad \forall\ n \geq 5$

    $n_0 = 5$ and c = 101 are constants need to determine
  - $100n + 5 \leq 100n + 5n = 105n \leq 105n^2\ \forall\ n \geq 1$

    $n_0 = 1$ and $c = 105$ are also constants need to determine
- Only need to find **some** positive constants c and n_0 satisfying the equality in the definition of asymptotic notation

Big-Oh Examples

$O(g(n))$ = $\{f(n)$: $\exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)\}$

- Example 3: Show that $3n^3 + 20n^2 + 5$ is $O(n^3)$

  Need constants c and n_0 such that $3n^3 + 20n^2 + 5 \leq \boldsymbol{cn^3}$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

- Example 4: Show that $3 \log n + 5$ is $O(\log n)$

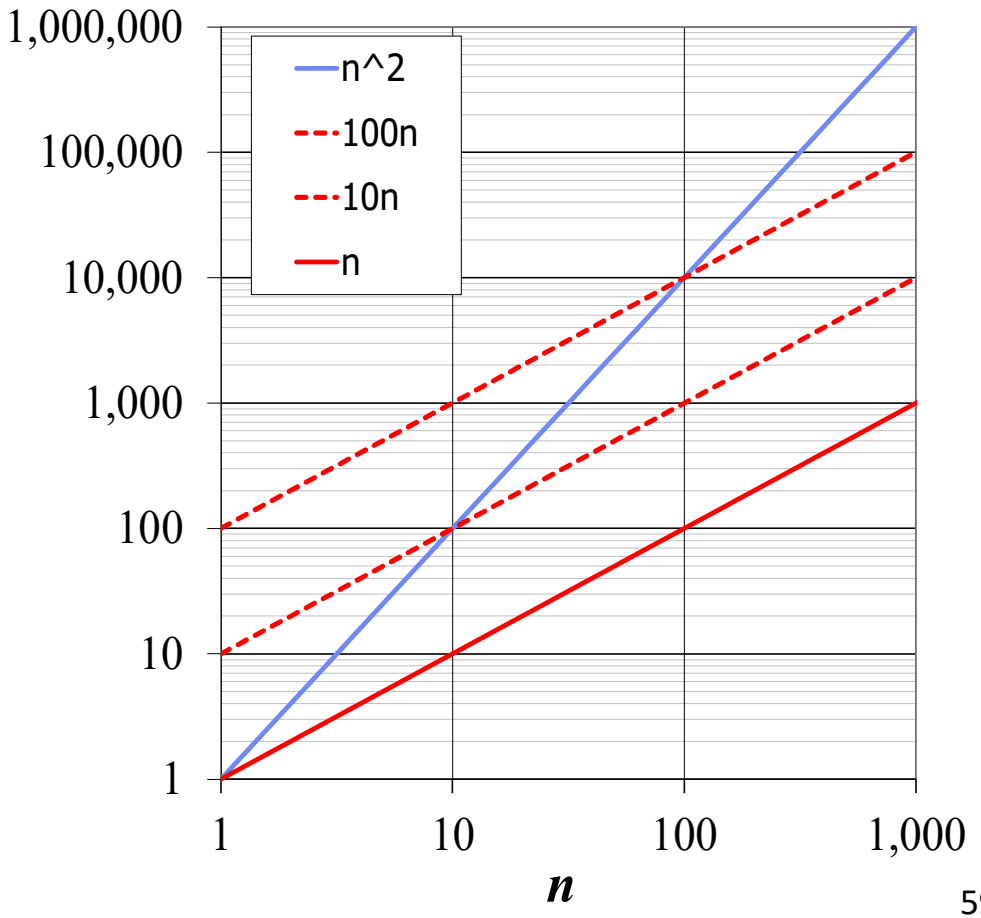
  Need constants c and n_0 such that $3 \log n + 5 \leq c \log n$ for $n \geq n_0$

  this is true for $c = 8$ and $n_0 = 2$

Big-Oh Examples

$O(g(n)) = \{f(n) : \exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)\}$

- Example 5: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant



Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more		
$f(n)$ grows more		
Same growth		

Inappropriate Expressions

$$f(n) \leq O(g(n))$$

$$f(n) \geq O(g(n))$$

Big-Oh Examples

- $50n^3 + 20n + 4$ is $O(n^3)$
  - Would be correct to say is $O(n^3+n)$
    - Not useful, as n^3 exceeds by far n, for large values
  - Would be correct to say is $O(n^5)$
    - OK, but $g(n)$ should be as close as possible to $f(n)$
- $3\log(n) + \log(\log(n)) = O(\ ?\ )$

•Simple Rule: Drop lower order terms and constant factors

Useful Big-Oh Rules

- If $f(n)$ is a polynomial of degree d: $f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$

then $f(n)$ is $O(n^d)$, i.e.,

1. Drop lower-order terms
2. Drop constant factors

Example: $3n^3 + 20n^2 + 5$ is $O(n^3)$

- If $f(n) = O(n^k)$ then $f(n) = O(n^p)$ with $\forall\ p > k$

Example: $2n^2 = O(n^2)$ then $2n^2 = O(n^3)$

When evaluate asymptotic $f(n) = O(g(n))$, we want to find function $g(n)$ with a slower growth rate as possible

- Use the smallest possible class of functions

Example: Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"

- Use the simplest expression of the class

Example: Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

O Notation Examples

- All these expressions are O(n):
  - n, $3n$, $61n + 5$, $22n – 5$, …
- All these expressions are O(n^2):
  - n^2, $9\ n^2$, $18\ n^2 + 4n – 53$, …
- All these expressions are O($n \log n$):
  - $n(\log n)$, $5n(\log 99n)$, $18 + (4n – 2)(\log (5n + 3))$, …

Properties

- If $f(n)$ is $O(g(n))$ then $af(n)$ is $O(g(n))$ for any a
- If $f(n)$ is $O(g_1(n))$ and $h(n)$ is $O(g_2(n))$ then
  - $f(n)+h(n)$ is $O(g_1(n)+g_2(n))$
  - $f(n)h(n)$ is $O(g_1(n)\ g_2(n))$
- If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$ then $f(n)$ is $O(h(n))$
- If $p(n)$ is a polynomial in n then $\log p(n)$ is $O(\log(n))$
- If $p(n)$ is a polynomial of degree d, then $p(n)$ is $O(n^d)$
- $n^x = O(a^n)$, for any fixed $x > 0$ and $a > 1$
  - An algorithm of order n to a certain power is better than an algorithm of order a (> 1) to the power of n
- $\log n^x$ is $O(\log n)$, for $x > 0$ – how?
- $\log^x n$ is $O(n^y)$ for $x > 0$ and $y > 0$
  - An algorithm of order $\log n$ (to a certain power) is better than an algorithm of n raised to a power y.

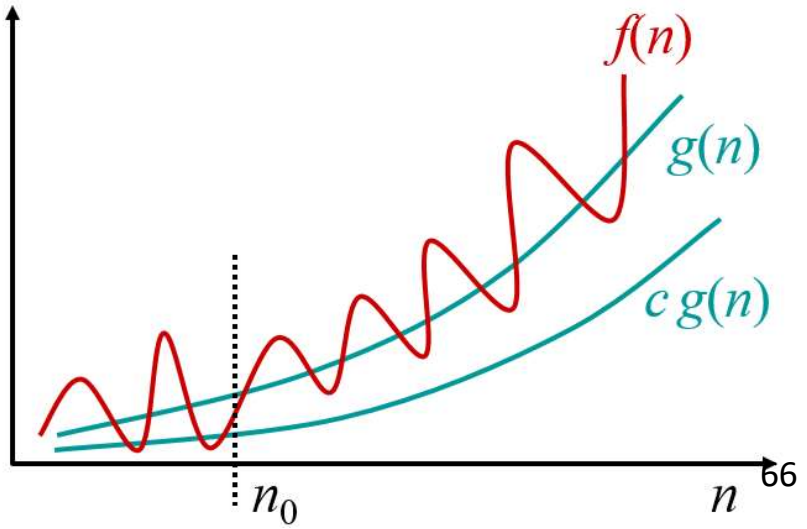
Ω-Omega notation

- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions

$$\Omega(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

- We say: $g(n)$ is asymptotic lower bound of the function $f(n)$, to within a constant factor, and write $f(n) = \Omega(g(n))$.
- $f(n) = \Omega(g(n))$ means that there exists some constant c such that $f(n)$ is always $\ge cg(n)$ for large enough n.
- $\Omega(g(n))$ is the set of functions that go to infinity no slower than $g(n)$



$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$
$\Theta(g(n)) \subset \Omega(g(n))$.

Omega Examples

$\Omega(g(n)) = \{f(n) : \exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq cg(n) \leq f(n)\}$

- Example 1: Show that $5n^2$ is $\Omega(n)$

  Need constants c and n_0 such that $\boldsymbol{cn} \leq 5n^2$ for $n \geq n_0$

  this is true for $c = 1$ and $n_0 = 1$

  Comment:

  - If $f(n) = \Omega(n^k)$ then $f(n) = \Omega(n^p)$ with $\forall\ p < k$.
  - When evaluate asymptotic $f(n) = \Omega(g(n))$, we want to find function $g(n)$ with a faster growth rate as possible

- Example 2: Show that $\sqrt{n} = \Omega(\lg n)$

Asymptotic notation in equations

Another way we use asymptotic notation is to simplify calculations:

- Use asymptotic notation in equations to replace expressions containing lower-order terms

Example:

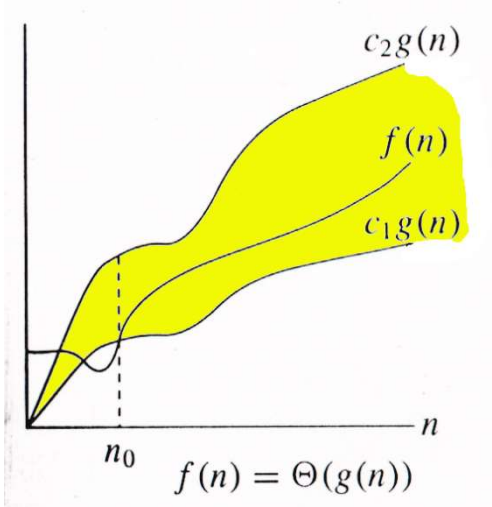
$$4n^3 + 3n^2 + 2n + 1 = 4n^3 + 3n^2 + \Theta(n)$$
$$= 4n^3 + \Theta(n^2) = \Theta(n^3)$$

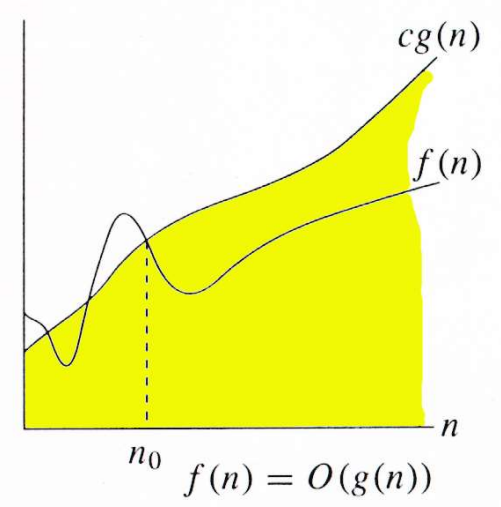
How to interpret?

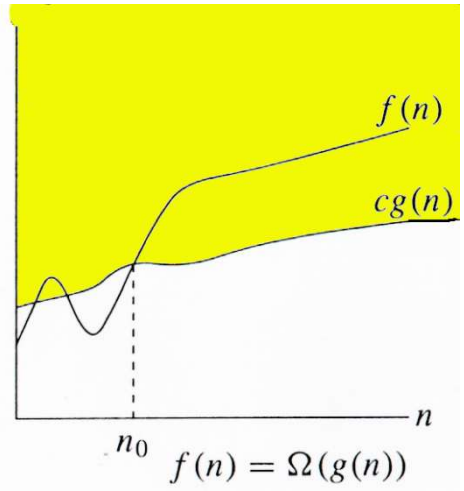
In equations, $\Theta(f(n))$ always stands for an ***anonymous function*** $g(n) \in \Theta(f(n))$

– In this example, we use $\Theta(n^2)$ stands for $3n^2 + 2n + 1$

Asymptotic notation







Graphic examples of $\boldsymbol{\Theta}$, O, and $\boldsymbol{\Omega}$

Theorem: For any two functions $f(n)$ and $g(n)$, we have $f(n) = \boldsymbol{\Theta}(g(n))$ if and only if

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

Theorem: For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Example 1: Show that $f(n) = 5n^2 = \Theta(n^2)$

Because:

- $5n^2 = O(n^2)$

  $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for $n \geq n_0$

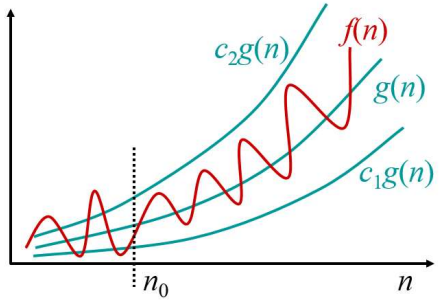
  let $c = 5$ and $n_0 = 1$

- $5n^2 = \Omega(n^2)$

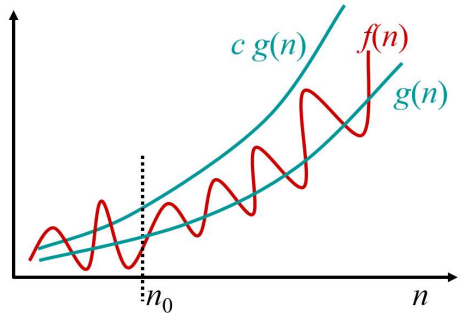
  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq cg(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

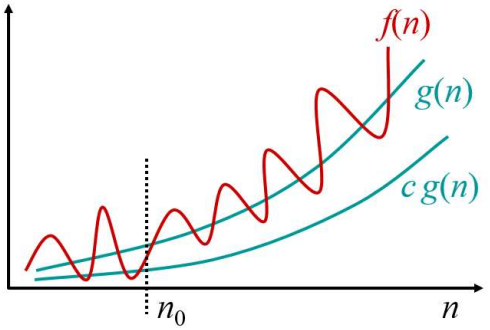
Therefore: $f(n) = \Theta(n^2)$



$f(n) = \Theta(g(n))$



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$

Theorem: For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Example 2: Show that $f(n) = 3n^2 - 2n + 5 = \Theta(n^2)$

Because:

$3n^2 - 2n + 5 = O(n^2)$

$f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for $n \geq n_0$

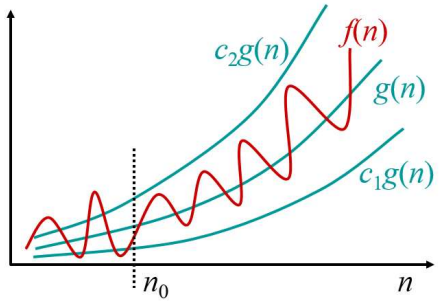
➔ pick c = ? and n_0 = ?

$3n^2 - 2n + 5 = \Omega(n^2)$

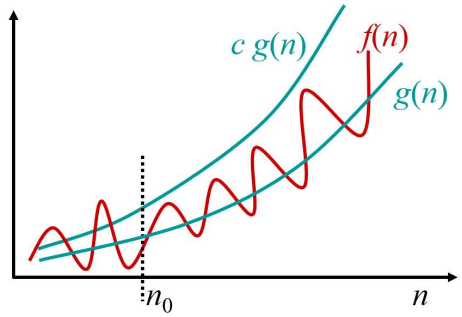
$f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq cg(n)$ for $n \geq n_0$

➔ pick c = ? and n_0 = ?

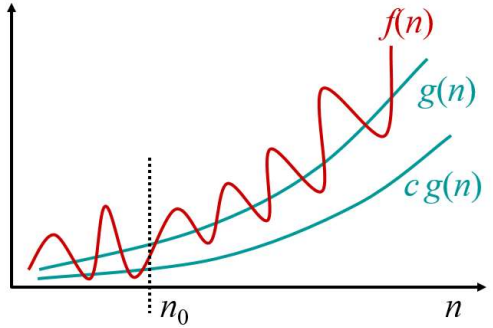
Therefore: $f(n) = \Theta(n^2)$



$f(n) = \Theta(g(n))$



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$

Exercise 1

Show that: $100n + 5 \neq \Omega(n^2)$

Ans: Contradiction

- Assume: $100n + 5 = \Omega(n^2)$

➔ $\exists\ c, n_0$ such that: $0 \leq cn^2 \leq 100n + 5$

- We have: $100n + 5 \leq 100n + 5n = 105n \quad \forall\ n \geq 1$
- Therefore: $cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$
- As $n > 0 \Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$

The above inequality cannot be satisfied since ***c*** must be a constant

Exercise 2

Show that: $n \neq \Theta(n^2)$

Ans: Contradiction

- Assume: $n = \Theta(n^2)$

➔

Exercise 3:Show that

a) $6n^3 \neq \Theta(n^2)$

Ans: Contradiction

- Assume: $6n^3 = \Theta(n^2)$

b) $n \neq \Theta(\log_2 n)$

Ans: Contradiction

- Assume: $n = \Theta(\log_2 n)$

Worst Case Analysis

- When to analyse an algorithm
  - Considering the cases only take
    - maximum amount of time

    (calculate the upper bound on running time of the algorithm)
  - If the algorithm is capable of solving cases in *f(n)*
    - then the worst case should not be greater than *c*f(n)*
- Useful if the algorithm is to be applied to cases
    - the upper bound of an algorithm must be known
- Example:
  - Response time for a nuclear power plant.

Average Time Analysis

- If the algorithm is going to be used many times
  - it is useful to know the average execution time on instances of size n
- It is harder to analyse the average case
  - the distribution of data should be known

Example: Insertion sorting average time is in the order of n^2

Best Case Analysis

- When to analyse an algorithm
  - Considering the cases only take
    - minimum amount of time

    (calculate the lower bound on running time of the algorithm)

The way to talk about the running time

- When people say "The running time for this algorithm is $O(f(n))$", it means that **the worst case running time is $O(f(n))$** (that is, no worse than $c*f(n)$ for large n, since big Oh notation gives an upper bound).
  - It means the worst case running time could be determined by some function $g(n) \in O(f(n))$
- When people say "The running time for this algorithm is $\Omega(f(n))$", it means that **the best case running time is $\Omega(f(n))$** (that is, no better than $c*f(n)$ for large n, since big Omega notation gives a lower bound).
  - It means the best case running time could be determined by some function $g(n) \in \Omega(f(n))$

o- Little oh notation

- For a given function $g(n)$, we denote by $o(g(n))$ the set of functions

$$o(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ 0 \le f(n) < cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

$f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity:

$$\lim_{n \to \infty} [f(n) / g(n)] = 0$$

$g(n)$ is an ***upper bound*** for $f(n)$ that is not asymptotically tight.

ω - Little omega notation

- For a given function $g(n)$, we denote by $o(g(n))$ the set of functions

$$\omega(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ 0 \le cg(n) < f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

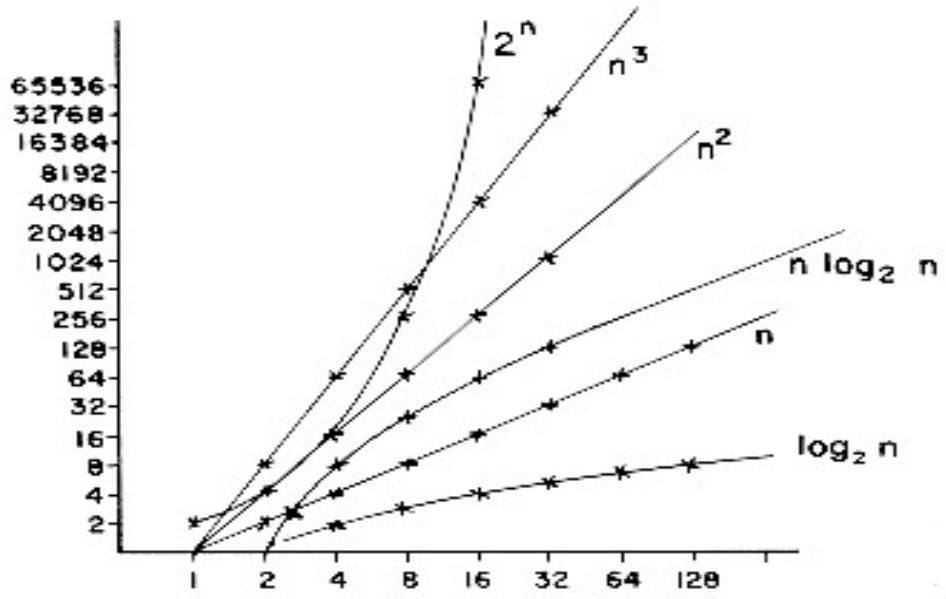
$f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity:

$$\lim_{n \to \infty} [f(n) / g(n)] = \infty$$

$g(n)$ is a ***lower bound*** for $f(n)$ that is not asymptotically tight.

Basic functions

- Often appear in algorithm analysis:
  - Constant ≈ ***1***
  - Logarithmic ≈ $\log_2 \boldsymbol{n}$
  - Linear ≈ ***n***
  - N-Log-N ≈ $\boldsymbol{n} \log_2 \boldsymbol{n}$
  - Quadratic ≈ $\boldsymbol{n}^2$
  - Cubic ≈ $\boldsymbol{n}^3$
  - Exponential ≈ $\boldsymbol{2}^n$



Let's practice classifying functions

Basic Functions

Which are more alike ?

n^{1000} n^2 2^n

polynomial

Basic Functions

Which are more alike ?

$1000n^2$ $3n^2$ $2n^3$

quadratic

Basic functions growth rates

n	log$n$	n	$n$log$n$	n^2	n^3	2^n
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,094	262,144	$1.84 * 10^{19}$
128	7	128	896	16,384	2,097,152	$3.40 * 10^{38}$
256	8	256	2,048	65,536	16,777,216	$1.15 * 10^{77}$
512	9	512	4,608	262,144	134,217,728	$1.34 * 10^{154}$
1024	10	1,024	10,240	1,048,576	1,073,741,824	$1.79 * 10^{308}$

Algorithm Types

- Time takes to solve an instance of a
  - Linear Algorithm is
    - Never greater than $c*n$
  - Quadratic Algorithm is
    - Never greater than $c*n^2$
  - Cubic Algorithm is
    - Never greater than $c*n^3$
  - Polynomial Algorithm is
    - Never greater than n^k
  - Exponential Algorithm is
    - Never greater than c^n

where c & k are appropriate constants

The analogy between comparing functions and comparing numbers

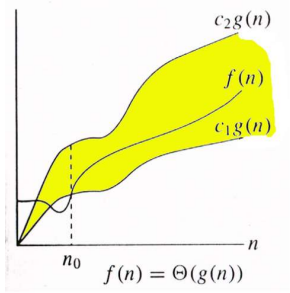
One thing you may have noticed by now is that these relations are kind of like the "<, >" relations for the numbers

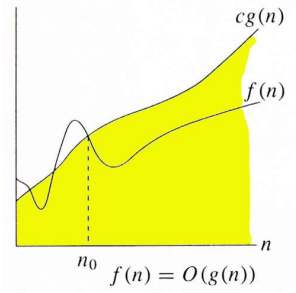
$$f \leftrightarrow g \;\approx\; a \leftrightarrow b$$

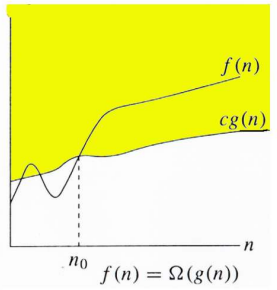
$$f(n) = O(g(n)) \;\approx\; a \leq b$$
$$f(n) = \Omega(g(n)) \;\approx\; a \geq b$$
$$f(n) = \Theta(g(n)) \;\approx\; a = b$$
$$f(n) = o(g(n)) \;\approx\; a < b$$
$$f(n) = \omega(g(n)) \;\approx\; a > b$$

"Relatives" of notations

- "Relatives" of the Big-Oh
  - Ω ($g(n)$): Big Omega – asymptotic *lower* bound
  - Θ ($g(n)$): Big Theta – asymptotic *tight* bound
- Big-Omega – think of it as the inverse of $O(n)$
  - $f(n)$ is Ω ($g(n)$) if $g(n)$ is $O(f(n))$
- Big-Theta – combine both Big-Oh and Big-Omega
  - $f(n)$ is Θ (g(n)) if $f(n)$ is $O(g(n))$ and $g(n)$ is Ω ($f(n)$)
- Make the difference:
  - $3n+3$ is O(n) and is Θ (n)
  - $3n+3$ is O(n^2) but is not Θ (n^2)







- Little-oh – $f(n)$ is $o(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is not Θ ($g(n)$)
  - $2n+3$ is $o(n^2)$
  - $2n + 3$ is $o(n)$?

Math you need to Review

- Exponents:
  $a^{(b+c)} = a^b a^c$
  $a^{bc} = (a^b)^c$
  $a^b / a^c = a^{(b-c)}$
  $b = a^{\log_a b}$
  $b^c = a^{c*\log_a b}$
- Logarithms:

$x = \log_b a$ is the exponent for $a = b^x$.

Natural log: $\ln a = \log_e a$

Binary log: $\lg a = \log_2 a$

$\lg^2 a = (\lg a)^2$

$\lg \lg a = \lg (\lg a)$

$$a = b^{\log_b a}$$

$$\log_c (ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b (1/a) = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

Logarithms and exponentials – Bases

- If the base of a logarithm is changed from one constant to another, the value is altered by a constant factor.
  - **Ex:** $\log_{10} n * \mathbf{\log_2 10} = \log_2 n$.
  - Base of logarithm is not an issue in asymptotic notation.
- Exponentials with different bases differ by a exponential factor (not a constant factor).
  - **Ex:** $2^n = \mathbf{(2/3)^n} * 3^n$.
- **Exponentials and polynomials:**

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0$$

$$\Rightarrow n^b = o(a^n)$$

Exercise

- Order the following functions by their asymptotic growth rates
  1. $n\log_2 n$
  2. $\log_2 n^3$
  3. n^2
  4. $n^{2/5}$
  5. $2^{log_2 n}$
  6. $\log_2(\log_2 n)$
  7. $\mathrm{Sqr}(\log_2 n)$

The way to remember these notations

Theta	$f(n) = \Theta(g(n))$	$f(n) \approx c\, g(n)$
Big Oh	$f(n) = O(g(n))$	$f(n) \leq c\, g(n)$
Big Omega	$f(n) = \Omega(g(n))$	$f(n) \geq c\, g(n)$
Little Oh	$f(n) = o(g(n))$	$f(n) \ll c\, g(n)$
Little Omega	$f(n) = \omega(g(n))$	$f(n) \gg c\, g(n)$

Properties

- Transitivity (truyền ứng)

$$f(n) = \Theta(g(n)) \;\&\; g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \;\&\; g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \;\&\; g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$

- Reflexivity

$$f(n) = \Theta(f(n)) \qquad f(n) = O(g(n)) \qquad f(n) = \Omega(g(n))$$

- Symmetry (đối xứng)

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n))$$

- Transpose Symmetry (Đối xứng chuyển vị)

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n))$$

Example: $A = 5n^2 + 100n$, $B = 3n^2 + 2$. Show that $A \in \Theta(B)$

Ans: $A \in \Theta(n^2)$, $n^2 \in \Theta(B) \Rightarrow A \in \Theta(B)$

Limits

- $\lim_{n\to\infty} [f(n) / g(n)] = 0 \Rightarrow f(n) \in o(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in O(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in \Theta(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] \Rightarrow f(n) \in \Omega(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)] = \infty \Rightarrow f(n) \in \omega(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)]$ undefined $\Rightarrow$ can't say

Exercise: Express functions in A in asymptotic notation using functions in B.

A	B	
$\log_3(n^2)$	**$\log_2(n^3)$**	**$A \in \Theta(B)$**

$\log_b a = \log_c a / \log_c b$; A = 2lg$n$ / lg3, B = 3lg$n$, A/B =2/(3lg3) $\Rightarrow$ A $\in \Theta$(B)

$n^{\lg 4}$	**$3^{\lg n}$**	**$A \in \omega(B)$**

$a^{\log b} = b^{\log a}$; B =$3^{\lg n}$=$n^{\lg 3}$; A/B =$n^{\lg(4/3)} \to \infty$ as $n\to\infty \Rightarrow$ A $\in \omega$ (B)

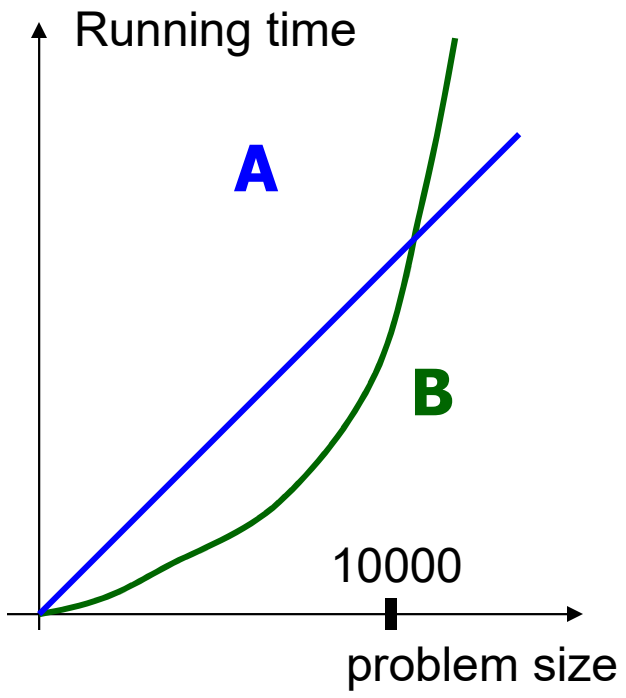
Exercise

Show that

1) $3n^2 - 100n + 6 = O(n^2)$
2) $3n^2 - 100n + 6 = O(n^3)$
3) $3n^2 - 100n + 6 \neq O(n)$
4) $3n^2 - 100n + 6 = \Omega(n^2)$
5) $3n^2 - 100n + 6 \neq \Omega(n^3)$
6) $3n^2 - 100n + 6 = \Omega(n)$
7) $3n^2 - 100n + 6 = \mathbf{\Theta}(n^2)$
8) $3n^2 - 100n + 6 \neq \mathbf{\Theta}(n^3)$
9) $3n^2 - 100n + 6 \neq \mathbf{\Theta}(n)$

Final notes

- Even though in this course we focus on the asymptotic growth using big-Oh notation, practitioners do care about constant factors occasionally
- Suppose we have 2 algorithms
  - Algorithm A has running time $30000n$
  - Algorithm B has running time $3n^2$
- Asymptotically, algorithm A is better than algorithm B
- However, if the problem size you deal with is always less than 10000, then the quadratic one is faster



Contents

1.4. Pseudocode

- To describe the algorithm, we can use a certain programming language. However, that could make the algorithm description complex and difficult to grasp. Therefore, to describe algorithms people often use pseudo language, which allows:
  - describing algorithms in everyday language
  - using command structures similar to those of programming languages.

In this course, we will mostly use pseudocode to describe an algorithm.

- Pseudocode is a high-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

```
Function arrayMax(A, n)
   //Input: array A of n integers
   //Output: maximum element of A
begin
        currentMax ← A[0]
        for i ← 1 to n – 1
          if (A[i] > currentMax) then
             currentMax ← A[i]
         endif;
   endfor;
  return currentMax;
end;
```

1.4. Pseudocode

- **Declare variables**

  integer x,y;

  real u, v;

  boolean a, b;

  char c, d;

  datatype x;

- **Assign statement**

  x = expression;

  or

  x ← expression;

  Example: x ← 1+4; y=a*y+2;

1.4. Pseudocode

Control flow:

```
if  condition then
      sequence of statements
else
      sequence of statements
endif;

while  condition do
   sequence of statements
endwhile;

repeat
  sequence of statements
until condition;
```

1.4. Pseudocode

Control flow:

```
for i=n1 to n2 [step d]
   sequence of statements
endfor;
```

```
Case
    condition1: statement1;
   condition2: statement2;
   .
   .
   .
   conditionn: statement n;
endcase;
```

1.4. Pseudocode

- **Input/output:**

  read(X); /* X is variable */

  print(data**);**

  or print(notification**);**

- **Function and procedure:**

```
Function name(arguments)
begin
    declare variables;
    statements inside the function;
    return (value);
end;

Procedure name(arguments)
begin
    declare variables;
    statements inside the procedure;
end;
```

Example 1: find max element of an array

```
Function arrayMax(A, n)
  //Input: array A of n integers
  //Output: maximum element of A
begin
      currentMax ← A[0]
      for i ← 1 to n – 1
       if (A[i] > currentMax) then
          currentMax ← A[i]
       endif;
  endfor;
 return currentMax;
end;
```

1.4. Pseudocode

Example 2: swap values of two variables

```
Procedure swap(x, y)
begin
        temp=x;
        x = y;
        y = temp;
end;
```

Or briefly as follow:

```
Procedure swap(x, y)
        temp=x;
        x = y;
        y = temp;
```

Pseudocode: Example 3

Problem: Find the smallest prime number that is greater than a given positive integer n

- First we build a function Is_Prime to test whether a positive integer m is a prime number
- Using this function, we build an algorithm to solve the problem:
  - If $m = a*b$ with $1 < a, b < m$, then one of the two numbers a and b will not exceed $\sqrt{m}$. Therefore, the largest prime number that is smaller than m will never exceed $\sqrt{m}$ ➔ m is the prime number if it does not have any integer divisors in the range $[2, \sqrt{m}]$

Pseudocode: Example 3

Algorithm to check whether a positive integer is a prime:

- **Input:** Positive integer *m*.
- **Output:** **true** if *m* is a prime, **false** otherwise.

```
Function  Is_prime(m)
begin
      i = 2;
      while (i*i <= m) and (m mod i ≠ 0) do
         i=i+1;
    endwhile;
      Is_prime = i > sqrt(m);
end Is_Prime;
```

Algorithm to find the smallest prime number that is greater than the positive integer *n*:

- Algorithm uses the function Is_prime as the sub procedure.
- **Input:** Positive integer *n*.
- **Output:** *m* is the smallest prime number that is greater than *n*.

```
procedure  Lagre_Prime(n)
begin
      m = n+1;
      while not  Is_prime(m)  do
          m=m+1;
    endwhile;
end;
```

Contents

Running time calculation

- Experimental evaluation of running time:
  - Write a program implementing the algorithm
  - Run the program and measure the running time
  - Cons of experimental evaluation:
    - It is necessary to implement the algorithm, which may be difficult
    - Results may not be indicative of the running time on other inputs not included in the experiment
    - In order to compare two algorithms, the same hardware and software environments must be used

  ➔ We need: **Theoretical Analysis of Running Time**
- Theoretical Analysis of Running Time:
  - Uses a pseudo-code description of the algorithm instead of an implementation
  - Characterizes running time as a function of the input size, ***n***
  - Takes into account all possible inputs
  - Allows us to evaluate the speed of an algorithm independent of the hardware/software environment (Changing the hardware/software environment affects the running time by a constant factor, but does not alter the growth rate of the running time)

Primitive Operations

- For theoretical analysis, we will count **primitive** or **basic** operations, which are simple computations performed by an algorithm

  could be implemented within the running time that is bounded above by a constant independent of the input data size.

- Examples of primitive operations:
  - Evaluating an expression x^2+e^y
  - Assigning a value to a variable `cnt ← cnt+1`
  - Indexing into an array `A[5]`
  - Calling a method `mySort(A,n)`
  - Returning from a method `return(cnt)`

Running Time Calculations: General rules

1. **Consecutive Statements**: The sum of running time of each segment.

- *Running time of* "P; Q", where P is implemented first, then Q, is

$$Time(P; Q) = Time(P) + Time(Q)\ ,$$

or if using asymptotic Theta:

$$Time(P; Q) = \Theta(\max(Time(P), Time(Q)).$$

2. **FOR loop**: The number of iterations times the time of the inside statements.

```
for i =1 to m do P(i);
```

Assume running time of $P(i)$ is $t(i)$, then the running time of for loop is $\sum_{i=1}^{m} t(i)$

3. **Nested loops**: The product of the number of iterations times the time of the inside statements.

```
for i =1 to n do
  for j =1 to m do P(j);
```

Assume the running time of $P(j)$ is $t(j)$, then the running time of this nested loops is:

Some Examples

Case1: for (i=0; i<n; i++)
for (j=0; j<n; j++)
k++;

$O(n^2)$

$O(n)$ work followed by $O(n^2)$ work, is also $O(n^2)$

$O(n^2)$

Running Time Calculations: General rules

4. If/Else

```
if (condition )
    S1;
else
    S2;
```

The testing time plus the larger running time of the S1 and S2.

Characteristic statement

- Definition. The characteristic statement is the statement being executed with frequency at least as well as any statement in the algorithm.
- If the execution time of each statement is bounded above by a constant, then the running time of the algorithm will be the same size as the number of times the execution of the characteristic statement

=> To evaluate the running time, one can count the number of times the characteristic statement being executed

Example: Calculating Fibonacci Sequences

```
function Fibrec(n)
    if n <2 then return n;
    else return Fibrec(n-1)+Fibrec(n-2);
```

- Fibonacci Sequence:
  - $f_0=0$;
  - $f_1=1$;
  - $f_n=f_{n-1}+f_{n-2}$
- Order of Fibrec is f_n
- Order of Fibiter is n

```
function Fibiter(n)
    i=0;
    j=1;
    for k=1 to n do
      j=i + j;
      i=j – i;
    return j;
```

Characteristic statement

• The number of times this characteristic statement being executed is n ➔ The running time of Fibiter is $O(n)$

n	10	20	30	50	100
Fibrec	8ms	1sec	2min	21days	10^9years
Fibiter	0.17ms	0.33ms	0.5ms	0.75ms	1.5ms

Exercise 1: Maximum Subarray Problem

Given an array of integers $A_1, A_2, \ldots, A_N$, find the maximum value of $\sum_{k=i}^{j} A_k$

For convenience, the maximum subsequence sum is zero if all the integers are negative.

Algorithm 1. Brute force

```
int maxSum = 0;
for (int i=0; i<n; i++) {
    for (int j=i; j<n; j++) {
        int sum = 0;
        for (int k=i; k<=j; k++)
            sum += a[k];
        if (sum > maxSum)
            maxSum = sum;
    }
}
```

Select the statement **`sum+=a[k]`** as the characteristic statement

➔ Running time of the algorithm: $O(n^3)$

Algorithm 2. Brute force with better implement

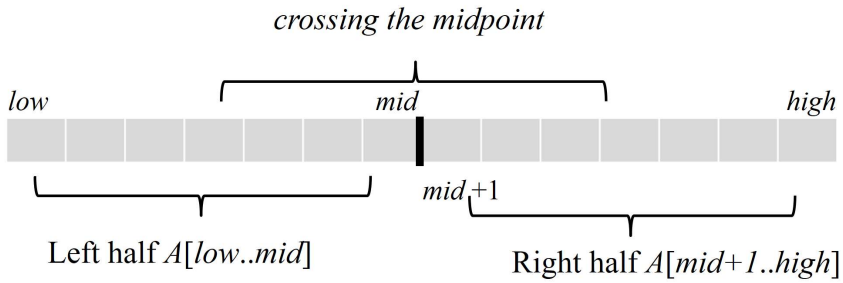
```
int maxSum = a[0];
for (int i=0; i<n; i++) {
   int sum = 0;
   for (int j=i; j<n; j++) {
      sum += a[j];
      if (sum > maxSum)
          maxSum = sum;
   }
}
```

$O(n^2)$

Algorithm 3. Recursive algorithm

```
MaxLeft(a, low, mid);
{
  maxSum = -∞; sum = 0;
  for (int k=mid; k>=low; k--) {
       sum = sum+a[k];
       maxSum = max(sum, maxSum);
   }
    return maxSum;
}
```

⬅ O(n)




```
MaxRight(a, mid, high);
{
     maxSum = -∞; sum = 0;
    for (int k=mid; k<=high; k++){
      sum = sum+a[k];
      maxSum = max(sum, maxSum);
    }
    return maxSum;
}
```

⬅ O(n)

```
MaxSub(a, low, high);
{
  if (low = high) return a[low] //base case: only 1 element
   else
   {    mid = (low+high)/2;
        wL = MaxSub(a, low, mid);
        wR = MaxSub(a, mid+1, high);
        wM = MaxLeft(a, low, mid) + MaxRight(a, mid, high);
        return max(wL, wR, wM);
   }
}
```

⬅ T(n/2)

⬅ T(n/2)

T(*n*) = 2T(*n*/2)+O(*n*)

➔ T(*n*) = O(*n*log*n*)

Algorithm 4. Dynamic programming

The primary steps of dynamic programming:

1. Divide:

- Define s_i the value of max subarray of the array $a_1, a_2, ..., a_i$, $i = 1, 2, ..., n$.
- Clearly, s_n is the solution.

3. Construct the final solution:

- $s_1 = a_1$
- Assume $i > 1$ and we already know the value of s_k with $k = 1, 2, ..., i\text{-}1$. Now we need to calculate the value of s_i which is the value of max subarray of the array:

  $a_1, a_2, ..., a_{i-1}, a_i$.
- We see that: the max subarray of this array $a_1, a_2, ..., a_{i-1}, a_i$ could either include the element a_i or not include the element a_i ➔ therefore, the max subarray of the array $a_1, a_2, ..., a_{i-1}, a_i$ could only be one of these 2 arrays:
  - The max subarray of the array $a_1, a_2, ..., a_{i-1}$
  - The max subarray of the array $a_1, a_2, ..., a_i$ *ending at* a_i.

➔ Thus, we have $s_i = max\ \{s_{i-1}, e_i\}$, $i = 2, ..., n$.

where e_i is the value of the max subarray $a_1, a_2, ..., a_i$ ending at a_i.

To calculate e_i, *we could use the recursive relation*:

- $e_1 = a_1$;
- $e_i = \max \{a_i, e_{i-1} + a_i\}$, $i = 2, ..., n$.

```
MaxSub(a)
{
    smax  = a[1];        (* smax – the value of max subarray  *)
    ei    = a[1];        (* ei   – the value of max subarray ending at a[i] *)
    for i = 2 to n {
          ei = max { a[i], ei + a[i] }
          smax = max { smax, ei }
    }
}
```

Exercise 2: Prime Algorithm

- The algorithm to check a positive integer $m > 1$ be a prime or not.

```
Algorithm  PRIME(m)
    for i=2 to sqrt(m) do
        if  m mod i = 0  then return FALSE
    return YES
```

- Running time: $T(n) = O(\sqrt{m})$. Polynomial algorithm?
- Data size: $n \approx \log_2 m$

$\Rightarrow \quad m \approx 2^n$

$\Rightarrow \quad T(n) = O(m^{1/2}) = O(2^{n/2})$.

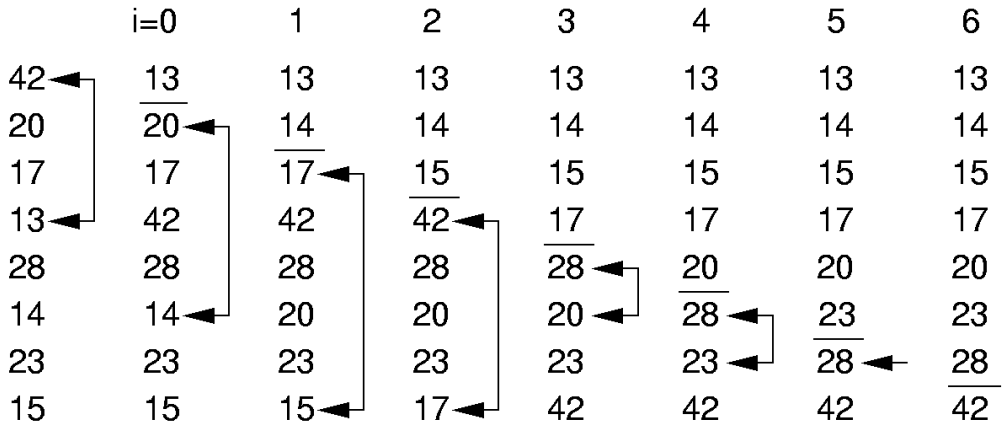
Exponential running time!

Exercise 3: Selection sort

- Sort a sequence of numbers in ascending order
- Algorithm:
  - Find the smallest and move it to the first place
  - Find the next smallest and move it to the second place
  - Find the next smallest and move it to the 3rd place
  - …

```
void selectionSort(int a[], int n){
  int i, j, index_min;
  for (i = 0; i < n-1; i++) {
    index_min = i;
     //Find the smallest element from a[i+1] till the last element
    for (j = i+1; j < n; j++)
       if (a[j] < a[index_min]) index_min = j;
    //move the element a[index_min] to the ith place:
    swap(a[i], a[index_min]);
  }
}
```

```
void swap(int &a,int &b)
{
   int temp = a;
   a = b;
   b = temp;
}
```



Important Series

$$S(N) = 1 + 2 + \ldots + N = \sum_{i=1}^{N} i = N(1+N)/2$$

- Sum of squares: $\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$ for large N
- Sum of exponents: $\sum_{i=1}^{N} i^k \approx \frac{N^{k+1}}{|k+1|}$ for large N and k ≠ -1
- Geometric series: $\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$
  - Special case when A = 2
    - $2^0 + 2^1 + 2^2 + \ldots + 2^N = 2^{N+1} - 1$

Exercise 4

- Give asymptotic big-Oh notation for the running time $T(n)$ of the following statement segment:

```
for (int i = 1; i<=n; i++)
      for (int j = 1; j<= i*i*i; j++)
       for (int k = 1; k<=n; k++)
                  x = x + 1;
```

- Ans:

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i^3} \sum_{k=1}^{n} 1$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{i^3} n = \sum_{i=1}^{n} n(\sum_{j=1}^{i^3} 1) = \sum_{i=1}^{n} ni^3$$

$$= n\sum_{i=1}^{n} i^3 \le n\sum_{i=1}^{n} n^3 = n^4 \sum_{i=1}^{n} 1 = n^5$$

So $T(n) = O(n^5)$.

Exercise 5

- Give asymptotic big-Oh notation for the running time $T(n)$ of the following statement segment:

a)
```
int x = 0;
for (int i = 1; i <=n; i *= 2)
      x=x+1;
```

- Ans:

The loop **for** is executed $\log_2 n$ times, therefore $T(n) = O(\log_2 n)$.

b)
```
int x = 0;
for (int i = n; i > 0; i /= 2)
      x=x+1;
```

- Ans:

The loop **for** is executed …..

Exercise 6

Give asymptotic big-Oh notation for the running time $T(n)$ of the following statement segment:

```
int n;
if (n<1000)
   for (int i=0; i<n; i++)
     for (int j=0; j<n; j++)
        for (int k=0; k<n; k++)
            cout << "Hello\n";
else
   for (int j=0; j<n; j++)
      for (int k=0; k<n; k++)
         cout << "world!\n";
```

Ans:

- $T(n)$ is the constant when $n<1000$. $T(n) = O(n^2)$.

Contents

1.6. Solving recurrence

- When an algorithm contains a **recursive call to itself**, its running time can often be described by a recurrence. A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs.

Example 1: Fibonacci Sequence:

- $f_0=0$;
- $f_1=1$;
- $f_n=f_{n-1}+f_{n-2}$

```
1    function Fibrec(n)
2        if (n < 2) then
3            return n;
4        else return Fibrec(n-1)+Fibrec(n-2);
```

Let $T(n)$ be the running time for the function *Fibrec*(n).

If $n = 0$ or $n = 1$, then the running time is some constant value, which is the time to do the test at line2 and return at line3. We can say that $T(0) = T(1) = 1$, since constants do not matter.

For $n \geq 2$, the time to execute the function is the constant work at line 2 plus the work at line 4:

- Line 4 consists of an addition and two function calls:
  - The first function call is *Fibrec*(n - 1) and hence, by the definition of T, requires $T(n - 1)$ units of time.
  - Similarly, the second call function call *Fibrec*(n - 2) requires $T(n - 2)$ units of time.

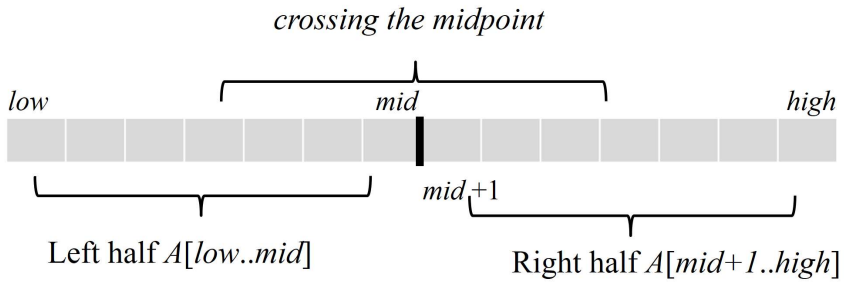
The total time required is then $T(n - 1) + T(n - 2) + 2$, where the 2 accounts for the work at line 2 plus the addition at line 4. Thus, for $n \geq 2$, we have the following formula for the running time of *Fibrec*(n):

$$\boldsymbol{T(n) = T(n - 1) + T(n - 2) + 2}$$

Example 2: Recursive algorithm to solve max subarray

```
MaxLeft(a, low, mid);
{
  maxSum = -∞; sum = 0;
  for (int k=mid; k>=low; k--) {
       sum = sum+a[k];
       maxSum = max(sum, maxSum);
   }
    return maxSum;
}
```

⟸ O(n)




```
MaxRight(a, mid, high);
{
     maxSum = -∞; sum = 0;
     for (int k=mid; k<=high; k++){
       sum = sum+a[k];
       maxSum = max(sum, maxSum);
    }
    return maxSum;
}
```

⟸ O(n)

```
MaxSub(a, low, high);
{
  if (low = high) return a[low] //base case: only 1 element
   else
   {    mid = (low+high)/2;
        wL = MaxSub(a, low, mid);
        wR = MaxSub(a, mid+1, high);
        wM = MaxLeft(a, low, mid) + MaxRight(a, mid, high);
        return max(wL, wR, wM);
   }
}
```

⟸ T(n/2)

⟸ T(n/2)

T(*n*) = 2T(*n*/2) + O(*n*)

1.6. Solving recurrence

- When an algorithm contains a **recursive call to itself**, its running time can often be described by a recurrence. A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs.
- This chapter offers three methods for solving recurrences--that is, for obtaining asymptotic "Θ" or "O" bounds on the solution:
  - ***Backward substitution*** starts from the equation itself, work backwards, substituting values of the function for previous ones
  - ***Recurrence trees*** involves mapping out the recurrence tree for an equation. Starting from the equation, you unfold each recursive call to the function and calculate the non-recursive cost at each level of the tree. Then, you find a general formula for each level and take a summation over all such levels
  - ***Master method*** provides bounds for recurrences of the form

1.6.1. Backward substitution

Backward substitution: this works exactly as its name suggests. Starting from the equation itself, work backwards, substituting values of the function for previous ones.

Example: $T(n)= T(n-1) + 2n$

where $T(1) = 5$

Answer: We begin by unfolding the recursion by a simple substitution of the function values.

- We observe that:

$$T(n-1) = T((n-1) -1) + 2(n-1)$$
$$= T(n-2) + 2 (n-1)$$

- Substituting into the original equation:

$$T(n)=T(n-2)+2(n-1)+2n$$

1.6.1. Backward substitution

- If we continue to do that we get

  $T(n) = T(n\text{-}2) + 2(n\text{-}1) + 2n$

  $= T(n\text{-}3) + 2(n\text{-}2) + 2(n\text{-}1) + 2n$

  $= T(n\text{-}4) + 2(n\text{-}3) + 2(n\text{-}2) + 2(n\text{-}1) + 2n$

  .....

  $= T(n\text{-}i) + \sum_{j=0}^{i-1} 2(n-j)$ *function's value at the i^{th} iteration*

- Solving the sum we get

  $T(n) = T(n\text{-}i) + 2n(i\text{-}1) - 2(i\text{-}1)(i\text{-}1+1)/2 + 2n$

  $= T(n\text{-}i) + 2n(i\text{-}1) - i^2 + i + 2n$

- We want to get rid of the recursive term $T(n\text{-}i)$. To do that, we need to know at what iteration we reach our base case, i.e. for what value of i can we use the initial condition $T(1)=5$?

We get the base case when $n - i = 1$ or $i = n - 1$

- Substituting in the equation above we get

  $T(n) = T(1) + 2n(n\text{-}1\text{-}1) - (n\text{-}1)^2 + (n\text{-}1) + 2n$

  $= 5 + 2n(n\text{-}2) - (n^2\text{-}2n+1) + (n\text{-}1) + 2n = n^2 + n + 3$

➔ $T(n) = O(n^2)$

1.6.2. Recurrence Trees

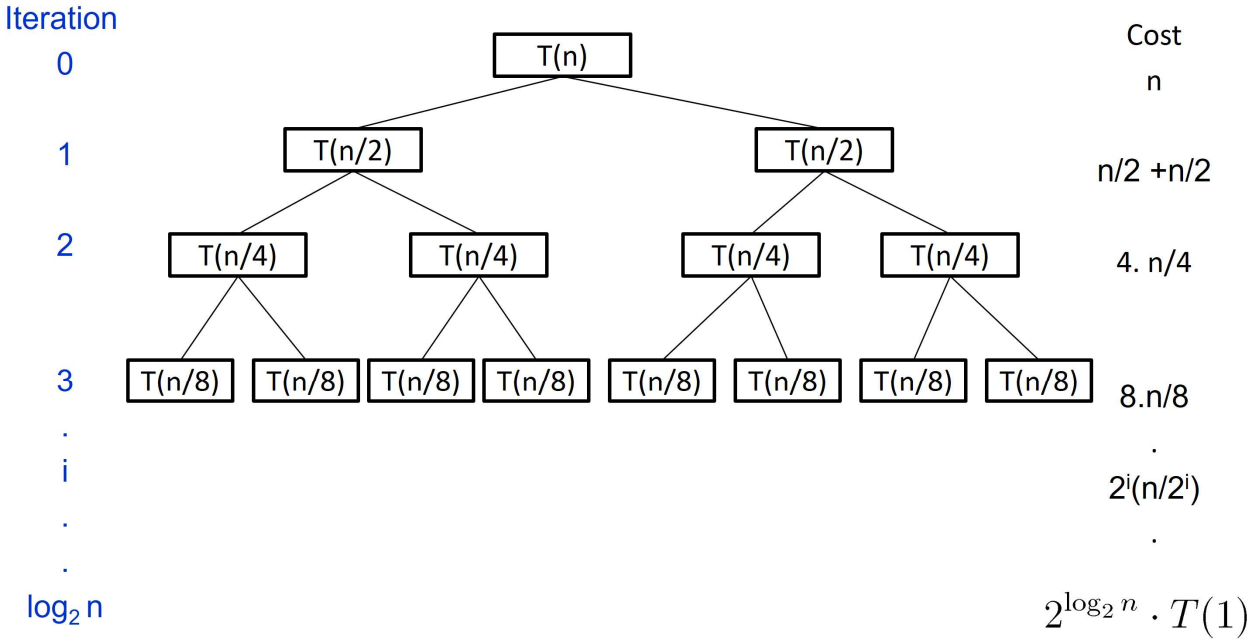
- When using recurrence trees, we graphically represent the recursion
- Each node in the tree is an instance of the function. As we progress downward, the size of the input decreases
- The contribution of each level to the function is equivalent to the number of nodes at that level times the non-recursive cost on the size of the input at that level
- The tree ends at the depth at which we reach the base case
- As an example, we consider a recursive function of the form

$$T(n) = 2T(n/2) + n, \quad T(1) = 4$$

1.6.2. Recurrence Trees

Example 1: Consider the following concrete example

$$T(n) = 2T(n/2) + n, \quad T(1) = 4$$



- The value of the function is the summation of the value of all levels.

$$T(n) = \sum_{i=0}^{log_2 n} 2^i (\frac{n}{2^i})$$

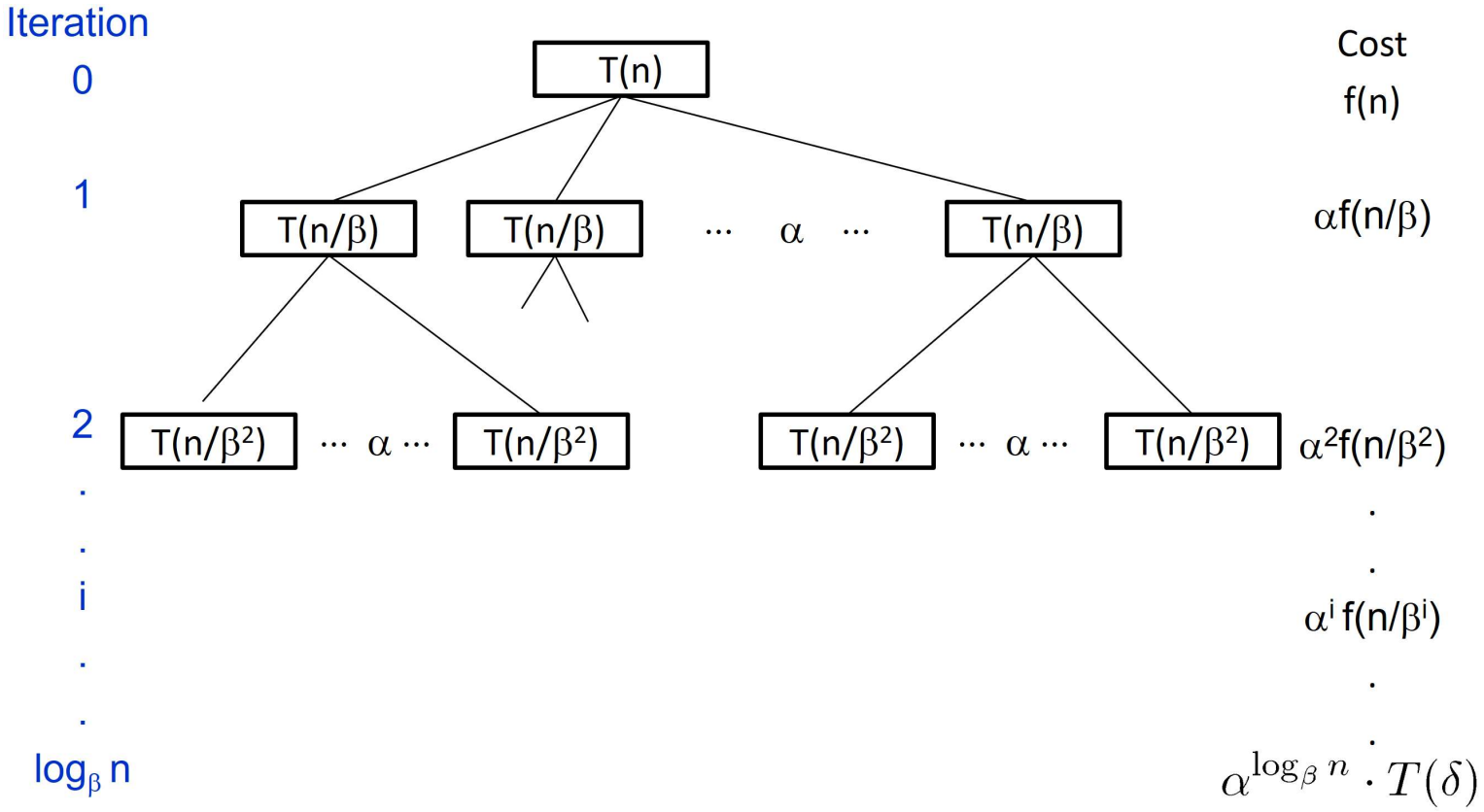
- We treat the last level as a special case since its non-recursive cost:

$$T(n) = 4n + \Sigma_{i=0}^{(\log_2 n - 1)} 2^i \frac{n}{2^i} = n(\log n) + 4n$$

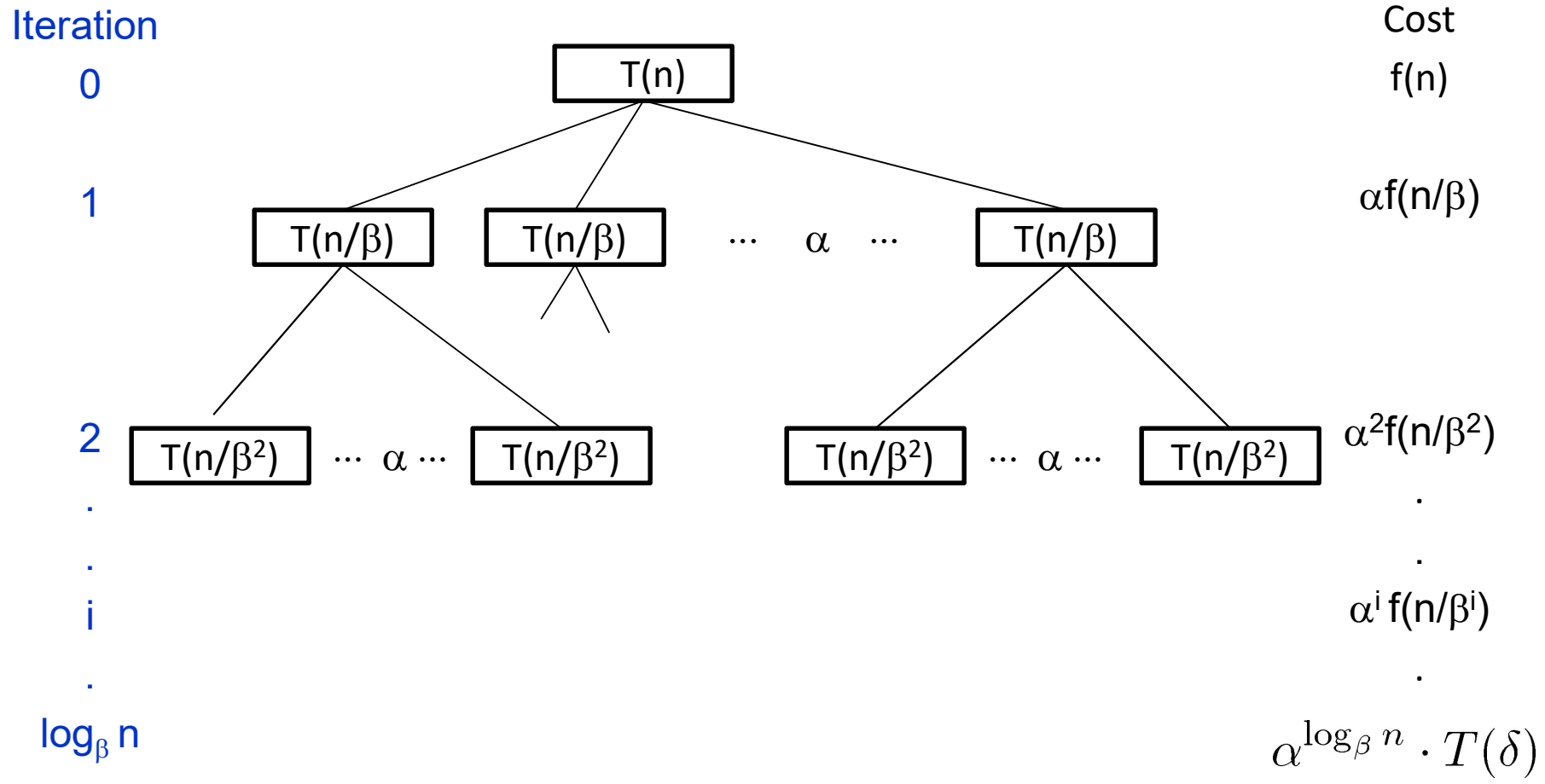
1.6.2. Recurrence Trees

Example 2: We consider a recursive function of the form

$$T(n) = \alpha T(n/\beta) + f(n), \quad T(\delta) = c$$



1.6.2. Recurrence Trees

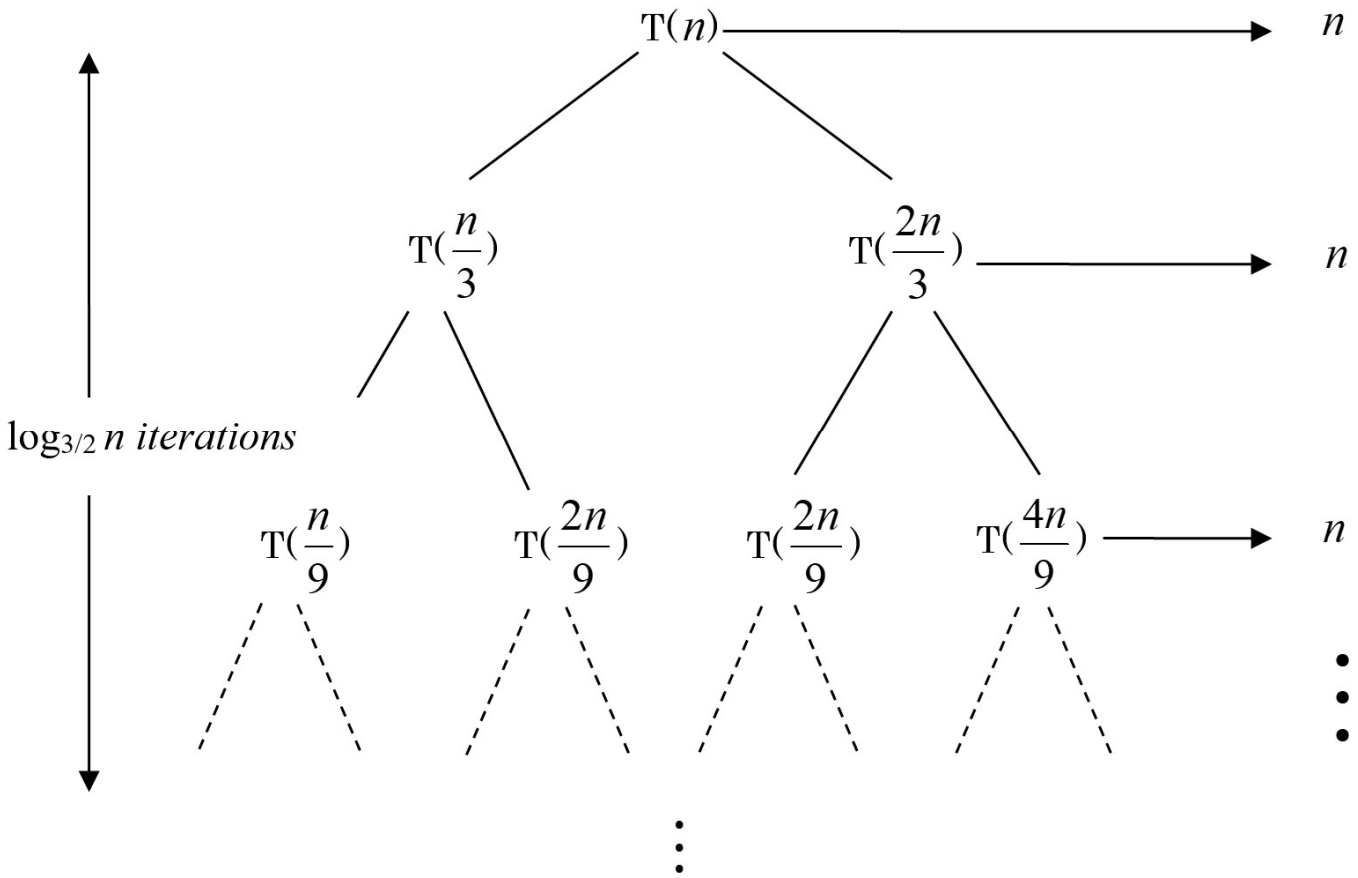


- The total value of the function is the summation over all levels of the tree:

$$T(n) = \sum_{i=0}^{log_\beta n} \alpha^i f(\frac{n}{\beta^i}) \quad \Rightarrow \quad T(n) = \alpha^{log_\beta n} T(\delta) + \sum_{i=0}^{(log_\beta n)-1} \alpha^i f(\frac{n}{\beta^i})$$

1.6.2. Recurrence Trees

Example 3: $T(n) = T(n/3) + T(2n/3) + n$, $T(1) = 1$



- Note that the tree have ($\log_{3/2} n$) levels. But, not all branches have same depth. The costs near the leaves hard to calculate. We can estimate.

1.6.2. Recurrence Trees

Example 3: $T(n) = T(n/3) + T(2n/3) + n, \quad T(1) = 1$

- *Overestimate*: By considering all branches to be of max depth, we have $T(n) \leq n\ (\log_{3/2} n) + n,$

and consequently

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

- *Underestimate*: By counting only the $\log_3 n$ complete levels, and ignoring the rest, we obtain $T(n) \geq n \log_3 n$

and hence

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

Thus, $T(n) = \Theta(n \log n)$.

1.6.3. Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

$$T(n) = a\,T(n/b) + f(n)$$

$$T(1) = c$$

where $a \geq 1$, $b \geq 2$, $c > 0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem: Pitfalls

- Let T(n) be a monotonically increasing function that satisfies

  $T(n) = a\,T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

- You cannot use the Master Theorem if
  - T(n) is not monotone, e.g. T(n) = sin(n)
  - f(n) is not a polynomial, e.g. T(n) = 2T(n/2)+ 2^n
  - b cannot be expressed as a constant, e.g. $T(n) = T(\sqrt{n})$

Master Theorem: Example 1

- Let $T(n) = T(n/2) + \frac{1}{2} n^2 + n$. What are the parameters?

  a = 1

  b = 2

  d = 2

  Therefore, which condition applies?

  $1 < 2^2$, case 1 applies

- We conclude that $T(n) \in \Theta(n^d) = \Theta(n^2)$

Master Theorem

- Let $T(n)$ be a monotonically increasing function that satisfies

  $T(n) = a\,T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c > 0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem: Example 2

- Let $T(n) = 2\,T(n/4) + \sqrt{n} + 42$. What are the parameters?

  a = 2

  b = 4

  d = 1/2

  Therefore, which condition applies?

  $2 = 4^{1/2}$, case 2 applies

- We conclude that $T(n) \in \Theta(n^d \log n) = \Theta(\log n \sqrt{n})$

- Let $T(n) = 2\,T(n/4) + \sqrt{n} + 42$. What are the parameters?

  a =

  b =

  d =

  Therefore, which condition applies?

Master Theorem: Example 3

- Let T(n)= 3 T(n/2) + (3/4)n + 1. What are the parameters?

  a = 3

  b = 2

  d = 1

  Therefore, which condition applies?

  $3 > 2^1$, case 3 applies

- We conclude that $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$

- Note that $\log_2 3 \approx 1.584\ldots$, can we say that $T(n) \in \Theta(n^{1.584})$????

  No, because $\log_2 3 \approx 1.5849\ldots$ and $n^{1.584} \notin \Theta(n^{1.5849})$

Master Theorem: 'Fourth' Condition

- Recall that we cannot use the Master Theorem if f(n) (the non-recursive cost) is not a polynomial.
- There is a limited 4th condition of the Master Theorem that allows us to consider polylogarithmic functions:

Corollary

If $f(n) \in \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$ then

$$T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$$

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

$T(n) = a\ T(n/b) + f(n)$

$T(1) = c$

where $a \geq 1$, $b \geq 2$, $c > 0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

‘Fourth’ Condition: Example

- Say we have the following recurrence relation

  $T(n) = 2\,T(n/2) + n\log n$

- Clearly, $a=2$, $b=2$, but $f(n)$ is not a polynomial.
- However, we have $f(n) \in \Theta(n\log n)$, $k=1$
- Therefore, by the 4th condition of the Master Theorem we can say that

$$T(n) \in \Theta(n^{\log_b a} \log^{k+1} n) = \Theta(n^{\log_2 2} \log^2 n) = \Theta(n \log^2 n)$$

Corollary

If $f(n) \in \Theta(n^{\log_b a} \log^k n)$ *for some* $k \geq 0$ *then*

$$T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$$

Example 4: Recursive algorithm to solve max subarray problem

Analyzing time complexity:

- Procedure `MaxLeft` and `MaxRight` requires

  $n/2 + n/2 = n$ additions

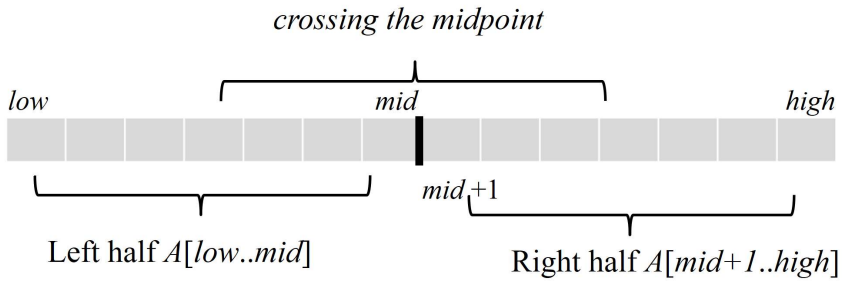
- Define T(n) the number of additions that the procedure `maxSub(a, 1, n)` need to perform, then we get the following recursion relation:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\frac{n}{2}) + T(\frac{n}{2}) + n = 2T(\frac{n}{2}) + n & n > 1 \end{cases}$$

```
MaxLeft(a, low, mid);
{
  maxSum = -∞; sum = 0;
  for (int k=mid; k>=low; k--) {
       sum = sum+a[k];
       maxSum = max(sum, maxSum);
   }
    return maxSum;
}

MaxRight(a, mid, high);
{
     maxSum = -∞; sum = 0;
    for (int k=mid; k<=high; k++){
       sum = sum+a[k];
       maxSum = max(sum, maxSum);
    }
    return maxSum;
}
```

```
MaxSub(a, low, high);
{
  if (low = high) return a[low] //base case: only 1 element
   else
   {    mid = (low+high)/2;
        wL = MaxSub(a, low, mid);
        wR = MaxSub(a, mid+1, high);
        wM = MaxLeft(a, low, mid) + MaxRight(a, mid, high);
        return max(wL, wR, wM);
   }
}
```



Example 4: Recursive algorithm to solve max subarray problem

- Let T(n)= 2T(n/2) + n. What are the parameters?

  a = 2

  b = 2

  d = 1

  Therefore, which condition applies?

  $2 = 2^1$, case 2 applies

- We conclude that $T(n) \epsilon \Theta(n^d \log n) = \Theta(n \log n)$

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

  $T(n) = a\,T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c > 0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Exercise 1: Recursive Binary Search

Input: An array *S* consists of *n* elements: S[0],…,S[*n*-1] in ascending order; Value ***key*** with the same data type as array *S*.

Output: the index in array if ***key*** is found, -1 if ***key*** is not found

Binary search algorithm: The value *key* either

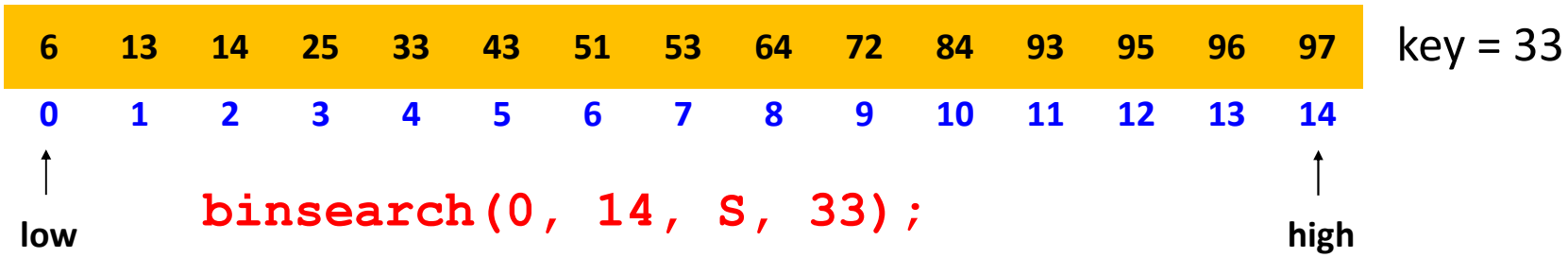
equals to the element at the middle of the array S,

or is at the left half (L) of the array S,

or is at the right half (R) of the array S.

(The situation L (R) happen only when *key* is smaller (larger) than the element at the middle of the array S)

```
int binsearch(int low, int high, int S[], int key)
```



Exercise 1: Recursive Binary Search

Input: An array *S* consists of *n* elements: S[0],…,S[*n*-1] in ascending order; Value ***key*** with the same data type as array *S*.

Output: the index in array if ***key*** is found, -1 if ***key*** is not found

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
     }
     else return -1;
}
```

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Exercise 1: Recursive Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       mid = (low + high) / 2;
       if (S[mid]==key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

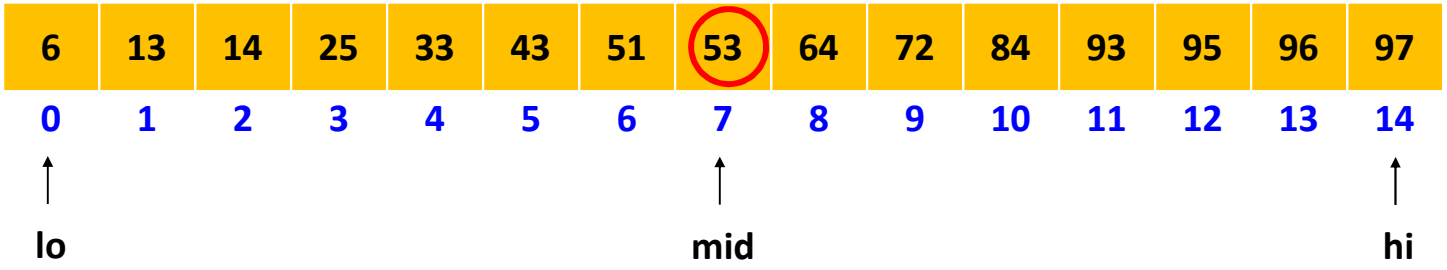
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

lo hi

Exercise 1: Recursive Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
     }
     else return -1;
}
```

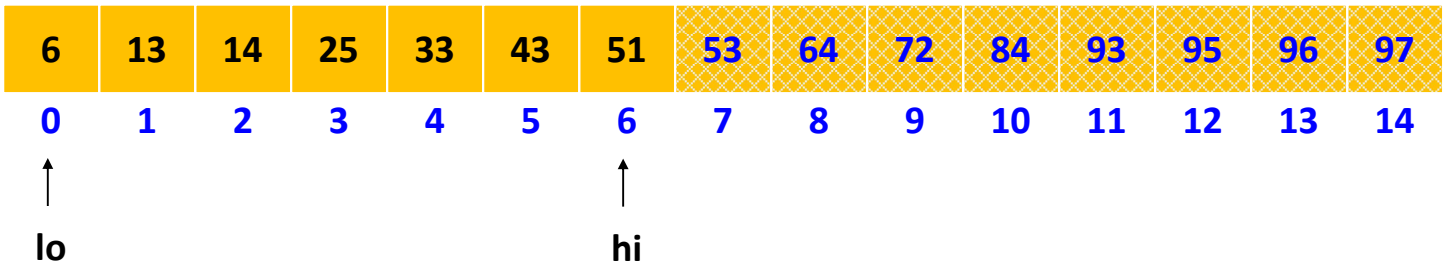
key=33



Exercise 1: Recursive Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       mid = (low + high) / 2;
       if (S[mid]==key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

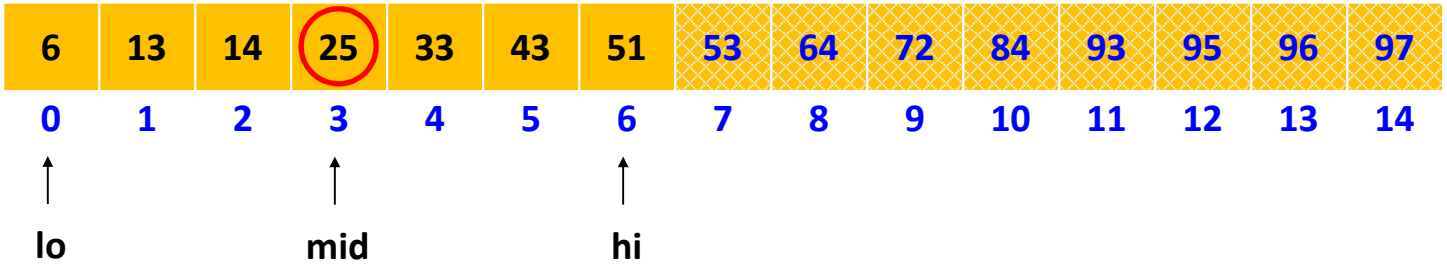
key=33



Exercise 1: Recursive Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
     }
     else return -1;
}
```

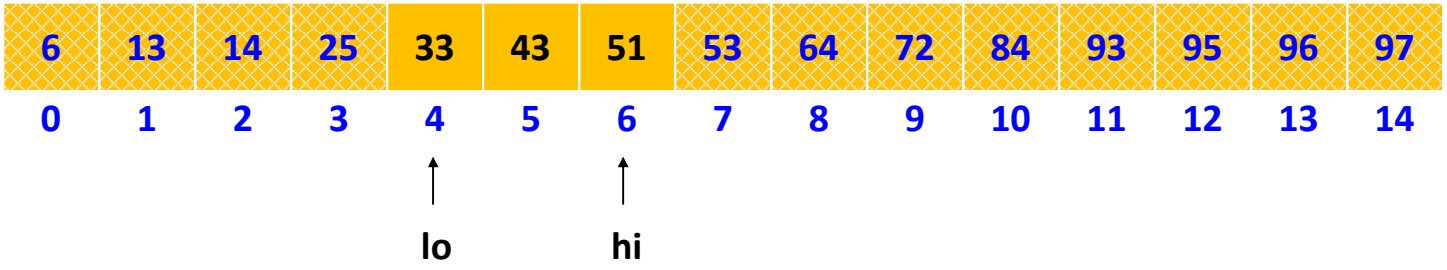
key=33



Exercise 1: Recursive Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       mid = (low + high) / 2;
       if (S[mid]==key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

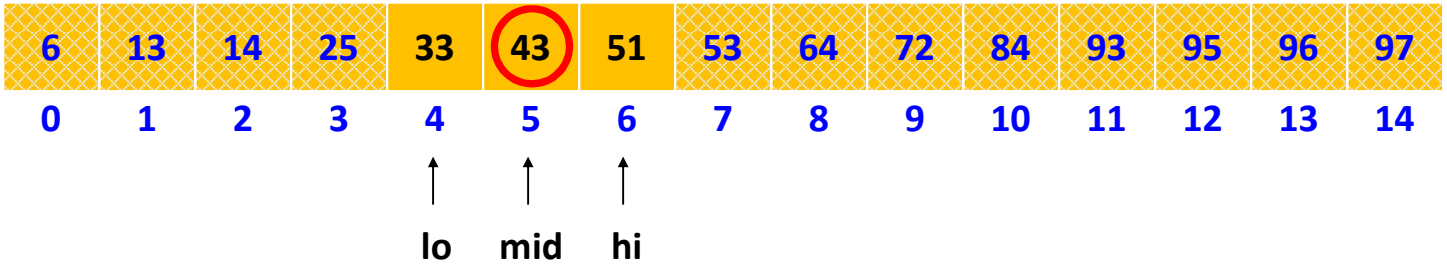
key=33



Exercise 1: Recursive Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       mid = (low + high) / 2;
       if (S[mid]==key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

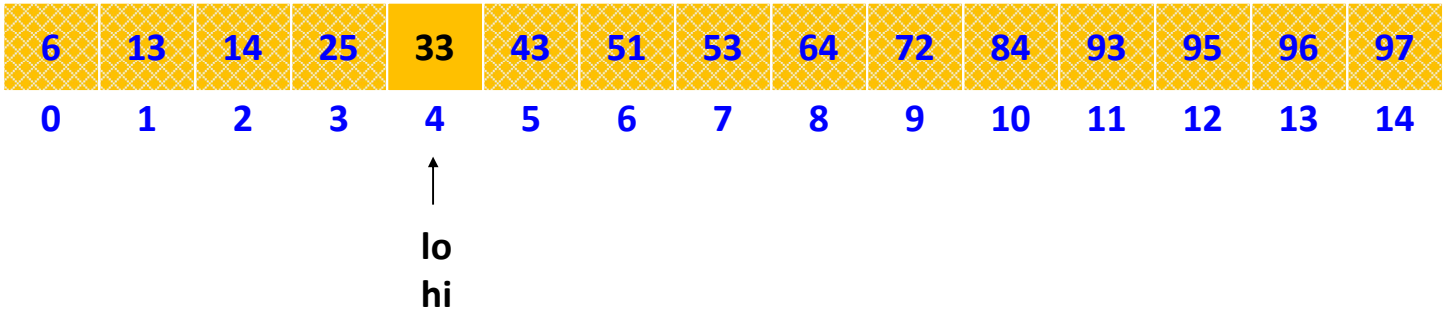
key=33



Exercise 1: Recursive Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       mid = (low + high) / 2;
       if (S[mid]==key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

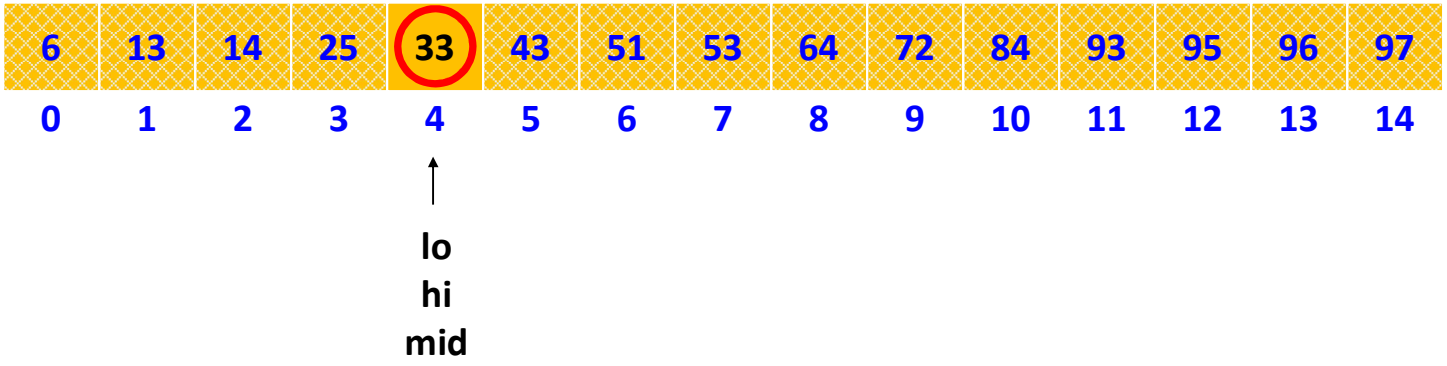
key=33



Exercise 1: Recursive Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       mid = (low + high) / 2;
       if (S[mid]==key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

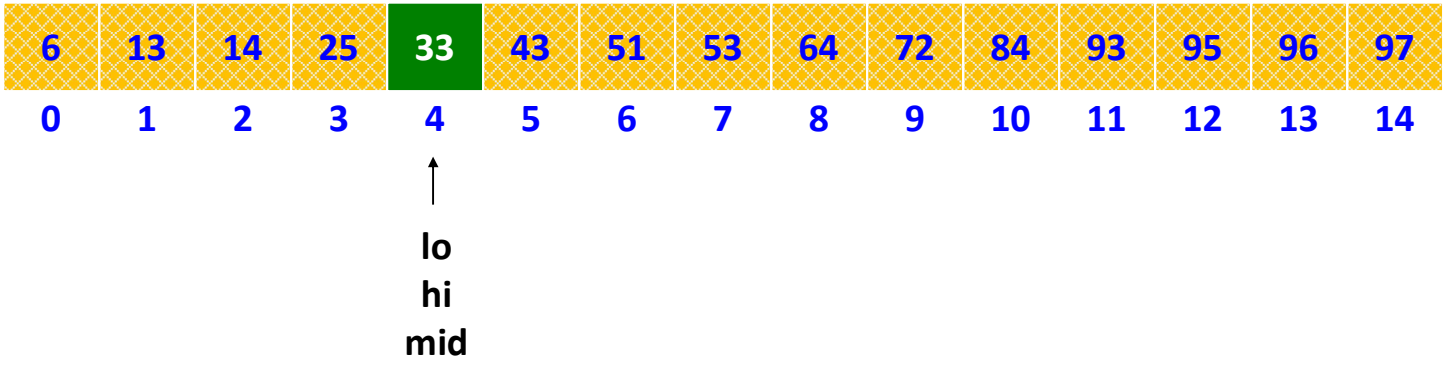


Exercise 1: Recursive Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       mid = (low + high) / 2;
       if (S[mid]==key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

key=31??



Exercise 1: Recursive Binary Search

Input: An array *S* consists of *n* elements: S[0],…,S[*n*-1] in ascending order; Value ***key*** with the same data type as array *S*.

Output: the index in array if ***key*** is found, -1 if ***key*** is not found

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       mid = (low + high) / 2;
       if (S[mid]==key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

➔ binsearch(0, *n*-1, *S*, *key*);

How many times is binsearch called in the worst case ?

Exercise 1: Recursive Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
     }
     else return -1;
}
```

➔ T(0) = ?

➔ binsearch(0, -1, *S*, *key*);

➔ T(1) = ?

➔ binsearch(0, 0, *S*, *key*);

➔ binsearch(0, -1, *S*, *key*);

➔ binsearch(1, 0, *S*, *key*);

➔ binsearch(0, *n*-1, *S*, *key*);

Let T(*n*): the number of times that binsearch is called in the worst case when array *S* has *n* elements

- $T(0) = 1$
- $T(1) = 2$
- $T(2) = T(1) + 1 = 3$
- $T(4) = T(2) + 1 = 4$
- $T(8) = T(4) + 1 = 4 + 1 = 5$
- $T(n) = T(n/2) + 1$

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

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Exercise 1: Recursive Binary Search

- Let T(n)= T(n/2) + 1. What are the parameters?

  a = 1

  b = 2

  d = 0

  Therefore, which condition applies?

  $1 = 2^0$, case 2 applies

- We conclude that $T(n) \epsilon \boldsymbol{\Theta}(n^d \log n) = \boldsymbol{\Theta}(\log n)$

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

  $T(n) = a\,T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c > 0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Exercise 1: Recursive Binary Search

Backward substitution: this works exactly as its name suggests. Starting from the equation itself, work backwards, substituting values of the function for previous ones

$T(n) = T(n/2) + 1$

where $T(1) = 2$

$T(n) = T(n/2) + 1$ substitute for $T(n/2)$

$= T(n/4) + 1 + 1$ substitute for $T(n/4)$

$= T(n/8) + 1 + 1 + 1$

$= T(n/2^3) + 3*1$ in more compact form

$= \ldots$

$= T(n/2^k) + k*1$ "inductive leap"

$T(n) = T(n/2^{\log n}) + \log n$ *"choose k = logn"*

$= T(n/n) + \log n$

$= T(1) + \log n = 2 + \log n$

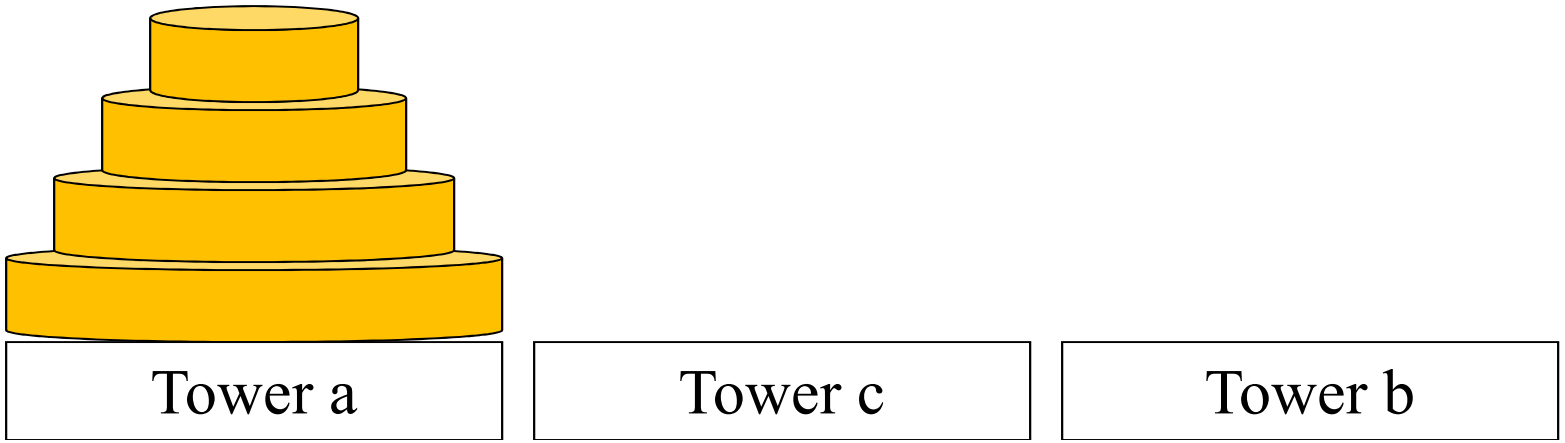
Exercise 2: Tower of Hanoi

The Tower of Hanoi, consists of three towers (a), (b), (c) together with n disks of different sizes. Initially these disks are stacked on the tower (a) in an ascending order, i.e. the smaller one sits over the larger one.

The objective of the game is to move all the disks from tower (a) to tower (c), following 3 rules:

- Only one disk can be moved at a time.
- Only the top disk can be moved
- No large disk can be sit over a smaller disk.

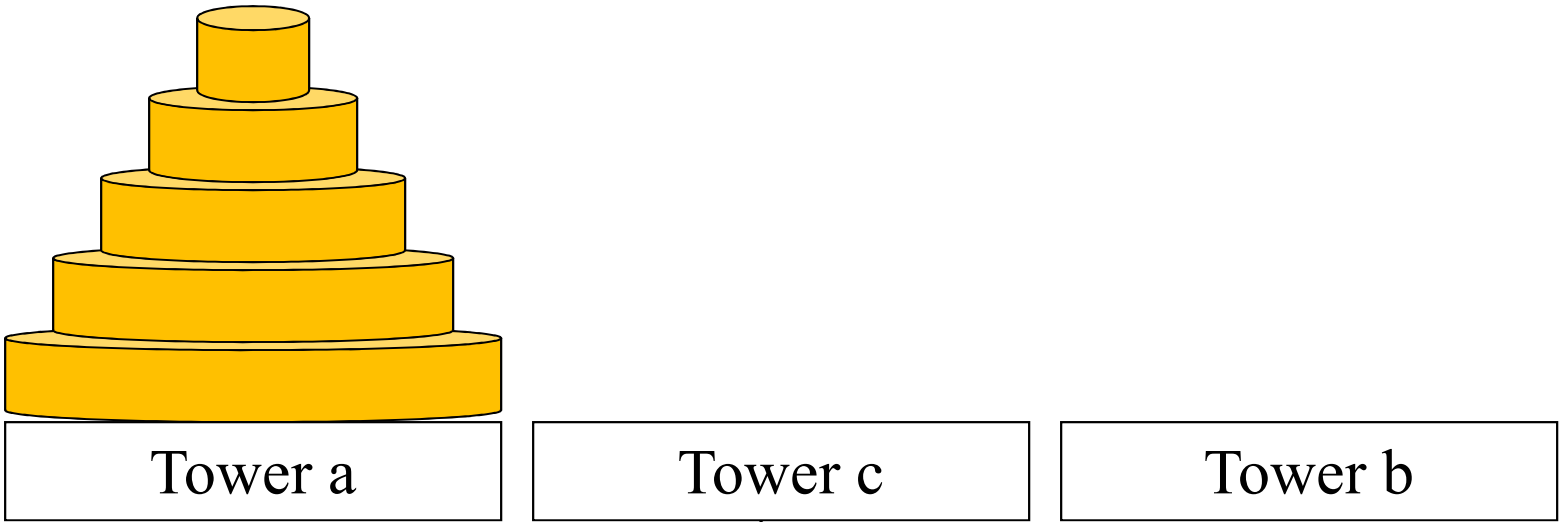
Let h_n denote the minimum number of moves needed to solve the Tower of Hanoi problem with n disks. What is the recurrence relation for h_n?



Tower of Hanoi: *n*=5

The objective of the game is to move all the disks from tower (a) to tower (c), following 3 rules:

1. Only one disk can be moved at a time.
2. Only the top disk can be moved
3. No large disk can be sit over a smaller disk.



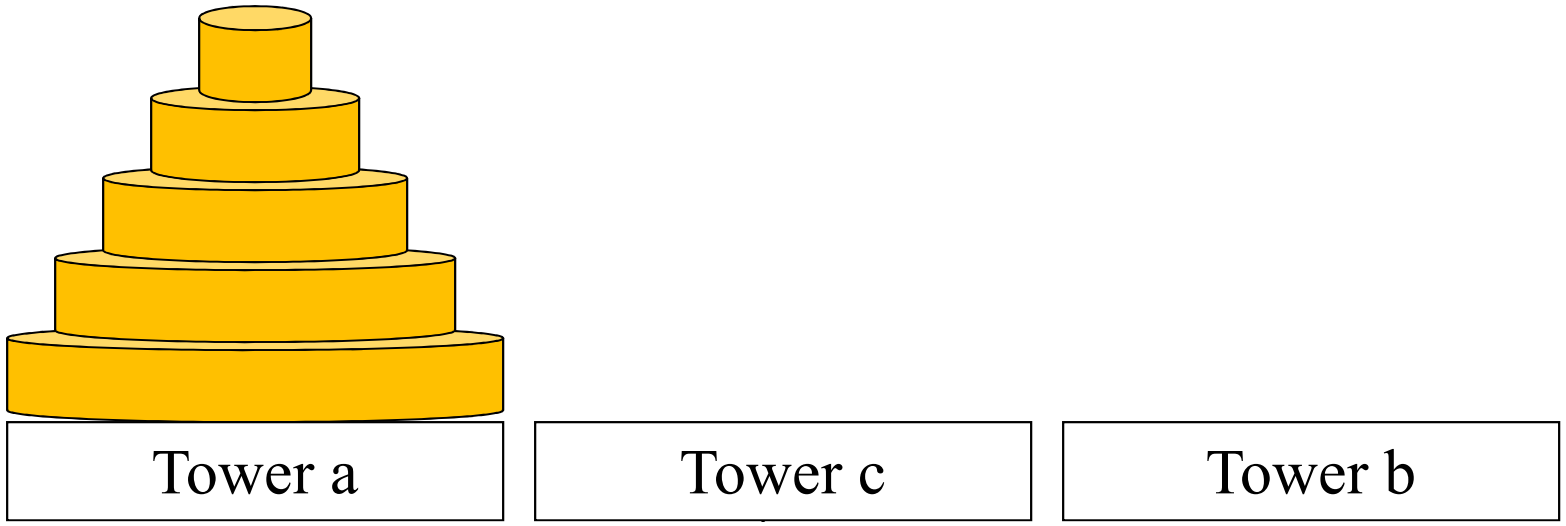
Exercise 2: Tower of Hanoi

- $h_1 = 1$
- For $n \geq 2$, we need to do 3 following steps to transfer all disks from tower (a) to tower (c):

  (1) Move the top n - 1 disks (following the rules of the game) from tower (a) to tower (b)

  (2) Move the largest disk to the tower (c)

  (3) Move the n-1 disks (following the rules of the game) from tower (b) to tower (c), placing them on top of the largest disk



Exercise 2: Tower of Hanoi

- $h_1 = 1$
- For $n \geq 2$, we need to do 3 following steps to transfer all disks from tower (a) to tower (c):

(1) Move the top n - 1 disks (following the rules of the game) from tower (a) to tower (b)

The problem of n-1 disks → #moves = h_{n-1}

(2) Move the largest disk to the tower (c)

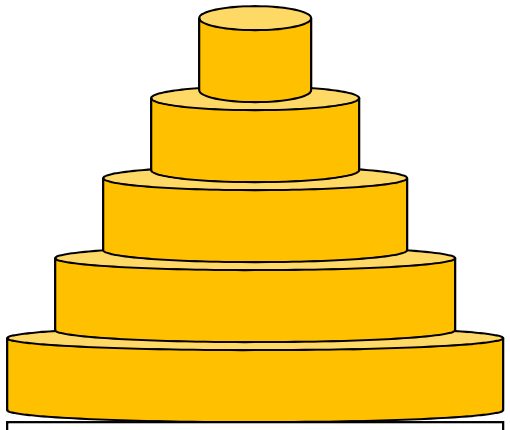
→ #moves = 1

(3) Move the n-1 disks (following the rules of the game) from tower (b) to tower (c), placing them on top of the largest disk

The problem of n-1 disks→ #moves = h_{n-1}

$$h_n = 2h_{n-1} + 1, n \geq 2$$

$$h_1 = 1$$



Tower of Hanoi: n=5

(1) Move the top n - 1 disks (following the rules of the game) from tower (a) to tower (b)

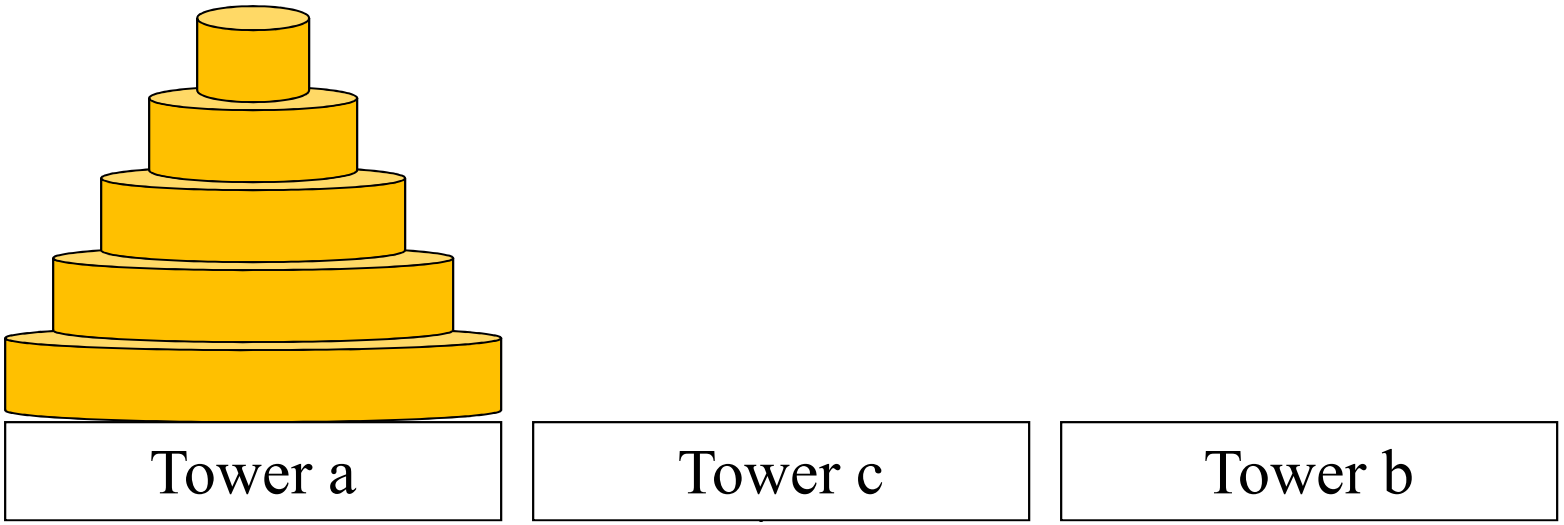
The problem of n-1 disks → #moves = h_{n-1}

(2) Move the largest disk to the tower (c)

→ #moves = 1

(3) Move the n-1 disks (following the rules of the game) from tower (b) to tower (c) , placing them on top of the largest disk

The problem of n-1 disks → #moves = h_{n-1}



Tower of Hanoi

The algorithm could be implemented as following:

//move *n* disks from tower *a* to tower *c* using tower *b* as an intermediary:

```
HanoiTower(n, a, c, b);
{
      if (n==1) then <move disk from tower a to tower c>
      else
      {
            HanoiTower(n-1,a,b,c);
            HanoiTower(1,a,c,b);
            HanoiTower(n-1,b,c,a);
      }
}
```

For $n \geq 2$, we need to do 3 following steps to transfer all disks from tower (a) to tower (c):

(1) Move the top n - 1 disks (following the rules of the game) from tower (a) to tower (b)

The problem of n-1 disks → #moves = h_{n-1}

(2) Move the largest disk to the tower (c)

→ #moves = 1

(3) Move the n-1 disks (following the rules of the game) from tower (b) to tower (c), placing them on top of the largest disk

The problem of n-1 disks→ #moves = h_{n-1}

Tower of Hanoi: Implementation

```
#include <bits/stdc++.h>
using namespace std;

void HanoiTower (int, char, char, char);
int i = 0;

int main()
{
    int n;
    cout<<" Input the number of disks = "; cin >>n;
    HanoiTower (n, 'a', 'c', 'b');
    cout <<"Total number of  disk movements = "<<i<<endl;
    return 0;
}

void HanoiTower (int n, char start, char finish, char spare)
{
    if (n == 1){
        cout<<" Move disk from tower "<<start<<" to tower "<<finish<<endl;
        i++;
        return;
    } else {
        HanoiTower (n-1, start, spare, finish);
        HanoiTower (1, start, finish, spare);
        HanoiTower (n-1, spare, finish, start);
    }
}
```

Exercise 2: Tower of Hanoi

- Let T(n) = 2T(n-1) + 1. What are the parameters?

  a =

  b =

  d =

$$h_n = 2\,h_{n-1} + 1$$

Therefore, which condition applies?

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

  $T(n) = a\,T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Exercise 2: Tower of Hanoi

Backward substitution: this works exactly as its name suggests. Starting from the equation itself, work backwards, substituting values of the function for previous ones

$$T(n) = 2\,T(n-1) + 1$$

$$= 2\,(2\,T(n-2) + 1) + 1 \quad = 2^2\,T(n-2) + 2 + 1$$

$$= 2^2(2\,T(n-3) + 1) + 2 + 1 \qquad = 2^3\,T(n-3) + 2^2 + 2 + 1$$

$$\ldots$$

$$= 2^{n-1}\,T(1) + 2^{n-2} + \ldots + 2 + 1$$

$$= 2^{n-1} + 2^{n-2} + \ldots + 2 + 1 \qquad \text{(because } T(1) = 1\text{)}$$

$$= 2^n - 1$$

➔ Time complexity: $O(2^n)$ which is exponential

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

History "Tower of Hanoi"

Legend: a group of Eastern monks are the keepers of three towers on which sit 64 golden rings. Originally all 64 rings were stacked on one tower with each ring smaller than the one above. The monks are to move the rings from this first tower to the third tower one at a time but never moving a larger ring on top of a smaller one. Once the 64 rings have all been moved, the world will come to an end.

➔ Number of moves = 2^{64}-1 = 18 446 744 073 709 551 615

➔ which would require more than *five billion centuries! (if assuming one move per second)*

It was invented by the French mathematician Eduoard Lucas in 1883

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Exercise 3: What's the runtime of this function?

```
int recursiveFun1(int n)
{
    if (n <= 0)
        return 1;
    else
        return 1 + recursiveFun1(n-1);
}
```

```
int recursiveFun2(int n)
{
    if (n <= 0)
        return 1;
    else
        return 1 + recursiveFun2(n-5);
}
```

```
int recursiveFun3(int n)
{
    if (n <= 0)
        return 1;
    else
        return 1 + recursiveFun3(n/5);
}
```

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

Exercise 3: What's the runtime of this function?

```
void recursiveFun4(int n, int m, int o)
{
    if (n <= 0)
    {
        printf("%d, %d\n",m, o);
    }
    else
    {
        recursiveFun4(n-1, m+1, o);
        recursiveFun4(n-1, m, o+1);
    }
}

int recursiveFun5(int n)
{
    for (i = 0; i < n; i += 2) {
        // do something
    }

    if (n <= 0)
        return 1;
    else
        return 1 + recursiveFun5(n-5);
}
```

NGUYỄN KHÁNH PHƯƠNG
Bộ môn KHMT – ĐHBK HN

QUESTIONS ???