

ENPM 809X

E. Gunduzhan

Module 2: Growth of functions; Divide and conquer algorithms; Insertion sort and merge sort

Growth of Functions

Order of Growth

- Order of growth of the running time of an algorithm is a measure of its efficiency
- Assumes large input sizes – asymptotic efficiency
 - The extra precision to calculate the exact running time is usually unnecessary
- An algorithm that is asymptotically more efficient will be the better choice for most inputs

Asymptotic Notations

- Let the worst-case running time of an algorithm be $T(n)$
 - $n \in \mathbb{Z}$ is the input size
- Definition:
 $\Theta(g(n)) = \{f(n): \text{there exist } c_1 > 0, c_2 > 0, \text{ and } n_0 > 0 \text{ such that } c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n > n_0\}$

That is, $T(n) = \Theta(g(n))$ means that $T(n)$ is “sandwiched” between $c_1g(n)$ and $c_2g(n)$ for large n

- Example:
If $T(n) = 5n^2 + 6n + 3$ we can write $T(n) = \Theta(n^2)$

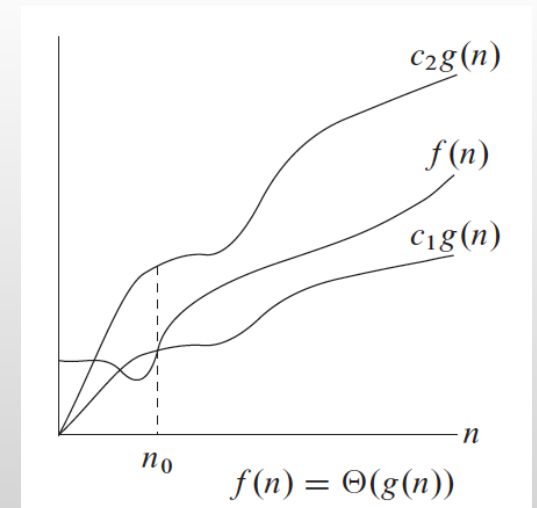


Figure 3.1-a from textbook

Asymptotic Notations (cont'd)

- In general, if $T(n) = a_d n^d + a_{d-1} n^{d-1} + \dots + a_1 n + a_0$ is any polynomial with degree d , we can write $T(n) = \Theta(n^d)$
“Keep only the highest order term and ignore the constant coefficient”

- Example:

$$T(n) = 5n^2 + 6n + 3$$

We want to find c_1 and c_2 such that $c_1 n^2 \leq 5n^2 + 6n + 3 \leq c_2 n^2$ for large n

Choose $c_2 = 6$ (any number larger than 5 would do)

Choose $c_1 = 5$ (any number less than or equal to 5 would do)

Therefore $T(n) = \Theta(n^2)$

Asymptotic Notations (cont'd)

- Definition:

$$O(g(n)) = \{f(n): \text{there exist } c > 0 \text{ and } n_0 > 0 \text{ such that} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n > n_0\}$$

That is, $T(n) = O(g(n))$ means that
 $g(n)$ is an asymptotic upper bound for $T(n)$

$T(n) = \Theta(g(n))$ implies that $T(n) = O(g(n))$
but not vice versa

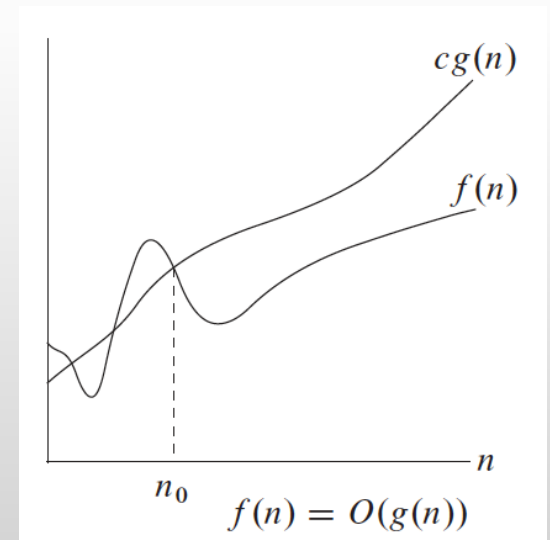


Figure 3.1-b from textbook

Asymptotic Notations (cont'd)

- Definition:

$$\Omega(g(n)) = \{f(n) : \text{there exist } c > 0 \text{ and } n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n > n_0\}$$

That is, $T(n) = \Omega(g(n))$ means that $g(n)$ is an asymptotic lower bound for $T(n)$

$T(n) = \Theta(g(n))$ implies that $T(n) = \Omega(g(n))$ but not vice versa;

However $T(n) = O(g(n))$ and $T(n) = \Omega(g(n))$ if and only if $T(n) = \Theta(g(n))$

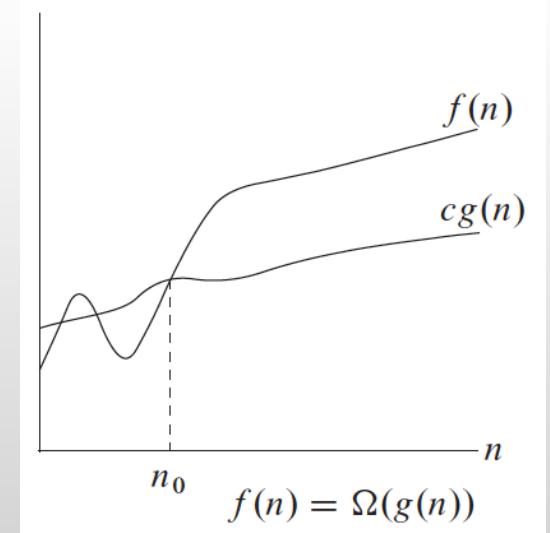


Figure 3.1-c from textbook

Example: (Problem 3.4-f)

Show that $f(n) = O(g(n))$ implies that $g(n) = \Omega(f(n))$

If $f(n) = O(g(n))$ then there exist $c > 0$ and $n_0 > 0$ such that

$$0 \leq f(n) \leq cg(n) \text{ for all } n > n_0$$

But then for $c' = 1/c > 0$, we have

$$0 \leq c'f(n) \leq g(n) \text{ for all } n > n_0$$

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

Example: (Problem 3.4-e)

Is $f(n) = \Theta(f(n/2))$ for all functions $f(n)$?

For the equality to hold, there must be $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that

$$c_1 f(n/2) \leq f(n) \leq c_2 f(n/2) \text{ for all } n > n_0$$

Let $f(n) = 2^n$. Then, we need

$$c_1 2^{n/2} \leq 2^n \leq c_2 2^{n/2} \text{ for all } n > n_0$$

$$c_1 \leq 2^{n/2} \leq c_2 \text{ for all } n > n_0$$

This cannot be satisfied for any choice of $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$.

Monotonicity of Functions

- A function $T(n)$ is:
 - monotonically increasing if $m \leq n$ implies $T(m) \leq T(n)$
 - monotonically decreasing if $m \leq n$ implies $T(m) \geq T(n)$
 - strictly increasing if $m < n$ implies $T(m) < T(n)$
 - strictly decreasing if $m < n$ implies $T(m) > T(n)$

Floors and Ceilings

- For any real number x , we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (“floor”)
- For any real number x , we denote the least integer greater than or equal to x by $\lceil x \rceil$ (“ceiling”)
- For all x ,

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

Insertion Sort Algorithm

- Problem: Given a sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$ find a permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$.
- Insertion Sort Algorithm:
 - Start with an empty output sequence
 - Take input numbers one by one, and insert each one into the output sequence in correct order
 - To find the correct order, we need to compare each input number with some or all of the numbers already in the output sequence

Pseudo Code for Insertion Sort

Can be implemented using only the memory allocated to the input sequence, without the need for separate memory for the output

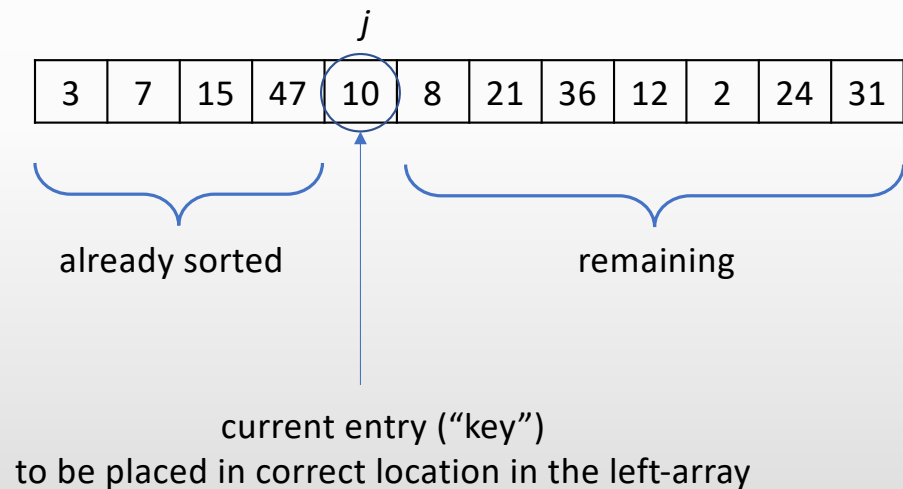
Insertion-Sort(A)

for $j = 2$ to $A.length$

$key = A[j]$

 insert $A[j]$ in correct
 location in $A[1..j]$

What should happen to the
existing entries in $A[1..j]$?



Pseudo Code for Insertion Sort

Insertion-Sort(A)

for $j = 2$ to $A.length$

$key = A[j]$

$i = j - 1$

 while $i > 0$ and $A[i] > key$

$A[i+1] = A[i]$

$i = i - 1$

$A[i+1] = key$

Example

	1	2	3	4	5	6
$j = 2$	5	2	4	6	1	3
$j = 3$	2	5	4	6	1	3
$j = 4$	2	4	5	6	1	3
$j = 5$	2	4	5	6	1	3
$j = 6$	1	2	4	5	6	3
	1	2	3	4	5	6

Proving Correctness of the Algorithm

- A **loop invariant** is a property of a program loop that is true before (and after) each iteration of the loop.
- We need to show that:
 - A loop invariant is true prior to the first iteration of the loop
 - If it is true before an iteration, then it must be true before the next iteration
- Then, when the loop terminates, the loop invariant can help showing the correctness of the algorithm
- Loop Invariant for Insertion Sort:
 - At the start of each iteration, the subarray $A[1..j - 1]$ consists of the elements originally in that subarray but in sorted order

Proving Correctness of the Algorithm (cont'd)

Insertion-Sort(A)

for $j = 2$ to $A.length$

$key = A[j]$

$i = j - 1$

 while $i > 0$ and $A[i] > key$

$A[i+1] = A[i]$

$i = i - 1$

$A[i+1] = key$

- Loop Invariant for Insertion Sort:

At the start of each iteration, the subarray $A[1..j - 1]$ consists of the elements originally in that subarray but in sorted order

- Verify initialization:

When $j = 2$, $A[1]$ is the single original element, and it is “sorted”

- Verify iterations:

If we assume $A[1..j - 1]$ is sorted at the beginning of an iteration, the while loop shifts entries larger than $A[j]$ to the right and inserts $A[j]$ in correct position, so it produces a sorted array $A[1..j]$ at the end of the iteration

- Termination:

When $j = n + 1$, the sorted array is $A[1..n]$, which is the entire input

Running Complexity of Insertion Sort

Insertion-Sort(A)

for $j = 2$ to $A.length$ n times
 $key = A[j]$ $(n - 1)$ times
 $i = j - 1$ $(n - 1)$ times
 while $i > 0$ and $A[i] > key$ $(n - 1)m_j$ times
 $A[i+1] = A[i]$ $(n - 1)(m_j - 1)$ times
 $i = i - 1$ $(n - 1)(m_j - 1)$ times
 $A[i+1] = key$ $(n - 1)$ times

- Best case: The input sequence in A is already sorted
 - Then the while-loop is not executed for any j (i.e. $m_j = 1$)
 - The best-case complexity is therefore $T(n) = an + b = \Theta(n)$
- Worst case: The input sequence in A is in reverse order
 - Then the while-loop is executed $j - 1$ times (i.e. $m_j = j$)
 - The statements inside the while loop are executed $\Theta(n^2)$ times
 - The worst-case complexity is $T(n) = an^2 + bn + c = \Theta(n^2)$

Example: Selection Sort (Exercise 2.2-2)

Selection-Sort(A)

```
for  $j = 1$  to ??  
     $key = A[j]$   
    for ??? To ???  
        if  $A[i] > key$   
             $key = ???$   
             $A[i] = ???$   
             $A[j] = ???$ 
```

- Sort n numbers stored in array A by first finding the smallest element of A and exchanging it with $A[1]$.
- Then find the second smallest element of A , and exchange it with $A[2]$, and so on.
- How many iterations of the outer loop are needed?
- What loop invariant does this algorithm maintain?
- What is the best-case running time?
- What is the worst-case running time?

Divide and Conquer Algorithms

Solutions by Recursion

- Divide a problem into smaller sub-problems
- Solve the sub-problems recursively until they are small enough for trivial solutions
- Combine the solutions to solve the original problem

Recurrences

- Equation or inequality that describes a function in terms of its value on smaller inputs
- Characterize running times of divide and conquer techniques
- Examples:

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

$$T(n) = T(2n/3) + T(n/2) + \Theta(n)$$

$$T(n) \leq 2T(n/2) + \Theta(n)$$

Example: Exercise 4.3-1

Show that the solution of $T(n) = T(n - 1) + n$ is $O(n^2)$

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

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

$$T(n) = T(n - 3) + 3n - 3$$

$$T(n) = T(n - 4) + 4n - 6$$

$$\vdots$$

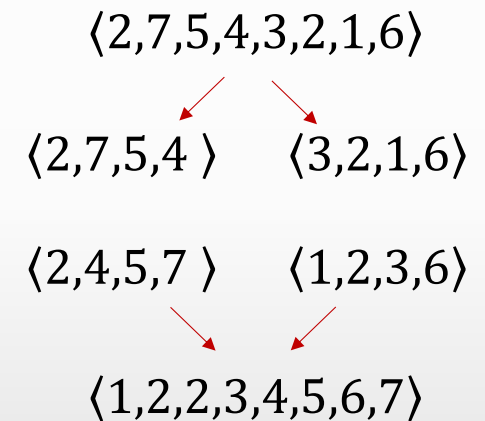
$$T(n) = T(n - k) + kn - \frac{(k-1)k}{2}$$

Setting $k = n - 1$:

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

Merge Sort

- A recursive solution for the sorting problem:
 - Divide the given n -element sequence into two $n/2$ -element subsequences
 - Sort the two subsequences recursively
 - Merge the sorted subsequences to form the sorted output



Merge Sort (cont'd)

- Termination: If $n = 1$, no need to sort, so just return the input
- Merging the two sorted subsequences efficiently is the key algorithm:
 - Start with the first elements in the two subsequences – each is the smallest in its subsequence
 - Select the smaller of the two for the output sequence
 - Move to the next element from the selected subsequence, and repeat
- May use “sentinel” values (i.e. ∞) in the subsequences to simplify end-of-sequence checks

Merge Sort – Merging Algorithm

Merge(A, p, q, r)

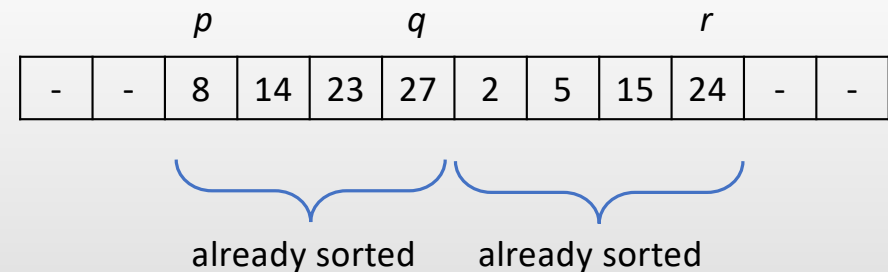
Create “Left” and “Right” subsequences

Start with the first element of each subsequence

Pick the smaller of the two elements and move to the next element

Assume $A[p..q]$ and $A[q + 1..r]$ are already sorted subsequences.

We want to merge them as the sorted sequence $A[p..r]$



Merge Sort – Merging Algorithm

Merge(A, p, q, r)

$n1 = q - p + 1$

$n2 = r - q$

for $i = 1$ to $n1$

$L[i] = A[p+i-1]$

for $j = 1$ to $n2$

$R[j] = A[q+j]$

$L[n1+1] = R[n2+1] = \infty$

$i = j = 1$

for $k = p$ to r

 if $L[i] \leq R[j]$

$A[k] = L[i]$

$i = i + 1$

 else

$A[k] = R[j]$

$j = j + 1$

Assume $A[p..q]$ and $A[q + 1..r]$ are already sorted subsequences.

We want to merge them as the sorted sequence $A[p..r]$

Create “Left” and “Right” subsequences of length $n1$ and $n2$

Terminate each subsequence with a “large” sentinel value

Start with the first element of each subsequence

Pick the smaller of the two elements and move to the next element

Merging Subsequences- Example

A: ...

p				q				r
2	4	5	7	3	2	1	6	

 ...

L:

2	4	5	7	∞
---	---	---	---	----------

 R:

1	2	3	6	∞
---	---	---	---	----------

L:

2	4	5	7	∞
----------	---	---	---	----------

 i R:

1	2	3	6	∞
----------	---	---	---	----------

 j

L:

2	4	5	7	∞
----------	---	---	---	----------

 i R:

1	2	3	6	∞
---	----------	---	---	----------

 j

L:

2	4	5	7	∞
---	----------	---	---	----------

 i R:

1	2	3	6	∞
---	----------	---	---	----------

 j

L:

2	4	5	7	∞
---	----------	---	---	----------

 i R:

1	2	3	6	∞
---	---	----------	---	----------

 j

A: ...

1	4	5	7	3	2	1	6
---	---	---	---	---	---	---	---

 ... k

A: ...

1	2	5	7	3	2	1	6
---	---	---	---	---	---	---	---

 ... k

A: ...

1	2	2	7	3	2	1	6
---	---	---	---	---	---	---	---

 ... k

A: ...

1	2	2	3	3	2	1	6
---	---	---	---	---	---	---	---

 ... k

Correctness of the Merging Algorithm

Merge(A, p, q, r)

$n1 = q - p + 1$

$n2 = r - q$

for $i = 1$ to $n1$

$L[i] = A[p+i-1]$

for $j = 1$ to $n2$

$R[j] = A[q+j]$

$L[n1+1] = R[n2+1] = \infty$

$i = j = 1$

for $k = p$ to r

 if $L[i] \leq R[j]$

$A[k] = L[i]$

$i = i + 1$

 else

$A[k] = R[j]$

$j = j + 1$

- Loop invariant:

$A[p..k-1]$ contains $k-p$ smallest elements in L and R in sorted order; $L[i]$ and $R[j]$ are the smallest remaining elements in the two subsequences

- If $k = p$, $A[p..p-1]$ is empty; and $i = j = 1$ so $L[i]$ and $R[j]$ are smallest
- At some value of k , suppose $L[i] \leq R[j]$
 - Then, $L[i]$ is the smallest remaining element in both subsequences and will be copied to $A[k]$; hence $A[p..k]$ will contain $k-p+1$ smallest elements in sorted order
 - i is incremented so that the new $L[i]$ is the smallest remaining in L
- Similar if $L[i] > R[j]$

Complexity of the Merging Algorithm

Merge(A, p, q, r)

$n1 = q - p + 1$

$n2 = r - q$

for $i = 1$ to $n1$

$L[i] = A[p+i-1]$

for $j = 1$ to $n2$

$R[j] = A[q+j]$

$L[n1+1] = R[n2+1] = \infty$

$i = j = 1$

for $k = p$ to r

 if $L[i] \leq R[j]$

$A[k] = L[i]$

$i = i + 1$

 else

$A[k] = R[j]$

$j = j + 1$

constant time

$\Theta(n)$ time

constant time

$\Theta(n)$ time total (each statement in the for loop requires constant time)

Number of elements to be merged: $n = r - p + 1$

Complexity of merging: $\Theta(n)$

Back to Merge Sort

Merge-Sort(A, p, r)

if $p < r$

$q = \lfloor (p + r) / 2 \rfloor$ \longrightarrow Find the index approximately half way between p and r

Merge-Sort(A, p, q) \longrightarrow Sort $A[p..q]$

Merge-Sort($A, q+1, r$) \longrightarrow Sort $A[q + 1..r]$

Merge(A, p, q, r) \longrightarrow Merge the two subarrays into a single sorted array

Sort elements of subarray $A[p..r]$

If $p \geq r$ then A has only one element, $A[p]$. (Just return that element in this case)

Complexity of Merge Sort

- If $n = 1$ then $T(n) = \Theta(1)$ (i.e. constant)
- For any other n , there are three tasks
 - Dividing the array into two subarrays: Takes constant time $\Theta(1)$
 - Sorting two subarrays of size $n/2$: Each takes $T(n/2)$
 - Merging the subarrays: Takes linear time as we just showed: $\Theta(n)$
- Recall that $\Theta(n) + \Theta(1) = \Theta(n)$
- Therefore, we can write the recurrence

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

Solution of the Recurrence


- Let's write $\Theta(n) = cn + k$
- Then:

$$T(n) = 2T(n/2) + cn + k$$

$$T(n/2) = 2T(n/4) + cn/2 + k \rightarrow T(n) = 4T(n/4) + 2cn + 3k$$

$$T(n/4) = 2T(n/8) + cn/4 + k \rightarrow T(n) = 8T(n/8) + 3cn + 7k$$

$$T(n/8) = 2T(n/16) + cn/8 + k \rightarrow T(n) = 16T(n/16) + 4cn + 15k$$

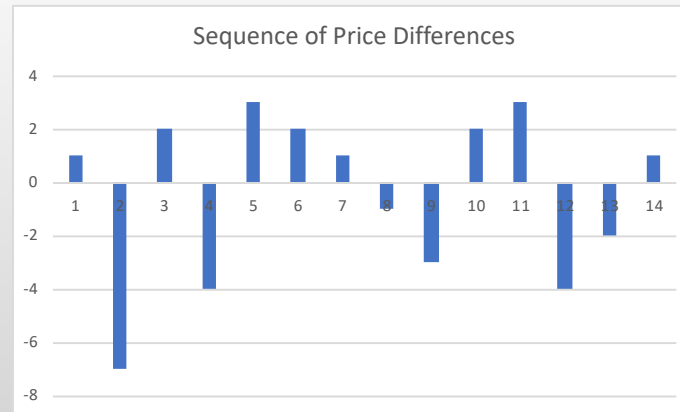
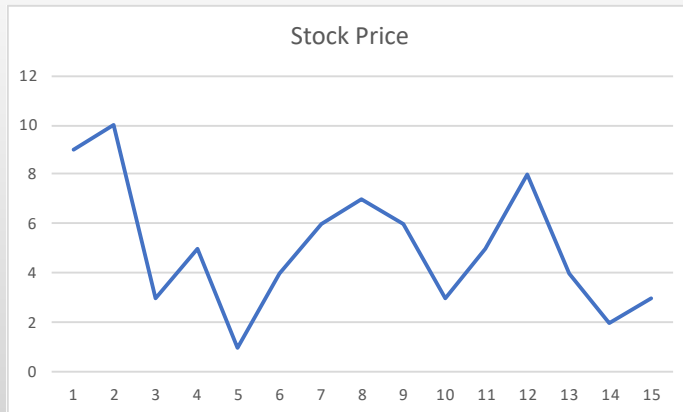

$$T(n) = 2^m T(n/2^m) + mcn + (2^m - 1)k$$

$$T(n) = nT(1) + cn \lg n + (n - 1)k = \Theta(n \lg n)$$

Let $n = 2^m$ (or, $m = \lg n$)

Maximum Subarray Problem

- Problem: Given a sequence of numbers $\langle a_1, a_2, \dots, a_n \rangle$ find a subarray of contiguous numbers $\langle a_i, a_{i+1}, \dots, a_{j-1}, a_j \rangle$ (where $1 \leq i \leq j \leq n$) such that the sum of the numbers $\sum_{m=i}^j a_m$ is the maximum over all possible subarrays.
- One application of the problem is determining maximum possible profit by buying and selling the stock of a company in a given period



$\langle 1, -7, 2, -4, 3, 2, 1, -1, -3, 2, 3, -4, 1 \rangle$

Maximum profit is realized if the stock is held during the maximum subarray of price differences and not other times

Brute Force Solution

- If we have a sequence of n numbers, there are $\binom{n}{2} = \frac{n(n-1)}{2}$ possible pairs for start/stop of the subarray (“buying and selling stock”)
- We could calculate the sum for each subarray and find the maximum, with a complexity of $\Theta(n^2)$

Recursive Solution

- Suppose we divide the given sequence $\langle a_1, a_2, \dots, a_n \rangle$ into two (approximate) halves:

$$\langle a_1, \dots, a_{n/2} \rangle \text{ and } \langle a_{n/2+1}, \dots, a_n \rangle$$

- Then the maximum subarray should be:
 - Entirely in the left subsequence,
 - Entirely in the right subsequence, or
 - Crossing the midpoint (i.e. start in the left subsequence and end in the right subsequence)
- We can recursively find the maximum subarrays of left and right subsequences – we need an algorithm to find the maximum subarray crossing the midpoint

Max Crossing Subarray

Find-Cross-Array(A, l, m, h)

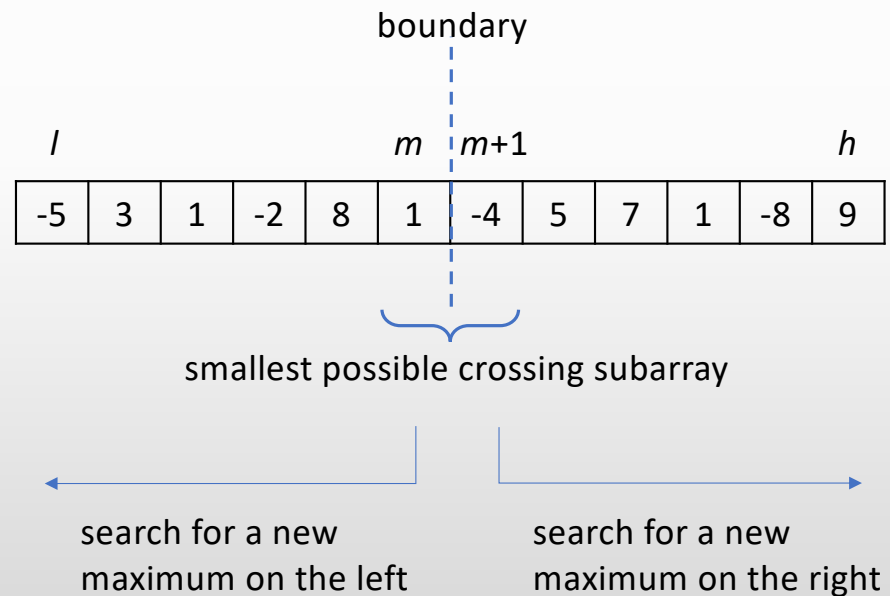
Initialize to the two elements on either side of the boundary

Find the maximizing index on the left

Find the maximizing index on the right

Return the indices and the maximum

Find max subarray in $A[l..h]$ that crosses over m



Max Crossing Subarray

Find-Cross-Array(A, l, m, h)

$best_sum = A[m] + A[m+1]$

$max_left = m$

$max_right = m+1$

$sum = best_sum$

for $i = m-1$ downto l

$sum = sum + A[i]$

 if $sum > best_sum$

$best_sum = sum$

$max_left = i$

$sum = best_sum$

for $i = m+2$ to h

$sum = sum + A[i]$

 if $sum > best_sum$

$best_sum = sum$

$max_right = i$

return($max_left, max_right, best_sum$)

Find max subarray in $A[l..h]$ that crosses over m

Initialize to the two elements on either side of the boundary

First go to left. If there is a new start index that increases the best sum, then modify max_left and $best_sum$ accordingly.

Then go to right. If there is a new stop index that increases the best sum, then modify max_right and $best_sum$ accordingly.

Return the start/stop indices of the max crossing subarray as well as its sum.

Complexity of Max Crossing Subarray

Find-Cross-Array(A, l, m, h)

$best_sum = A[m] + A[m+1]$

$max_left = m$

$max_right = m+1$

$sum = best_sum$

for $i = m-1$ downto l

$sum = sum + A[i]$

 if $sum > best_sum$

$best_sum = sum$

$max_left = i$

$sum = best_sum$

for $i = m+2$ to h

$sum = sum + A[i]$

 if $sum > best_sum$

$best_sum = sum$

$max_right = i$

return($max_left, max_right, best_sum$)

Constant time $\Theta(1)$

Each statement in the loop takes constant time. The entire loop takes $\Theta(n)$, where $n = h - l + 1$

Each statement in the loop takes constant time. The entire loop takes $\Theta(n)$

Total complexity of the max crossing subarray algorithm is $\Theta(n)$

Recursive Solution for Maximum Subarray Problem

Find-Max-Subarray(*A, l, h*)

Find maximum subarray in $A[l..h]$

if $l == h$ return($l, h, A[l]$)

If there is only one element, return it as the max subarray and exit

$m = \lfloor (l + h) / 2 \rfloor$



Find the midpoint of the array

($left_l, left_h, left_sum$) =

Find-Max-Subarray(A, l, m)



Recursive call for the left half

($right_l, right_h, right_sum$) =

Find-Max-Subarray($A, m+1, h$)



Recursive call for the right half

($cross_l, cross_h, cross_sum$) =

Find-Cross-Array(A, l, m, h)



Find maximum crossing subarray

if $left_sum \geq right_sum$ and $left_sum \geq cross_sum$

return($left_l, left_h, left_sum$)

else if $right_sum \geq left_sum$ and $right_sum \geq cross_sum$

return($right_l, right_h, right_sum$)

else

return($cross_l, cross_h, cross_sum$)



Return the parameters corresponding to the maximum sum of the three

Complexity Analysis

Find-Max-Subarray(A, l, h)

if $l == h$ return($l, h, A[l]$)



Base case takes constant time, i.e. $T(1) = \Theta(1)$

$m = \lfloor (l + h) / 2 \rfloor$

($left_l, left_h, left_sum$) =

Find-Max-Subarray(A, l, m)



($right_l, right_h, right_sum$) =

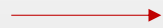
Find-Max-Subarray($A, m+1, h$)



Each recursive step operates on arrays with (approximately) half size, therefore the two steps take $2T(n/2)$

($cross_l, cross_h, cross_sum$) =

Find-Cross-Array(A, l, m, h)



As we showed before, the crossing solution takes $\Theta(n)$.

if $left_sum \geq right_sum$ and $left_sum \geq cross_sum$

return($left_l, left_h, left_sum$)

else if $right_sum \geq left_sum$ and $right_sum \geq cross_sum$

return($right_l, right_h, right_sum$)

else

return($cross_l, cross_h, cross_sum$)

All the other lines take constant time, $\Theta(1)$

Complexity Analysis (cont'd)

Therefore, combining the complexity for each part, and once again noting that $\Theta(n) + \Theta(1) = \Theta(n)$, we get

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

This is the same recurrence as the merge sort.

Therefore, recursive solution to maximum subarray problem has complexity:

$$T(n) = \Theta(n \lg n)$$

Recursive Algorithms for Matrix Multiplication

- If A and B are two $n \times n$ matrices, their product, $C = A \cdot B$ is given by

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

- Since there are n^2 elements c_{ij} and computation of each requires n multiplications and $n - 1$ additions, brute force matrix multiplication has $\Theta(n^3)$ complexity.
- Instead, we can try to decompose the problem into multiplication of smaller matrices. Assuming n is a power of 2, we can write

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where each of the 12 submatrices is $(n/2) \times (n/2)$.

Recursive Algorithms for Matrix Multiplication (cont'd)

- Calculation of each C_{uv} requires two $(n/2) \times (n/2)$ matrix multiplications and one matrix addition, e.g.

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$

- Decomposing an $n \times n$ matrix into $(n/2) \times (n/2)$ matrices requires $\Theta(n^2)$ copy operations, or it can also be done by index operations without actually copying – in $\Theta(1)$ time.
- In either case, the addition of $(n/2) \times (n/2)$ matrices above requires $\Theta(n^2)$ operations.
- Therefore, the complexity of the straightforward recursion is given by
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$
- This recurrence also has $\Theta(n^3)$ complexity, so no improvement over the brute force multiplication.

Strassen's Algorithm for Matrix Multiplication

- Reduces the number of $(n/2) \times (n/2)$ matrix multiplications from 8 to 7.
- Increases the number of matrix additions, but there are still constant number of them, so this does not increase asymptotic complexity.
- The algorithm first calculates 10 sums/differences of $(n/2) \times (n/2)$ matrices:

$$\begin{array}{llll} S_1 = B_{12} - B_{22} & S_2 = A_{11} + A_{12} & S_3 = A_{21} + A_{22} & S_4 = B_{21} - B_{11} \\ S_5 = A_{11} + A_{22} & S_6 = B_{11} + B_{22} & S_7 = A_{12} - A_{22} & S_8 = B_{21} + B_{22} \\ S_9 = A_{11} - A_{21} & S_{10} = B_{11} + B_{12} & & \end{array}$$

Strassen's Algorithm for Matrix Multiplication (cont'd)

- At the recursive step it performs $(n/2) \times (n/2)$ matrix multiplications to calculate 7 matrices:

$$\begin{array}{llll} P_1 = A_{11} \cdot S_1 & P_2 = S_2 \cdot B_{22} & P_3 = S_3 \cdot B_{11} & P_4 = A_{22} \cdot S_4 \\ P_5 = S_5 \cdot S_6 & P_6 = S_7 \cdot S_8 & P_7 = S_9 \cdot S_{10} & \end{array}$$

- Finally, C_{uv} are computed by additions/subtractions of these 7 matrices:

$$\begin{array}{ll} C_{11} = P_4 + P_5 + P_6 - P_2 & C_{12} = P_1 + P_2 \\ C_{21} = P_3 + P_4 & C_{22} = P_1 + P_5 - P_3 - P_7 \end{array}$$

Complexity of Strassen's Algorithm

- Requires 7 multiplications and 18 additions/subtractions of $(n/2) \times (n/2)$ matrices – in addition to the decomposition.
- The recurrence becomes:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

with the solution: $T(n) = \Theta(n^{\lg 7})$

- Reducing the multiplier of the recurrence from 8 to 7 reduces the complexity from $\Theta(n^3)$ to $\Theta(n^{\lg 7}) = \Theta(n^{2.81})$

Solving Recurrences by Guessing/Substitution

- Example: Suppose we want to upper bound the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

- Also suppose we could guess $T(n) = O(n \lg n)$.
- To validate / prove our guess, we need to show (by induction) that $T(n) \leq cn \lg n$ for some constant $c > 0$ and $n \geq n_0$:
 - Assume the bound holds for all $m < n$, in particular for $m = \lfloor n/2 \rfloor$.
 - Then $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$.
 - Substituting into the recurrence,

$$T(n) \leq 2c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) + n \leq cn \lg(n/2) + n = cn \lg n - cn + n$$

$$T(n) \leq cn \lg n$$

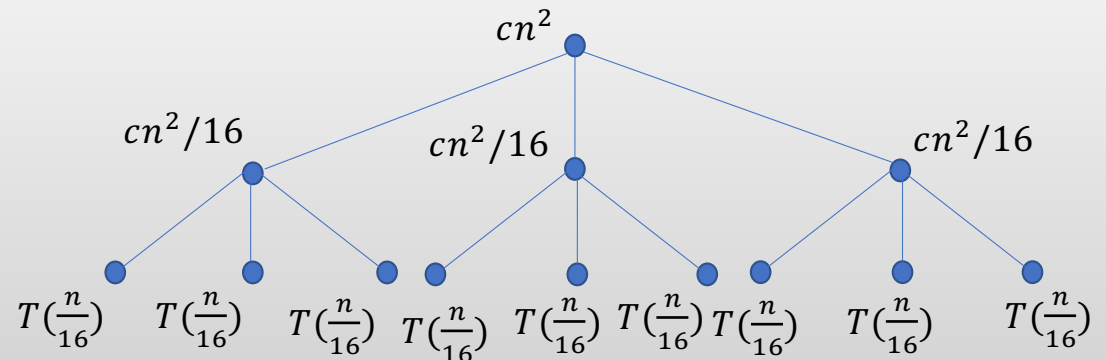
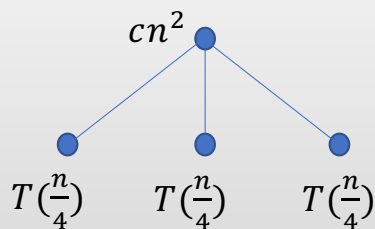
Solving Recurrences by Guessing/Substitution (cont'd)

- Also need to show the boundary condition that $T(n_0) \leq cn_0 \lg n_0$
- In the recurrence with $n = 2$, we get $T(2) = 2T(1) + 2 = 4$
- But, by choosing sufficiently large c we can always satisfy

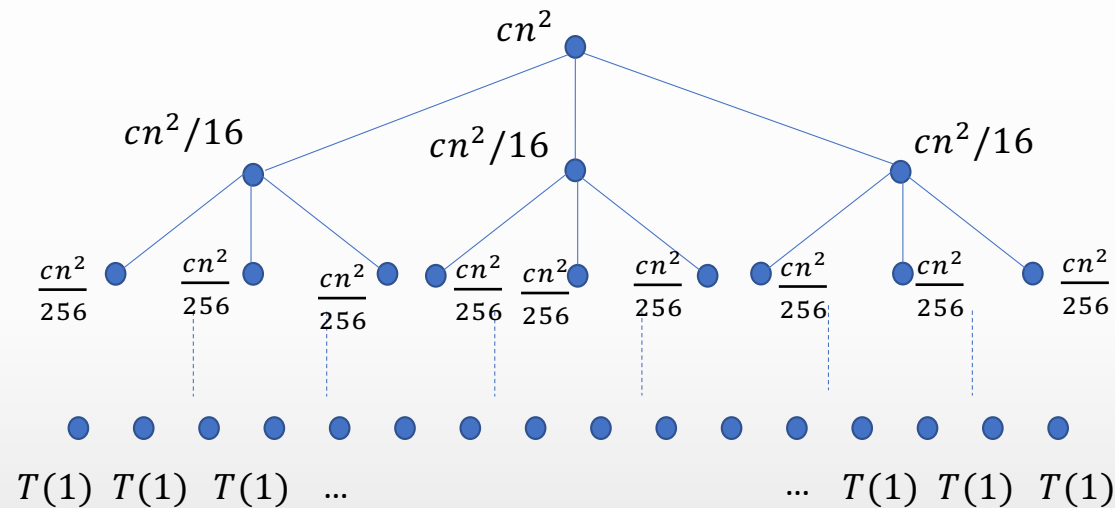
$$T(2) = 4 \leq c2 \lg 2 = 2c$$

Recursion Trees

- Help guessing a solution for a recurrence – the solution can then be proved by substitution and induction
- Each level of the tree corresponds to a level in the recursion
- The number of branches of a node is equal to the number of recursive calls for smaller size problems
- Example: $T(n) = 3T(n/4) + \Theta(n^2)$



Recursion Trees (cont'd)



Level	Nodes	Complexity
$l = 0$	1	cn^2
$l = 1$	3	$3cn^2/16$
$l = 2$	9	$9cn^2/256$
$l = \log_4 n$	$3^{\log_4 n}$	$\Theta(n^{\log_4 3})$

Total complexity:

$$T(n) = \sum_{l=0}^{\log_4 n} cn^2 \left(\frac{3}{16}\right)^l = cn^2 \frac{1 - \left(\frac{3}{16}\right)^{\log_4 n + 1}}{1 - \left(\frac{3}{16}\right)} < cn^2 \frac{1}{1 - \left(\frac{3}{16}\right)} = O(n^2)$$

Master Method for Solving Recurrences

- Solves some recurrences, in which a problem of size n is divided into a sub-problems, each with size n/b (where $a \geq 1$ and $b > 1$ are two constants), i.e.

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




- Master Theorem:

In the above recurrence $T(n)$ has the following asymptotic bounds:

- 1) If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
- 2) If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$
- 3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$, then $T(n) = \Theta(f(n))$

- Note that the three cases of the Master Theorem **don't cover** all possibilities, i.e. there are gaps between cases 1&2 and between cases 2&3.

Master Method for Solving Recurrences (cont'd)

- If there was no $f(n)$ term in the recurrence, $T(n) = aT(n/b)$, then the solution would be $T(n) = n^{\log_b a}$
- Three cases of the Master Theorem:
 - 1) If $f(n) = O(n^{\log_b a - \epsilon})$, then $f(n)$ grows slower than $n^{\log_b a}$
first term dominates  $T(n) = \Theta(n^{\log_b a})$
 - 2) If $f(n) = \Theta(n^{\log_b a})$, then $f(n)$ grows at the same rate as $n^{\log_b a}$
both terms contribute  $T(n) = \Theta(n^{\log_b a} \lg n)$
 - 3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$, then $f(n)$ grows faster than $n^{\log_b a}$
 $f(n)$ term dominates  $T(n) = \Theta(f(n))$

Examples for Master Method

- Example 1: $T(n) = 9T(n/3) + n$ $n^{\log_b a} = n^{\log_3 9}$ $f(n) = n = n^{\log_3 9 - 1}$
case-1 applies $\Rightarrow T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$
- Example 2: $T(n) = T(2n/3) + 1$ $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$ $f(n) = 1$
case-2 applies $\Rightarrow T(n) = \Theta(n^0 \lg n) = \Theta(\lg n)$
- Example 3: $T(n) = 3T(n/4) + n \lg n$ $n^{\log_b a} = n^{\log_4 3}$ $f(n) = n \lg n$
case-3 applies $\Rightarrow T(n) = \Theta(f(n)) = \Theta(n \lg n)$
- Example 4: $T(n) = 2T(n/2) + n \lg n$ $n^{\log_b a} = n^{\log_2 2} = n$ $f(n) = n \lg n$
 $n \lg n$ grows faster than n but slower than n^ϵ for any $\epsilon > 0$
 \Rightarrow Master Theorem does not apply