CE100 Algorithms and Programming II

Solving Recurrences / The Divide-and-Conquer

Author: Asst. Prof. Dr. Uğur CORUH

## CE100 Algorithms and Programming II

## Week-2 (Solving Recurrences / The Divide-and-Conquer)

#### Spring Semester, 2021-2022

Download [DOC-PDF](ce100-week-2-recurrence.tr.md_doc.pdf), [DOC-DOCX](ce100-week-2-recurrence.tr.md_word.docx), [SLIDE](ce100-week-2-recurrence.tr.md_slide.pdf), [PPTX](ce100-week-2-recurrence.tr.md_slide.pptx)

## Solving Recurrences

## Outline (1)

* Solving Recurrences
  + Recursion Tree
  + Master Method
  + Back-Substitution

## Outline (2)

* Divide-and-Conquer Analysis
  + Merge Sort
  + Binary Search
  + Merge Sort Analysis
  + Complexity

## Outline (3)

* Recurrence Solution

## Solving Recurrences (1)

* Reminder: Runtime of *MergeSort* was expressed as a recurrence
* Solving recurrences is like solving differential equations, integrals, etc.
  + Need to learn a few tricks

## Solving Recurrences (2)

**Recurrence:** An equation or inequality that describes a function in terms of its value on smaller inputs.

Example :

## Recurrence Example

* Simplification: Assume
* Claimed answer :
* Substitute claimed answer in the recurrence:
* True when

## Technicalities: Floor / Ceiling

Technically, should be careful about the floor and ceiling functions (as in the book).

e.g. For merge sort, the recurrence should in fact be:,

But, it’s usually ok to:

* ignore floor/ceiling
* solve for the exact power of 2 (or another number)

## Technicalities: Boundary Conditions

* Usually assume: for sufficiently small
  + Changes the exact solution, but usually the asymptotic solution is not affected (e.g. if polynomially bounded)
* For convenience, the boundary conditions generally implicitly stated in a recurrence
  + assuming that
  + for sufficiently small

## Example: When Boundary Conditions Matter

Exponential function: Assume  
 e.g.

The difference in solution more dramatic when:

## Solving Recurrences Methods

We will focus on 3 techniques

* Substitution method
* Recursion tree approach
* Master method

## Substitution Method

The most general method:

* Guess
* Prove by induction
* Solve for constants

## Substitution Method: Example (1)

Solve (assume )

1. Guess (need to prove and separately)
2. Prove by induction that for large (i.e. )
   * Inductive hypothesis: for any
   * Assuming ind. hyp. holds, prove

## Substitution Method: Example (2)

Original recurrence:

From inductive hypothesis:

Substitute this into the original recurrence:

* desired - residual
* when

## Substitution Method: Example (3)

So far, we have shown:

We can choose and

But, the proof is not complete yet.

**Reminder: Proof by induction:** *1.Prove the base cases* haven’t proved the base cases yet *2.Inductive hypothesis for smaller sizes* *3.Prove the general case*

## Substitution Method: Example (4)

* We need to prove the base cases
  + Base: for small (e.g. for )
* We should show that:
  + for , This holds if we pick big enough
* So, the proof of is complete
* But, is this a tight bound?

## Example: A tighter upper bound? (1)

* Original recurrence:
* Try to prove that ,
  + i.e.  for all
* **Ind. hyp:** Assume that for
* **Prove the general case:**

## Example: A tighter upper bound? (2)

Original recurrence: Ind. hyp: Assume that for Prove the general case:

## Example: A tighter upper bound? (3)

**Original recurrence:** **Ind. hyp:** Assume that for **Prove the general case:**

* So far, we have:
* No matter which positive c value we choose, this does not show that
* Proof failed?

## Example: A tighter upper bound? (4)

* What was the problem?
  + The inductive hypothesis was not strong enough
* **Idea:** Start with a stronger inductive hypothesis
  + Subtract a low-order term
* **Inductive hypothesis:** for
* **Prove the general case:**

## Example: A tighter upper bound? (5)

**Original recurrence:**

**Ind. hyp:** Assume that for

Prove the general case:

## Example: A tighter upper bound? (6)

We now need to prove

for the base cases.

(implicit assumption)

for small enough (e.g. )

* We can choose c1 large enough to make this hold

We have proved that

## Substitution Method: Example 2 (1)

For the recurrence ,

prove that

i.e.  for any

**Ind. hyp:** for any

**Prove general case:**

since

Proof succeeded – no need to strengthen the ind. hyp as in the last example

## Substitution Method: Example 2 (2)

We now need to prove that for the base cases for (implicit assumption) for

is sufficiently small (i.e. constant)

We can choose small enough for this to hold

We have proved that

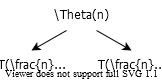
## Substitution Method - Summary

* Guess the asymptotic complexity
* Prove your guess using induction
  + Assume inductive hypothesis holds for
  + Try to prove the general case for
    - Note: prove the inequality ignore lower order terms, If the proof fails, strengthen the ind. hyp. and try again
* Prove the base cases (usually straightforward)

## Recursion Tree Method

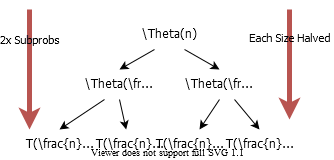
* A recursion tree models the runtime costs of a recursive execution of an algorithm.
* The recursion tree method is good for generating guesses for the substitution method.
* The recursion-tree method can be unreliable.
  + Not suitable for formal proofs
* The recursion-tree method promotes intuition, however.

## Solve Recurrence (1) :



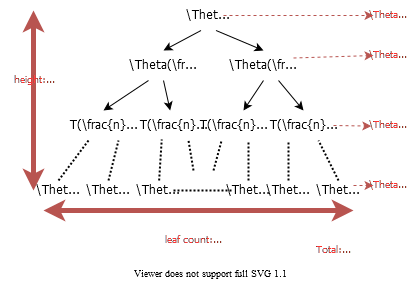
alt:“alt” height:250px center

## Solve Recurrence (2) :



alt:“alt” height:450px center

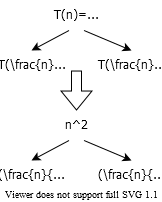
## Solve Recurrence (3) :



alt:“alt” height:500px center

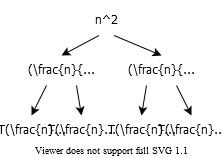
## Example of Recursion Tree (1)

Solve

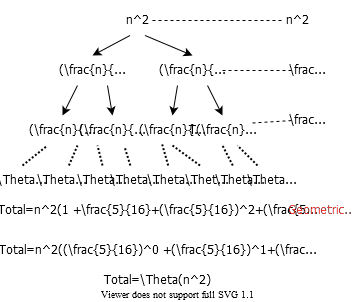


alt:“alt” height:500px center

## Example of Recursion Tree (2)

Solve 

## Example of Recursion Tree (3)

Solve 

## The Master Method

* A powerful black-box method to solve recurrences.
* The master method applies to recurrences of the form
* where , and is **asymptotically positive**.

## The Master Method: 3 Cases

(TODO : Add Notes )

* Recurrence:
* Compare with
* Intuitively:
  + **Case 1:** grows polynomially slower than
  + **Case 2:** grows at the same rate as
  + **Case 3:** grows polynomially faster than

## The Master Method: Case 1 (Bigger)

* Recurrence:
* *Case 1:* for some constant
* i.e., grows polynomialy slower than (by an factor)
* **Solution:**

## The Master Method: Case 2 (Simple Version) (Equal)

* Recurrence:
* *Case 2:*
* i.e., and grow at similar rates
* **Solution:**

## The Master Method: Case 3 (Smaller)

* *Case 3:* for some constant
* i.e., grows polynomialy faster than (by an factor)
* and the following regularity condition holds:
  + for some constant
* Solution:

## The Master Method Example (case-1) :

* grows polynomially slower than
* CASE-1:

## The Master Method Example (case-2) :

* grows at similar rate as
* CASE-2:

## The Master Method Example (case-3) (1) :

* grows polynomially faster than

## The Master Method Example (case-3) (2) : (con’t)

* Seems like CASE 3, but need to check the regularity condition
* Regularity condition for some constant
* for
* CASE-3:

## The Master Method Example (N/A case) :

* grows slower than
  + but is it polynomially slower?
  + for any
    - is not CASE-1
    - Master Method does not apply!

## The Master Method : Case 2 (General Version)

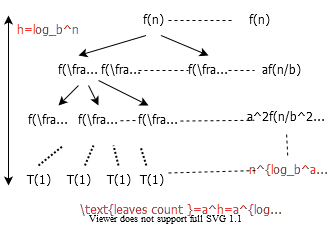
* Recurrence :
* Case 2: for some constant
* Solution :

## General Method (Akra-Bazzi)

Let be the unique solution to

Then, the answers are the same as for the master method, but with instead of *(Akra and Bazzi also prove an even more general result.)*

## Idea of Master Theorem (1)

Recursion Tree: 

## Idea of Master Theorem (2)

CASE 1 : The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.

## Idea of Master Theorem (3)

CASE 2 : The weight is approximately the same on each of the levels.

## Idea of Master Theorem (4)

CASE 3 : The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.

## Proof of Master Theorem: Case 1 and Case 2

* Recall from the recursion tree (note )

## Proof of Master Theorem Case 1 (1)

* for some

## Proof of Master Theorem Case 1 (2)

= An increasing geometric series since

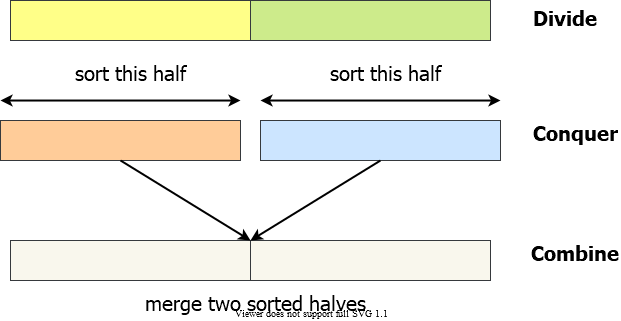
## Proof of Master Theorem Case 1 (3)

**Q.E.D.** (Quod Erat Demonstrandum)

## Proof of Master Theorem Case 2 (limited to k=0)

**Q.E.D.**

## The Divide-and-Conquer Design Paradigm (1)



alt:“alt” height:500px center

## The Divide-and-Conquer Design Paradigm (2)

1. **Divide** we divide the problem into a number of subproblems.
2. **Conquer** we solve the subproblems recursively.
3. **BaseCase** solve by Brute-Force
4. **Combine** subproblem solutions to the original problem.

## The Divide-and-Conquer Design Paradigm (3)

Merge-Sort

## Selection Sort Algorithm

SELECTION-SORT(A)  
 n = A.length;  
 for j=1 to n-1  
 smallest=j;  
 for i= j+1 to n  
 if A[i]<A[smallest]  
 smallest=i;  
 endfor  
 exchange A[j] with A[smallest]  
 endfor

## Selection Sort Algorithm

* Sequential Series
* Drop low-order terms
* Ignore the constant coefficient in the leading term

## Merge Sort Algorithm (initial setup)

Merge Sort is a recursive sorting algorithm, for initial case we need to call Merge-Sort(A,1,n) for sorting

initial case

A : Array  
p : 1 (offset)  
r : n (length)  
Merge-Sort(A,1,n)

## Merge Sort Algorithm (internal iterations)

internal iterations

A : Array  
p : offset  
r : length  
Merge-Sort(A,p,r)  
 if p=r then (CHECK FOR BASE-CASE)  
 return  
 else  
 q = floor((p+r)/2) (DIVIDE)  
 Merge-Sort(A,p,q) (CONQUER)  
 Merge-Sort(A,q+1,r) (CONQUER)  
 Merge(A,p,q,r) (COMBINE)  
 endif

## Merge Sort Combine Algorithm (1)

Merge(A,p,q,r)  
 n1 = q-p+1  
 n2 = r-q  
  
 //allocate left and right arrays   
 //increment will be from left to right   
 //left part will be bigger than right part  
  
 L[1...n1+1] //left array  
 R[1...n2+1] //right array  
  
 //copy left part of array  
 for i=1 to n1  
 L[i]=A[p+i-1]  
  
 //copy right part of array  
 for j=1 to n2  
 R[j]=A[q+j]  
  
 //put end items maximum values for termination  
 L[n1+1]=inf  
 R[n2+1]=inf  
  
 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

## Example : Merge Sort

1. **Divide:** Trivial.
2. **Conquer:** Recursively sort 2 subarrays.
3. **Combine:** Linear- time merge.
   * Subproblems
   * Subproblemsize
   * Work dividing and combining

## Master Theorem: Reminder

* + Case 1:
  + Case 2:
  + Case 3: and for

## Merge Sort: Solving the Recurrence

Case-2: holds for

## Binary Search (1)

Find an element in a sorted array:

**1. Divide:** Check middle element. **2. Conquer:** Recursively search 1 subarray. **3. Combine:** Trivial.

## Binary Search (2)

## Binary Search (3) : Iterative

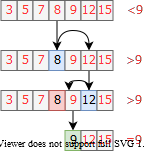
ITERATIVE-BINARY-SEARCH(A,V,low,high)  
 while low<=high  
 mid=floor((low+high)/2);  
 if v == A[mid]  
 return mid;  
 elseif v > A[mid]  
 low = mid + 1;  
 else  
 high = mid - 1;  
 endwhile  
 return NIL

## Binary Search (4): Recursive

RECURSIVE-BINARY-SEARCH(A,V,low,high)  
 if low>high  
 return NIL;  
 endif  
  
 mid = floor((low+high)/2);  
  
 if v == A[mid]  
 return mid;  
 elseif v > A[mid]  
 return RECURSIVE-BINARY-SEARCH(A,V,mid+1,high);  
 else  
 return RECURSIVE-BINARY-SEARCH(A,V,low,mid-1);  
 endif

## Binary Search (5): Recursive

## Binary Search (6): Example (Find 9)



alt:“alt” height:450px center

## Recurrence for Binary Search (7)

* Subproblems
* Subproblemsize
* Work dividing and combining

## Binary Search: Solving the Recurrence (8)

* **Case 2:** holds for

## Powering a Number: Divide & Conquer (1)

**Problem**: Compute an, where n is a natural number

NAIVE-POWER(a, n)  
 powerVal = 1;  
 for i = 1 to n  
 powerVal = powerVal \* a;  
 endfor  
return powerVal;

* What is the complexity?

## Powering a Number: Divide & Conquer (2)

* Basic Idea:

## Powering a Number: Divide & Conquer (3)

POWER(a, n)  
 if n = 0 then   
 return 1;  
 else if n is even then  
 val = POWER(a, n/2);  
 return val \* val;  
 else if n is odd then  
 val = POWER(a,(n-1)/2)  
 return val \* val \* a;  
 endif

## Powering a Number: Solving the Recurrence (4)

* **Case 2:** holds for

## Correctness Proofs for Divide and Conquer Algorithms

* **Proof by induction** commonly used for Divide and Conquer Algorithms
* **Base case:** Show that the algorithm is correct when the recursion bottoms out (i.e., for sufficiently small n)
* **Inductive hypothesis:** Assume the alg. is correct for any recursive call on any smaller subproblem of size ,
* **General case:** Based on the inductive hypothesis, prove that the alg. is correct for any input of size n

## Example Correctness Proof: Powering a Number

* **Base Case:** is correct, because it returns
* **Ind. Hyp:** Assume is correct for any
* **General Case:**
  + In function:
    - If is :
      * (due to ind. hyp.)
      * it returns
    - If is :
      * (due to ind. hyp.)
      * it returns
* The correctness proof is complete

## References

* [Introduction to Algorithms, Third Edition | The MIT Press](https://mitpress.mit.edu/books/introduction-algorithms-third-edition)
* [Bilkent CS473 Course Notes (new)](http://nabil.abubaker.bilkent.edu.tr/473/)
* [Bilkent CS473 Course Notes (old)](http://cs.bilkent.edu.tr/~ugur/teaching/cs473/)
* [Insertion Sort - GeeksforGeeks](https://www.geeksforgeeks.org/insertion-sort/)
* [NIST Dictionary of Algorithms and Data Structures](https://xlinux.nist.gov/dads/)
* [NIST - Dictionary of Algorithms and Data Structures](https://xlinux.nist.gov/dads/)
* [NIST - big-O notation](https://xlinux.nist.gov/dads/HTML/bigOnotation.html)
* [NIST - big-Omega notation](https://xlinux.nist.gov/dads/HTML/omegaCapital.html)