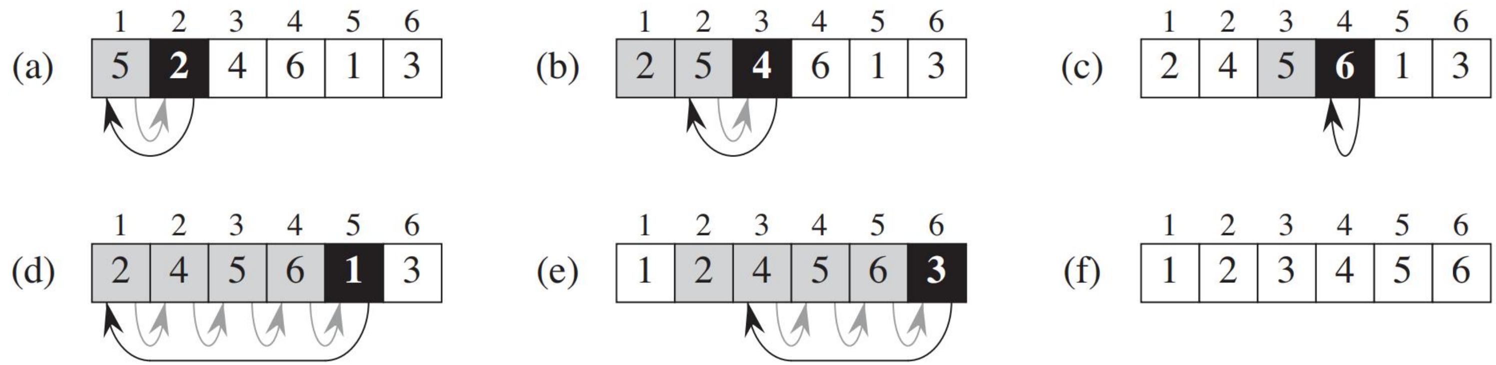
Review of Last Lecture

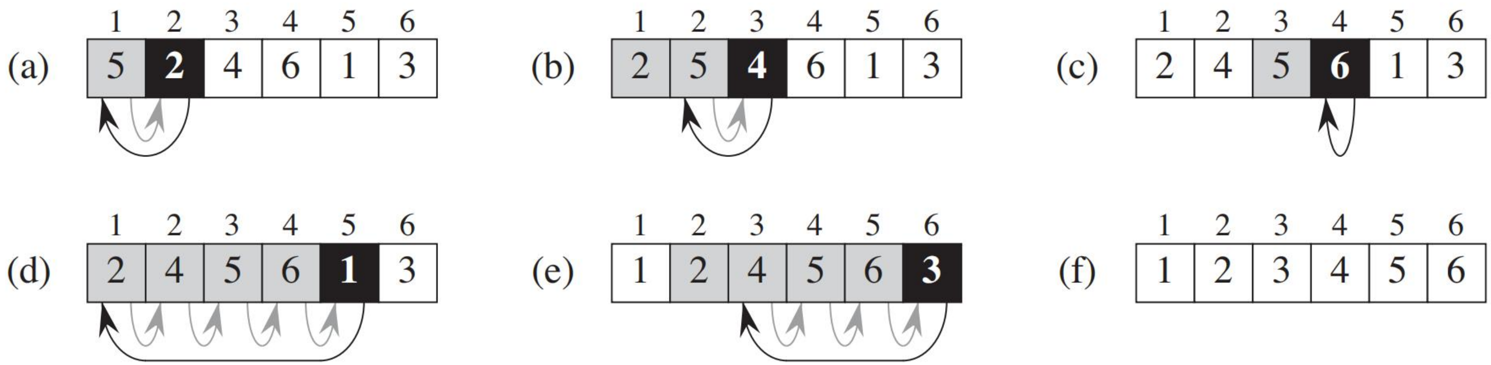
Insertion Sort



INSERTION-SORT(A)

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted
           sequence A[1 .. j - 1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Running Time of Insertion Sort



INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

Running Time of Insertion Sort

- Running time:
  - Best case: $pn+q$
  - Worst case: an^2+bn+c

- The extra precision is not usually worth the effort of computing it.

- For large enough inputs n, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself, *eg* $O(an^2+bn+c) = O(n^2)$ for $a>0$.

Comparison of Functions

• Function:	ω	Ω	Θ	O	o
• Real number:	$>$	$\geq$	$=$	$\leq$	$<$

Theorem 3.1

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 $f(n) = \Omega(g(n))$. ■

Running Time of Insertion Sort

- Running time $T(n)$:
  - Best case: $pn+q$
  - Worst case: an^2+bn+c

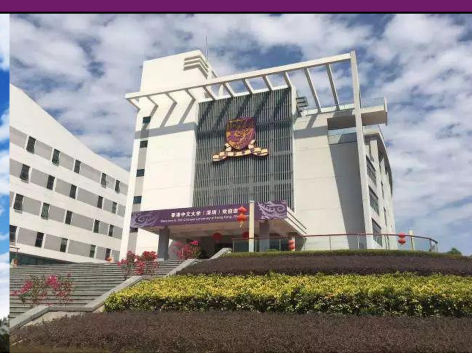
- $T(n) = \Theta(n^2)$ in the worst case
- $T(n) \neq \Theta(n^2)$ in general

- $T(n) = \Omega(n)$ in the best case
- $T(n) = \Omega(n)$ in general

- $T(n) = O(n^2)$ in the worst case
- $T(n) = O(n^2)$ in general

CSC3100: Designing Algorithms

Kaiming Shen

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

Recursion

Recursion

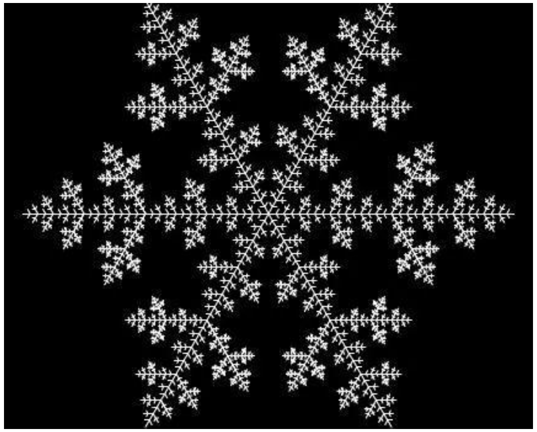
What is recursion?

- Self-reference
- Recursive function: based upon itself
- Solution of the whole problem is composed of solutions of sub-problems

$$f(x) = 2f(x-1) + x^2$$

```
int f( int x ){
        if ( x == 0 )
                return 0;
        else
                return 2 * f( x – 1 ) + x*x;
}
```

Recursion



Recursion

Characteristics of a recursive definition:

- It has a stopping point. (Base case)
- It recursively evaluate an expression with a variable *n* monotonically decreasing.
- Base case must be reached.

```
fun (N)
{
  if N == 0
      return 0;
  else
      return fun (N-1) + N - 1;
}
```

Fibonacci Numbers

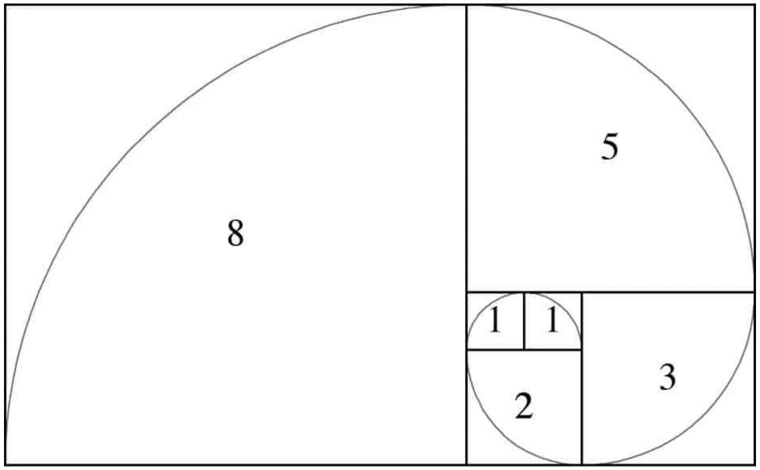
$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$ for $n>1$

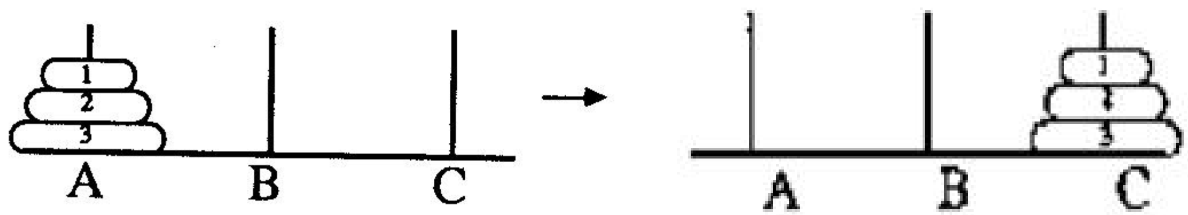
e.g., $F_2 = 1+0 = 1$, $F_3 = 1+1 = 2$, $F_4 = 2+1 = 3$, ...

```
Fibonacci (N)
{
   if N == 0
         return 0;
   else if N == 1
         return 1
   else
         return Fibonacci (N-1) + Fibonacci (N-2)
}
```



Tower of Hanoi

Target: Move all disks from peg A to peg C.



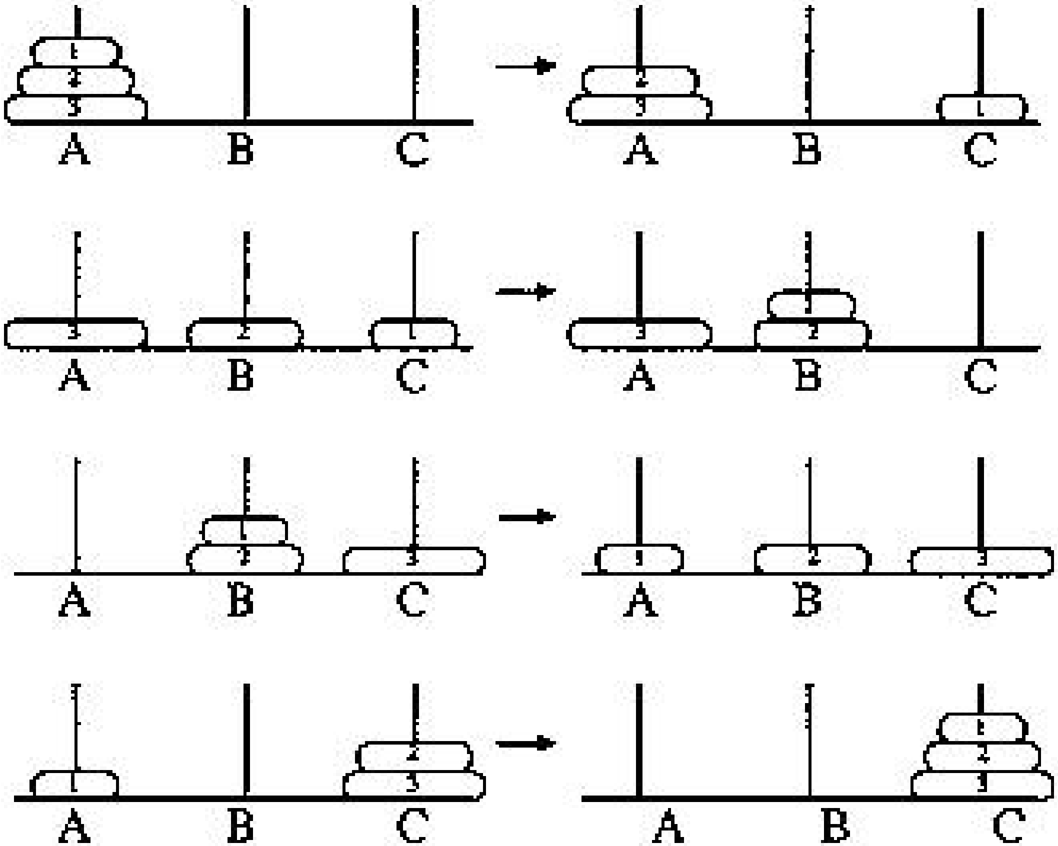
Constraints:

(1) only one disk can be moved at a time

(2) at no time may a disk be placed on top of a smaller disk.

Tower of Hanoi

When we have only three disks, i.e., N=3



Tower of Hanoi

Solution:

- If n = 1, move the single disk from A to C and stop; (base case)
- Otherwise, move the top n-1 disks from A to B, using C as auxiliary; (recursive case)
- Move the remaining disk from A to C;
- Move the n-1 disks from B to C, using A as auxiliary.

Tower of Hanoi

```
Hanoi (n, A, C, B)
    if n == 1// If only one disk, make the move and return
        move remaining disk from A to C;
        return;
    else
/* move top n-1 disks from A to B, with C as auxiliary*/
        Hanoi (n-1, A, B, C);
/* move remaining disk from A to C */
        move remaining disk from A to C;
/* move n-1 disks from B to C, A as auxiliary */
        Hanoi (n-1, B, C, A);
```

Evaluating Recurrence

Mathematical Induction

The Principle of Mathematical Induction

Suppose that for each natural number n, we have a statement P_n for which the following two conditions hold:

1. P_1 is true.
2. For each natural number k, if P_k is true, then P_{k+1} is true.

Then all of the statements are true; that is, P_n is true for all natural numbers n.

Mathematical Induction

Let P_n denote the statement that $1 + 3 + 5 + \cdots + (2n - 1) = n^2$.
Then we want to show that P_n is true for all natural numbers n.

Step 1 We must check that P_1 is true. But P_1 is just the statement that $1 = 1^2$, which is true.
Step 2 Assuming that P_k is true, we must show that P_{k+1} is true.
Thus we assume that

That is the induction hypothesis. We must now show that

To derive equation (2) from equation (1), we add the quantity [2(k+1)-1] to both sides of equation (1).

$$\begin{aligned} 1 + 3 + 5 + \cdots + (2k - 1) + [2(k + 1) - 1] &= k^2 + [2(k + 1) - 1] \\ &= k^2 + 2k + 1 \\ &= (k + 1)^2 \end{aligned}$$

That is, $1 + 3 + 5 + \cdots + (2k + 1) + [2(k + 1) - 1] = (k + 1)^2$. So P_{k+1} is true.

Substitution Method

Step 1: Guess the running time T(n).

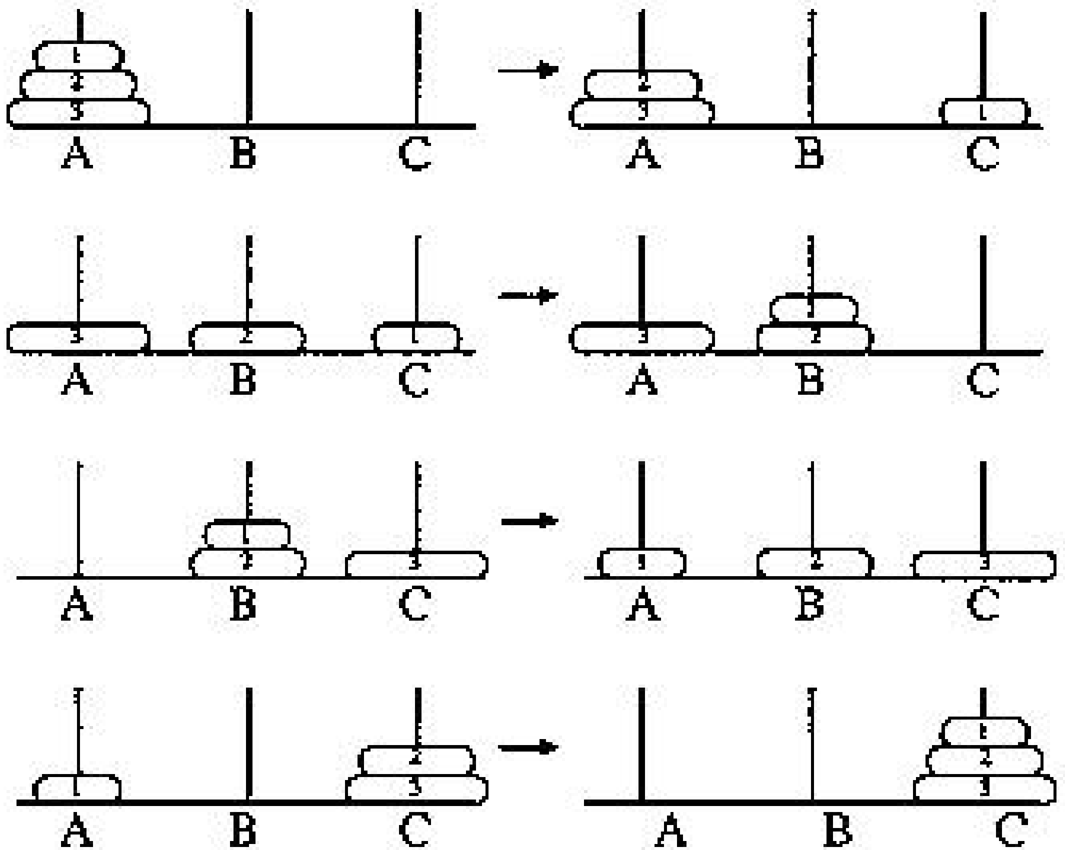
Step 2: Verify guess via mathematical induction

- Show that T(1).
- Show that T(n) is correct if T(n-1) is correct.
- Or, show that T(n) is correct if T(m) is correct for all m<n.

For the tower of Hanoi problem, the input size n is the number of disks to move.

Substitution Method

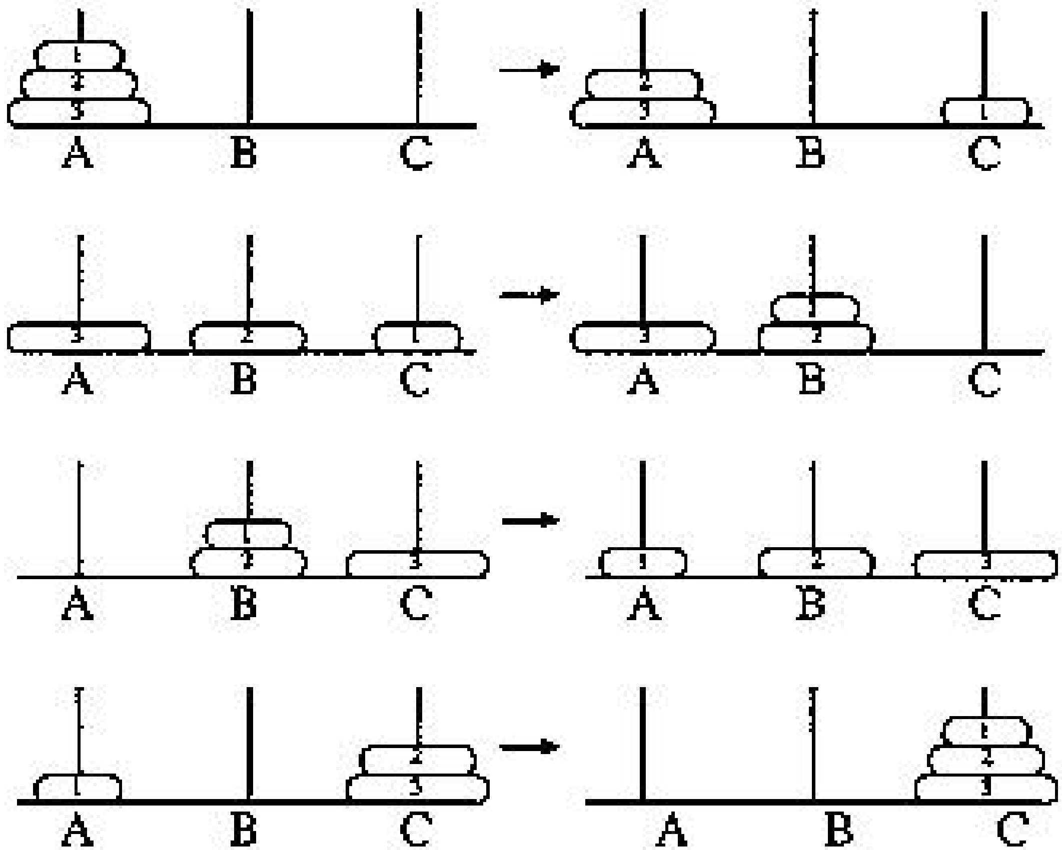
$T(1) = 1$, $T(2) = 3$, $T(3) = 7$, $T(n) = ?$



Substitution Method

T(1) = 2^1-1; T(2) = 2^2-1, T(3) = 2^3-1,

T(n) = 2^n-1?



Substitution Method

Solution:

- If n = 1, move the single disk from A to C and stop;
- Otherwise, move the top n-1 disks from A to B, using C as auxiliary; T(n-1) moves
- Move the remaining disk from A to C; 1 move
- Move the n-1 disks from B to C, using A as auxiliary. T(n-1) moves

Hence, $T(n) = 2*T(n-1)+1$

Substitution Method

- Guess $T(n) = 2^n-1$
- Check $2^1-1 = 1$ is indeed # of moves if $n=1$
- Assume $2^{(n-1)}-1$ is # of moves given $n-1$ disks
- Prove that 2^n-1 is # of moves given n disks
- Proof:
  - $T(n-1) = 2^{(n-1)}-1$
  - $T(n) = 2*T(n-1)+1 = 2*2^{(n-1)}-2+1 = 2^n-1$

Recursion-Tree Method

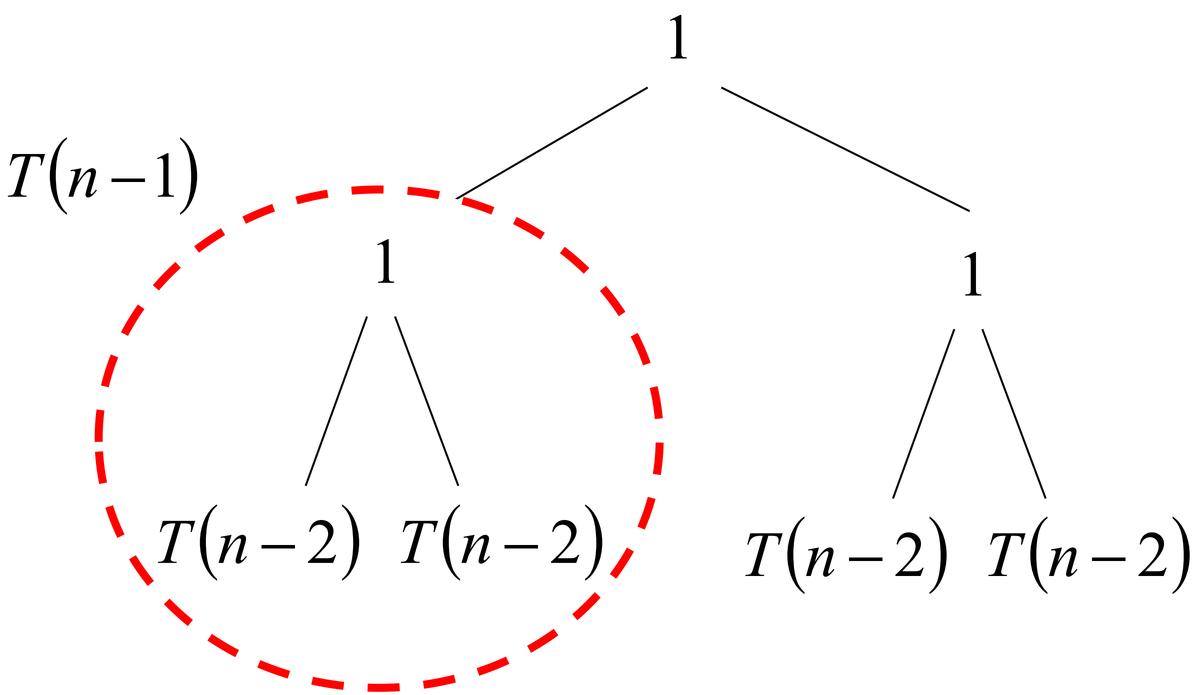
- We aim to visualize the iterations
- T(n) = 2*T(n-1)+1

$$1$$

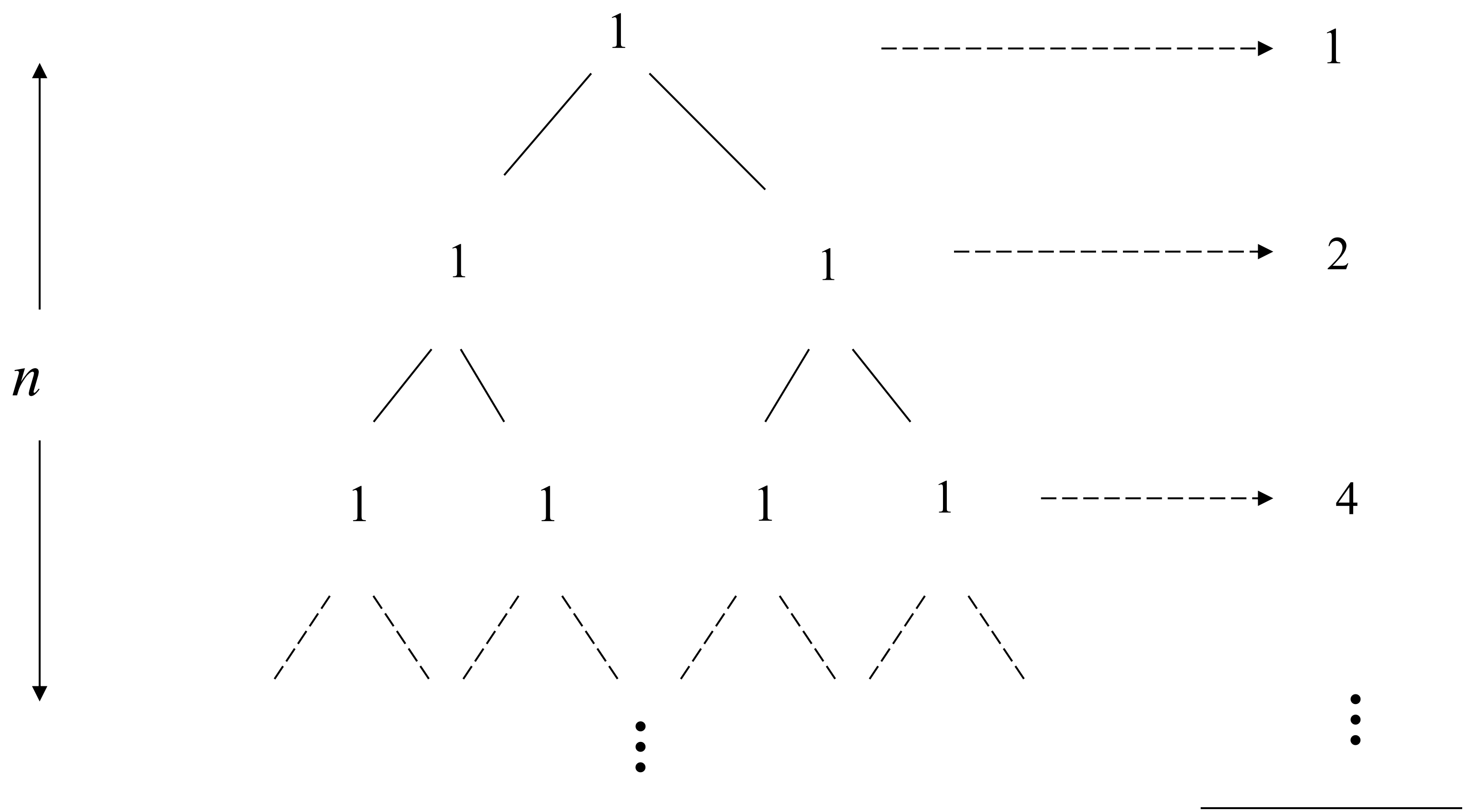
$$T(n-1) \qquad\qquad T(n-1)$$

Recursion-Tree Method

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



Recursion-Tree Method



Total: $1 + 2 + 2^2 + 2^3 + \ldots + 2^{n-1}$

$= 2^n - 1$

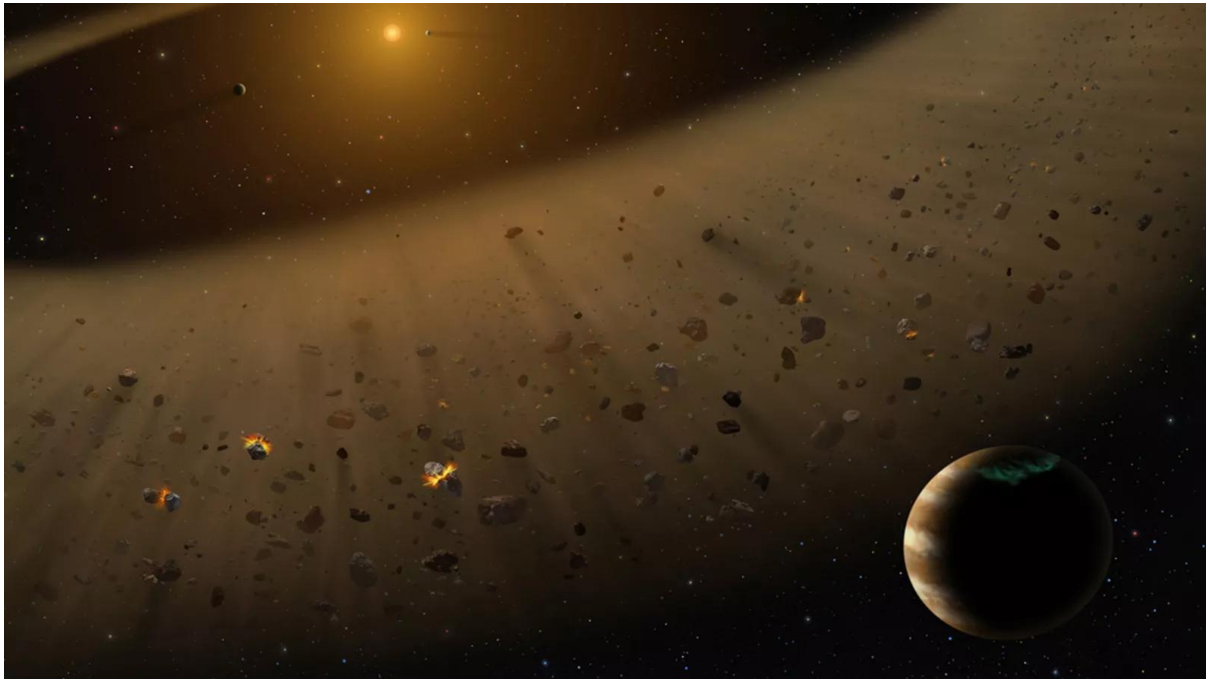
Tower of Hanoi

According to a legend of obscure origin, there exists an ancient temple where priests have been shuffling **64** golden disks between three pegs for many centuries. When the priests finally succeed in transferring all of the disks, **the world will end**.

source: https://psychology.wikia.org/wiki/Tower_of_Hanoi

Tower of Hanoi

- It requires 2^64-1 moves
- Suppose each move consumes 1 sec
- Lifespan of sun is about 10 billion years
- 10 billion years << 2^64-1 sec



Merge Sort

Divide-and-Conquer

Divide the problem into a number of subproblems

Conquer the subproblems by solving them recursively (further divide if not small enough).

- **Recursive case:** subproblems are still large;
- **Base case**: If the subproblems are small enough, may solve them by brute force.

Combine the subproblem solutions to give a solution to the original problem.

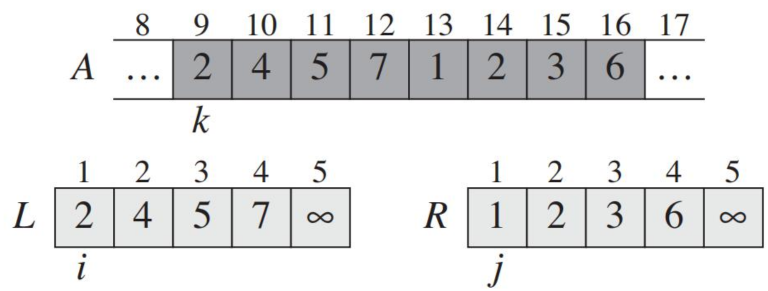
Merge Sort

- A sorting algorithm based on divide and conquer.
- The worst-case running time of Merge Sort is $\Theta(n \lg n)$ whereas that of Insertion Sort is $\Theta(n^2)$.
- Each subproblem is to sort a subarray A[p,...,r].
- Set p=1, r=n at the beginning. (Original problem)

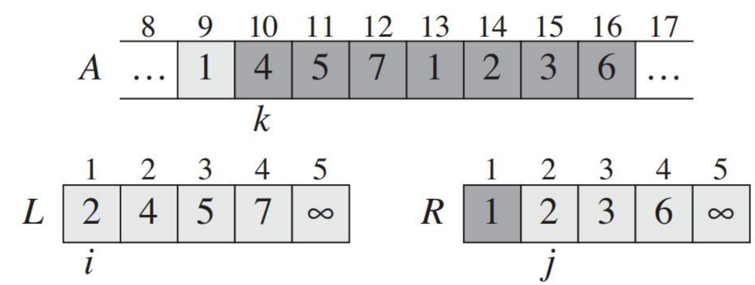
Merge Sort

- Divide it into two subarrays A[p,…,q] and A[q+1,…,r], where q is the midpoint.
- Conquer by recursively sorting the two subarrays A[p,…,q] and A[q+1,…,r].
- Merge the two sorted subarrays A[p,…,q] and A[q+1,…,r].

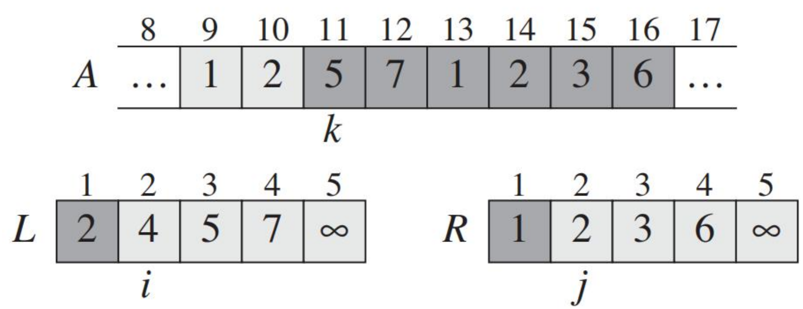
Merge Sort



(a)

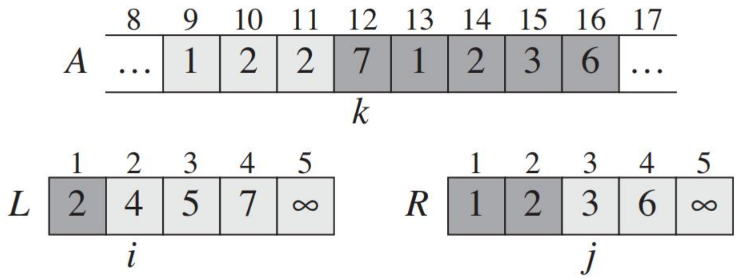


(b)

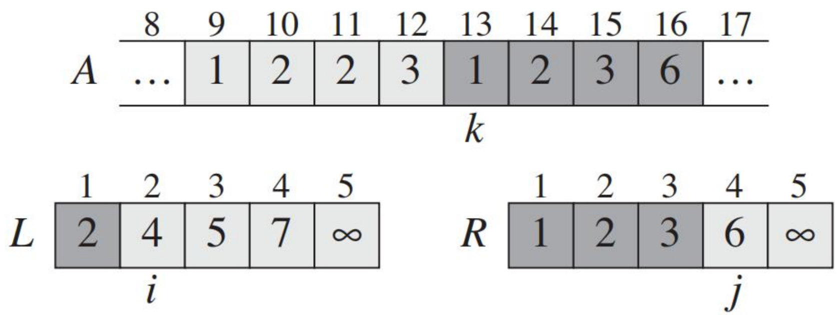


(c)

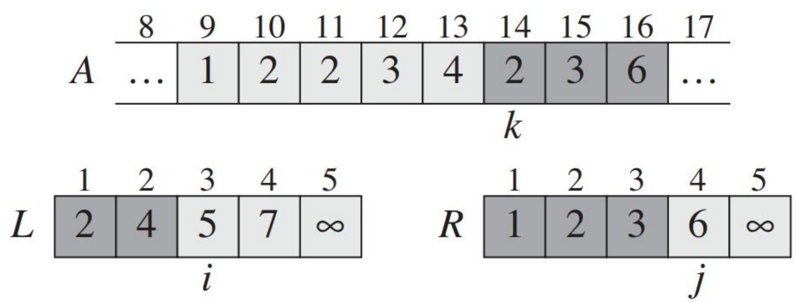
Merge Sort



(d)

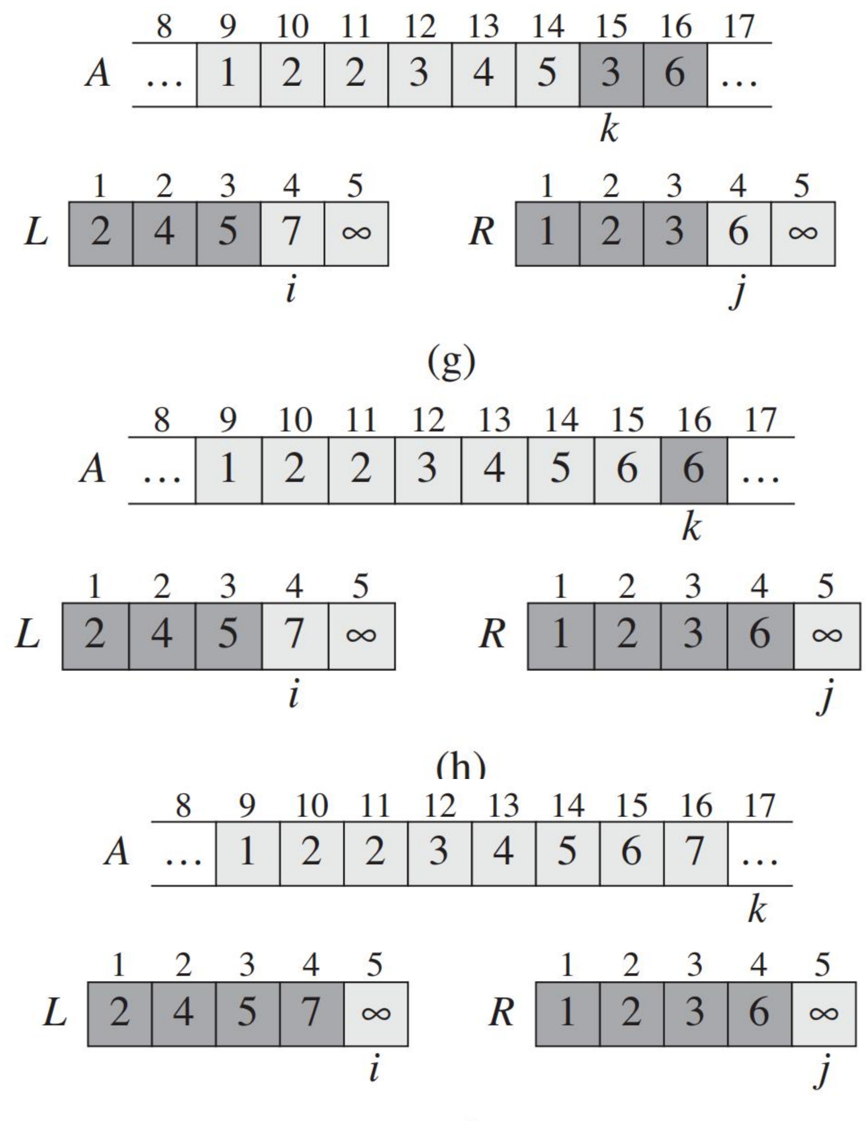


(e)



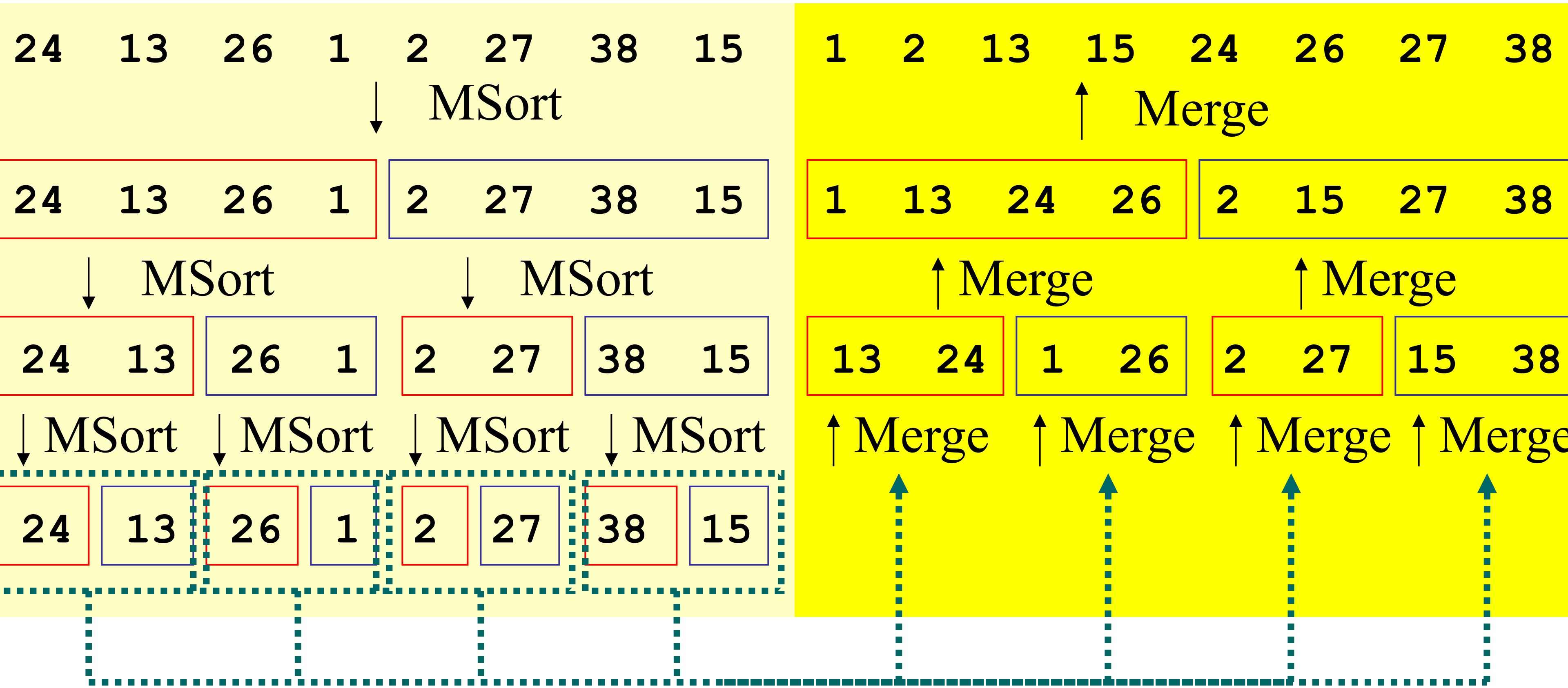
(f)

Merge Sort



(g) (h) (i)

Merge Sort



24 13 26 1 2 27 38 15
MSort
24 13 26 1
2 27 38 15
MSort
MSort
24 13
26 1
2 27
38 15
MSort
MSort
MSort
MSort
24
13
26
1
2
27
38
15
1 2 13 15 24 26 27 38
Merge
1 13 24 26
2 15 27 38
Merge
Merge
13 24
1 26
2 27
15 38
Merge
Merge
Merge
Merge

Merge Sort

```
MERGE-SORT(A, p, r)
if p < r                                 ▷ Check for base case
   then q ← ⌊(p + r)/2⌋                  ▷ Divide
        MERGE-SORT(A, p, q)              ▷ Conquer
        MERGE-SORT(A, q + 1, r)          ▷ Conquer
        MERGE(A, p, q, r)                ▷ Combine
```

Merge Sort

```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

Running Time of Merge Sort

MERGE-SORT(A, p, r)

```
if p < r                                ▷ Check for base case
  then q ← ⌊(p + r)/2⌋                  ▷ Divide
       MERGE-SORT(A, p, q)              ▷ Conquer
       MERGE-SORT(A, q + 1, r)          ▷ Conquer
       MERGE(A, p, q, r)                ▷ Combine
```

- Let f(n) be complexity of Merge-Sort(A,p,r) where |r-q|+1 = n.
- Two Conquer steps require f(n/2) each.
- Combine step requires $\Theta(n)$. (Why?)
- Thus, $T(n) = 2*T(n/2) + \Theta(n)$.

Substitution Method

- Guess $T(n) = \Theta(n\log(n))$
- Check for n=2: $T(2) = \Theta(2)$ (Why?)
- Assume $T(m) = \Theta(m\log(m))$ is true for any $m < n$
- Prove that $T(n)$ also holds true
- Proof:
  - Let $m=n/2$; assume $T(m) = \Theta(n/2*\log(n/2))$
  - $T(n) = 2*T(n/2)+ \Theta(n) = \Theta(n*\log(n/2))+ \Theta(n)$
    $= \Theta(n*\log(n)-n)+ \Theta(n) = \Theta(n\log(n))$

Subtle Mistake

- Guess $T(n) = \Theta(n)$ (this is a wrong guess!)
- Assume $T(m) = \Theta(m)$ any $m < n$
- Prove that $T(n)$ also holds true
- Fake Proof:
  - Let $m=n/2$, so $T(m) = \Theta(n/2)$
  - $T(n) = 2*T(n/2) + \Theta(n) = \Theta(n) + \Theta(n) = \Theta(n)$

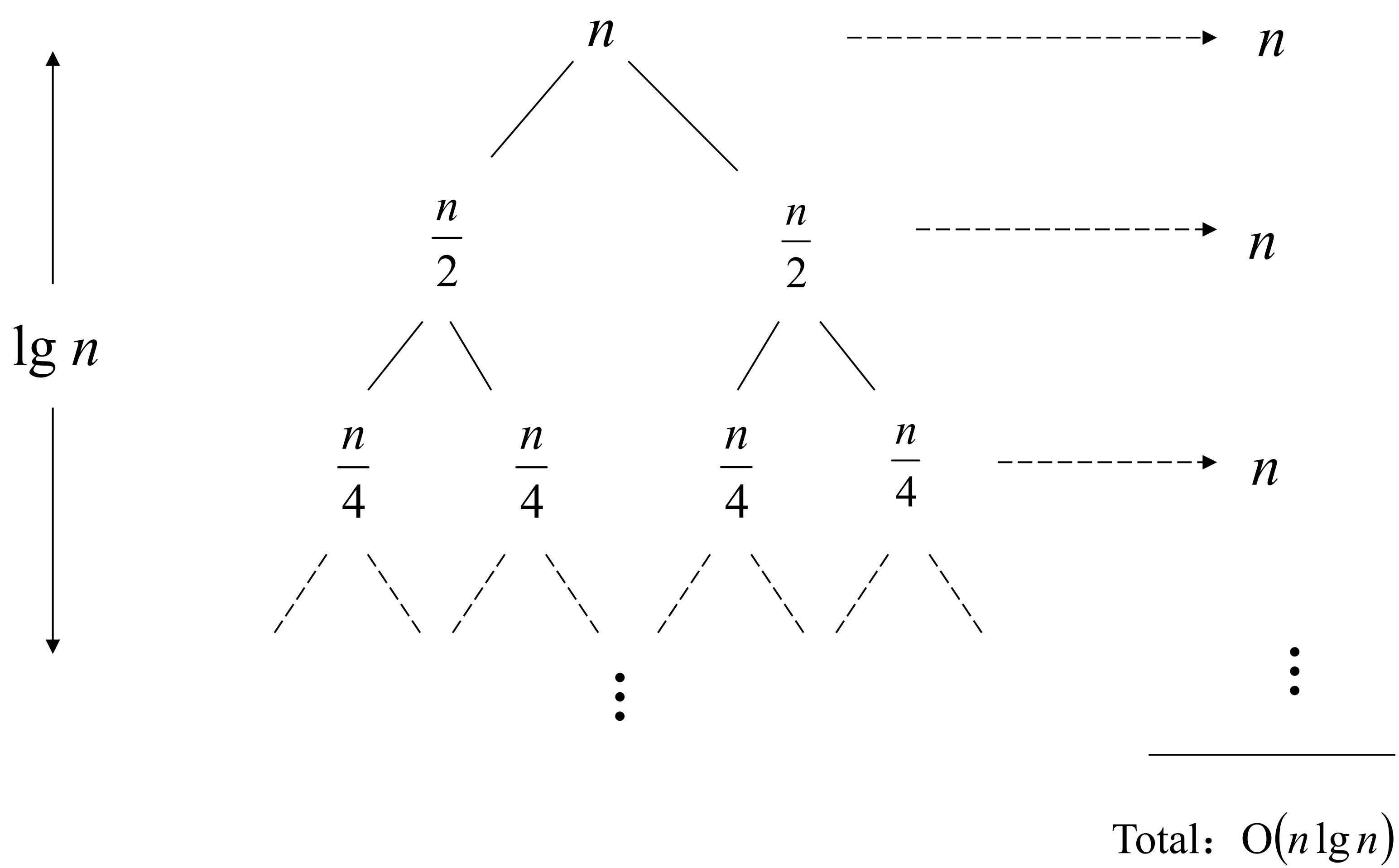
What is wrong?

Subtle Mistake

- Recall that $T(n) = \Theta(n)$ iff there exist constants c_1 and c_2 such that $c_1 n \le T(n) \le c_2 n$ when n is large.

- Fake Proof:
  - Let $m=n/2$, so $T(m) = \Theta(n/2)$
  - $c_1 n/2 \le T(n/2) \le c_2 n/2$
  - $T(n) = 2*T(n/2)+ \Theta(n)$ suggests ...
  - $2c_1 n \le T(n) \le 2c_2 n$
  - But we wish to show $c_1 n \le T(n) \le c_2 n$

Recursion-Tree Method



Master Theorem

$$T(n) = aT(n/b) + f(n)$$

CASE 1: If $f(n) = O(n^{\log_b a-\varepsilon})$, then $T(n) = O(n^{\log_b a})$

CASE 2: If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$

CASE 3: If $f(n) = \Omega(n^{\log_b a+\varepsilon})$ and $af(n/b) \leq cf(n)$, then $T(n) = \Theta(f(n))$

Merge sort: $a = 2$, $b = 2 \Rightarrow n^{\log_b a} = n$

$\Rightarrow$ CASE 2 $T(n) = \Theta(n \lg n)$.

Sketch of Proof

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ aT(n/b) + f(n) & \text{if } n = b^i\,, \end{cases}$$

T(n) = a*T(n-1) + f(b^i)
T(n-1) = a*T(n-2) + f(b^(i-1))
...
T(b) = a*T(1) + f(b)

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)\,.$$

Case I: If $f(n) = O(n^{\log_b a-\varepsilon})$

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) .$$

$$\begin{aligned}
\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} &= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} (b^\epsilon)^j \\
&= n^{\log_b a-\epsilon} \left(\frac{b^{\epsilon \log_b n} - 1}{b^\epsilon - 1}\right) \\
&= O(n^{\log_b a})
\end{aligned}$$

Case II: If $f(n) = \Theta(n^{\log_b a})$

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) .$$

$$\begin{aligned}
\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} &= n^{\log_b a} \sum_{j=0}^{\log_b n - 1} \left(\frac{a}{b^{\log_b a}}\right)^j \\
&= n^{\log_b a} \sum_{j=0}^{\log_b n - 1} 1 \\
&= n^{\log_b a} \log_b n .
\end{aligned}$$

Case III: If $f(n) = \Omega(n^{\log_b a+\varepsilon})$

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \, .$$

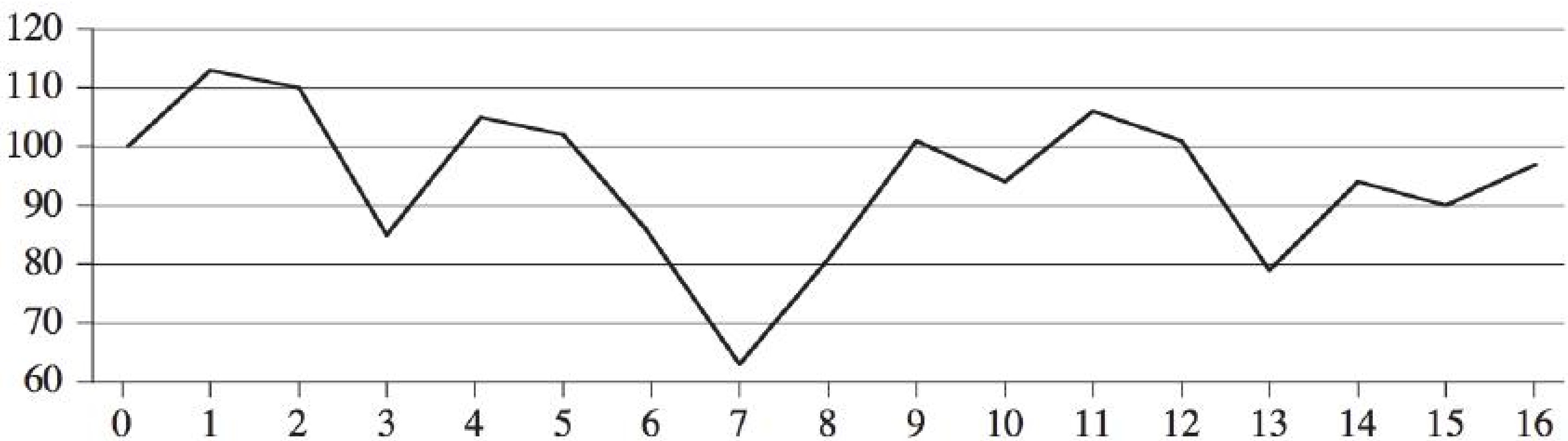
- Clearly, $T(n) = \Omega(f(n))$ (consider j=0)
- $af(n/b) \leq cf(n)$ => $a^j f(n/b^j) \leq c^j f(n)$,

$$\begin{aligned}
& \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \\
\leq\ & \sum_{j=0}^{\log_b n-1} c^j f(n) + O(1) \\
\leq\ & f(n) \sum_{j=0}^{\infty} c^j + O(1) \\
=\ & f(n) \left(\frac{1}{1-c}\right) + O(1) \\
=\ & O(f(n)) \, ,
\end{aligned}$$

Maximum-Subarray Problem

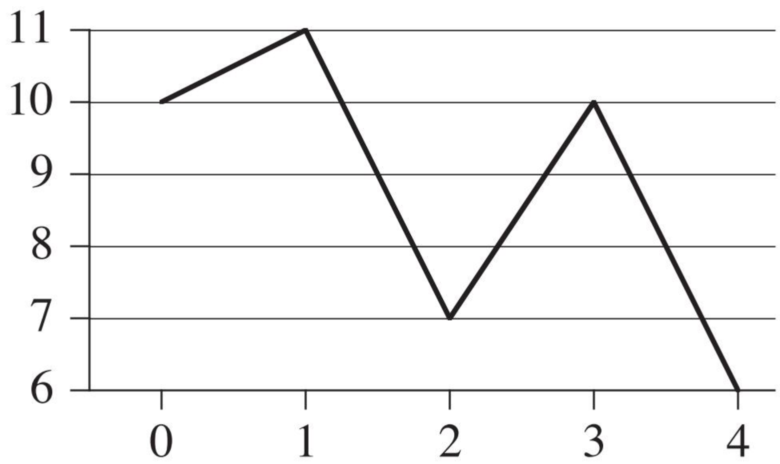
Stock Buying and Selling

- Profit = selling price - buying price
- You are allowed to buy a unit of stock only one time and then sell it at a later date.



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97

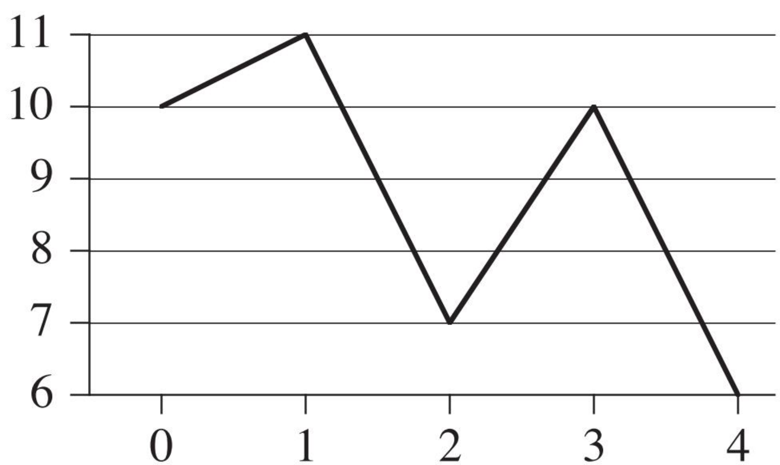
Two Naive Methods



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

- You are not able to buy lowest & sell highest.
- Method 1: Buy at the lowest price.
- Method 2: Sell at the highest price.
- Optimal: Buy at $7 and sell at $10.

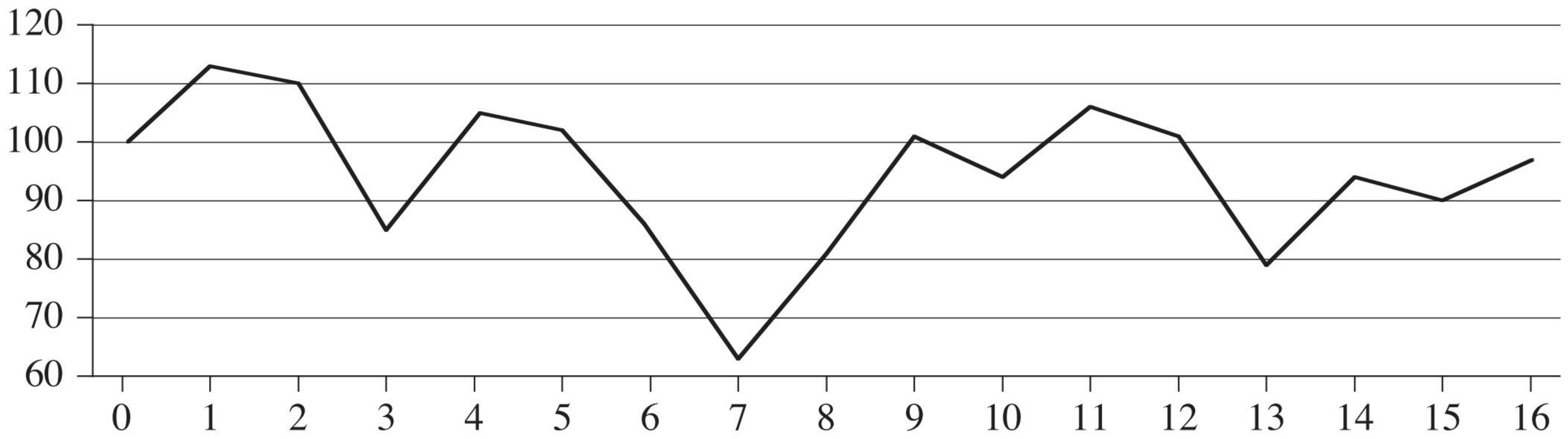
Brute-Force Method



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

- Evaluate all possible date pairs (a,b) where $a<b$
- (0,1) (0,2) (0,3) (0,4) (1,2) (1,3) (1,4) (2,3) (2,4) (3,4)
- Running time = $1+2+\ldots+(n-1) = \Theta(n^2)$

Transformation



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- Consider the change in price from the previous day.
- Profit = net change from the first day to last day.
- Max profit = find a subarray over which net change is maximized.

Max-Subarray Problem

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

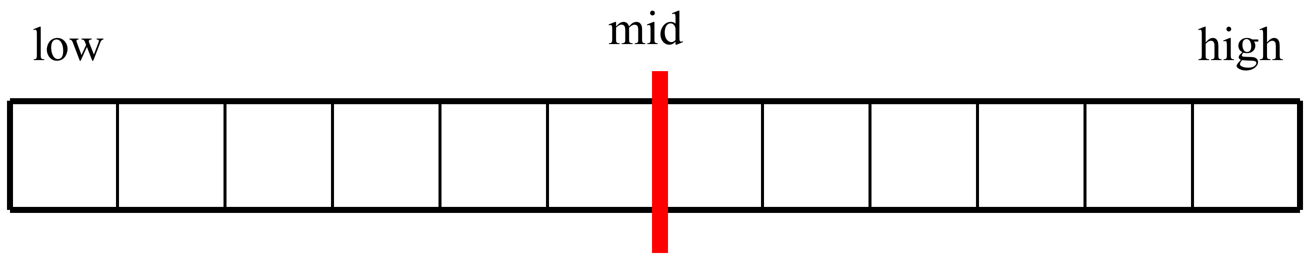
maximum subarray

- We seek the nonempty continuous subarray of A with the largest sum, namely *maximum subarray*.

- Q: How many subarrays are there if A.length=m?
- A: Each subarray defined by (first,last) where first ≠ last, so there are $\binom{m}{2}$ subarrays in total, i.e., $\Theta(m^2)$.

- Since m = n-1, $\Theta(m^2) = \Theta((n-1)^2) = \Theta(n^2)$

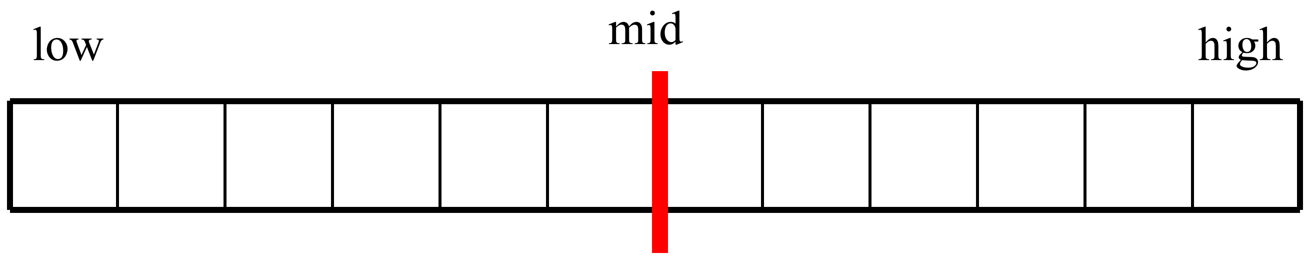
Max-Subarray Problem



- Divide the array [low,...,high] into two equal parts.
  - mid = n/2
  - Left part [low,...,mid]
  - Right part [mid+1,...,high]

- Maximum subarray must lie in one of 3 places:
  - Entirely in left part
  - Entirely in right part
  - Crossing the midpoint

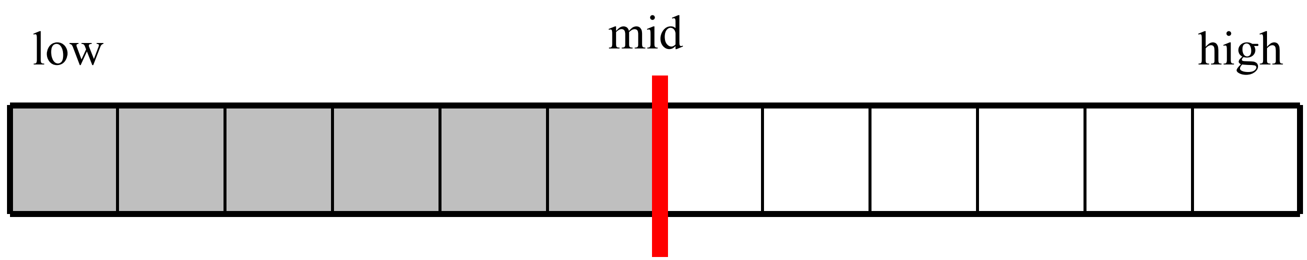
Max-Subarray Problem



Main idea:

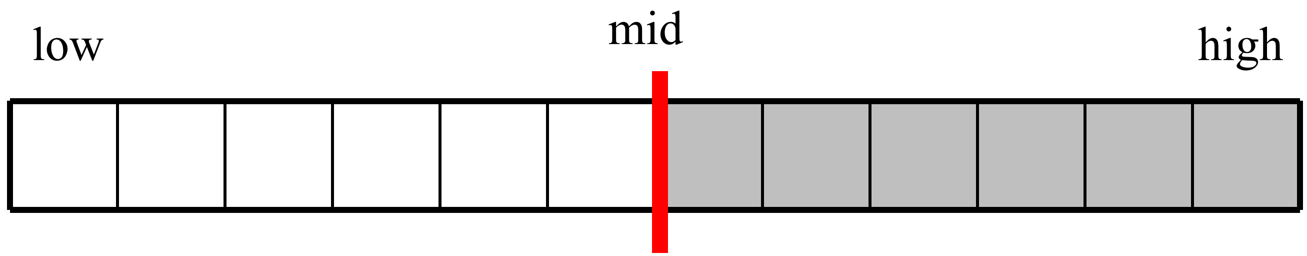
- Find the “max subarray” A_{left} assuming that it entirely lies in the left part.
- Find the “max subarray” A_{right} assuming that it entirely lies in the right part.
- Find the “max subarray” A_{mid} assuming that it crosses the midpoint.
- Obtain the real max subarray by comparing the above possible “max subarrays”.

Max-Subarray in the Left Case



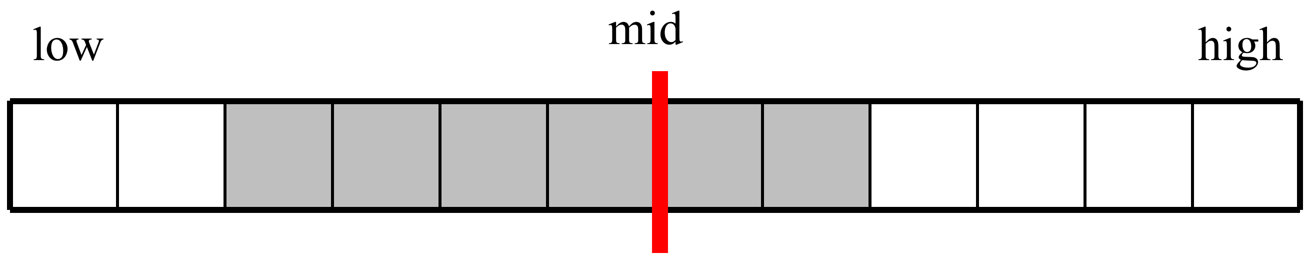
- A_{left} entirely lies in the left part [low,...,mid].
- This is just a smaller instance of our original problem.
- Original: Max-Subarray(low,high)
- Subproblem of A_{left}: Max-Subarray(low,mid)

Max-Subarray in the Right Part



- A_{right} entirely lies in the left part [mid+1,…,high].
- This is just a smaller instance of our original problem.
- Original: Max-Subarray(low,high)
- Subproblem of A_{right}: Max-Subarray(mid+1,high)

Max-Subarray in the Mid Case



- A_{mid} = [a,...,b] crosses the midpoint
- So low ≤ a ≤ mid, mid+1 ≤ b ≤ high
- This is quite different from the original problem, so we cannot call Max-Subarray recursively.
- Observe that we can optimize a and b separately, each done in linear time.

Max-Subarray in the Mid Case

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

Combination

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])              // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Divide and Conquer

- **Divide** the subarray into two subarrays. Find the midpoint *mid* of the subarrays, and consider the subarrays A[*low* ..*mid*] And A[*mid* +1..*high*]
- **Conquer** by finding a maximum subarrays of A[*low* ..*mid*] and A[*mid*+1..*high*].
- **Combine** by finding a maximum subarray that crosses the midpoint, and using the best solution out of the three.

Running Time

- Recurence: $T(n)=2T(n/2)+\Theta(n)$.
- The above recurence equation is exactly the same as for Merge Sort, so $T(n) = \Theta(n \lg n)$

Summary

- Recursion
  - Fibonacci Numbers
  - Tower of Hanoi
- Recurence Analysis
  - Substitution Method
  - Recursion-Tree Method
  - Master Method
- Divide-and-Conquer
  - Merge Sort
  - Maximum-Subarray Problem