CSC 311 DESIGN AND ANALYSIS OF ALGORITHMS

SESSION TOPICS

Time Complexities for some general algorithms

Recurrence relations

Recurrences by substitution

Recursion tree method

Master Method

Sorting

SelectionSort

QuickSort

MergeSort

Selection

Time Complexities for some algorithms

$y \leftarrow x^3 + 10x - 20$
1 1 1 1 1

assign; raise; add; multiply; subtract = 5 operations; T(n) = 5 = constant = 1 =O(1)

```
func add(x,y){
        return x+y;
}
```

T(n)
1
So T(n) = 1 = constant = O(1)

```
func sum(A,n){
        s=0;
        for i=1 to n{
            s = s + A[i];
        }
        return s;
}
```

T(n)
1
n+1
n

1
T(n) = 2n+3=O(n); linear

Time Complexities for some algorithms

```
func addMat(A,B,C){
        for i=1 to n{
            for j = 1 to n{
                C[i,j]=A[i,j]+B[i,j];
            }
        }
}
```

T(n)
$n+1$
$n(n+1)$
$n*n=n^2$
$T(n) = 2n^2+2n+1$; quadratic
$= O(n^2)$

```
func mulMat(A,B,C){
        for i=1 to n{
            for j = 1 to n{
                C[i,j]=0
                For k=1 to n{
                    C[i,j]+=A[i,k]*B[k,j];
                }
            }
        }
}
```

T(n)
$n+1$
$n(n+1)$
$n*n=n^2$
$n^2(n+1)$
$n*n*n=n^3$
$T(n) = 2n^3 + 3n^2 + 2n + 1$
Cubic $= O(n^3)$

Time Complexities for some algorithms

```
func bsearch(A,x,n){                          -> T(n)
      i=1; j=n                                -> 1+1
       while i<j {                            -> logn
         mid = (i+j)/2;                       -> logn
         if (x<A[mid]) j=mid-1                -> 0.5logn
             else  if (x>A[mid]) i=mid+1;     -> 0.5logn
                else return mid;
      }
}
```

T(n) = 3logn+2;
= O(logn)
Logarithmic

```
func give(n){                 T(n)
      if (n>0{
          print (n)           1
          give(n-1)           T(n-1)
          give(n-1)           T(n-1)
       }
}
```

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

A recurrence relation, needs to be solved- will turn out to be exponential

Time Complexities for some algorithms

```
func rep1(n){
        for (i=1;i<n;i++) {
            print (I);
        }
}
```

T(n)
n+1
n
T(n) = 2n+1;
O(n)

```
func rep2(n){
        for (i=1;i<n;i=i+2) {
            print (I);
        }
}
```

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

Time Complexities for some algorithms

```
func rep3(n){                              → T(n)
    for (i=0;i<n;i++) {                    → n+1
        for (j=0;j<n;j++){                 → n(n+1)
            x=x*j;some statement           → n*n
        }
    }
}
```

T(n) = $2n^2+3n+1$;

O(n^2)

```
func rep4(n){                              → T(n)
    for (i=0;i<n;i++) {
        for (j=0;j<i;j++){
            x=x*j; // some statement
        }
    }
}
```

T(n)

trace i, j, no of j times

To see that the total time is 1+2+3..n= n(n+1)/2

O(n^2)

Time Complexities for some algorithms

```
func rep5(n){
        p=0;
        for (i=1;p<n;i++) {
              p=p+i;
        }
}
```

T(n)

Trace: i p

1	0+1
2	1+2
k	1+2+..k

$p=k(k+1)/2$; $k=n^{½}$ $O(n^{½})$

```
func rep6(n){
        for (i=1;i<n;i=i*2 {
              x=x*i;// some statement
        }
}
```

T(n)

trace i, to see that total time is 2^k, when $n = 2^k$

$k=\log_2 n$

$O(\log_2 n)$

Time Complexities for some algorithms

```
func rep7(n){
        for (i=n;i>1;i=i/2) {
             x=x+10;// some statement
        }
}
```

T(n)

Trace: i

$n, n/2, n/2^2, n/2^3\ n/2^4, n/2^5, ..\ n/2^k$

for $n/2^k = 1$, $n=2^k$

$k=\log_2 n$, $O(\log_2 n)$

```
func rep8(n){
        for (i=0;i<n;i++ {
             x=x*i;// some statement
        }
         for (j=0;j<n;j++ {
             x=x*i;// some statement
        }
 }
```

T(n)

n

n

2n

O(n)

Time Complexities for some algorithms

```
func rep9(n){
    p=0;
    for (i=1;i<n;i=i*2 {
        p++;
    }
    for (j=0;j<p;j=j*2 {
        x=x*i;// some statement
    }
}
```

T(n)

p = logn

logp ie. loglogn

O(loglogn)

Summary

for (i=1;i<n;i++) ---- O(n)
for (i=1;i<n;i=i+2) ---- O(n)
for (i=1;i>n;i=i--) ---- O(n)
for (i=1;i<n;i=i*2) ---- O($\log_2$n)
for (i=1;i<n;i=i*3) ---- O($\log_3$n)
for (i=1;i<n;i=i/2) ---- O($\log_2$n)

RECURRENCE RELATIONS

Recall

Recurrence relation: **Is an equation that recursively defines a sequence or multidimensional array of values, once one or more initial terms are given;**

Each further term of the sequence or array is defined as a function of the preceding terms.

Example

$F_n = g(n, f_{n-1})$ **for** $n>0$ **where** $g:N\times X \rightarrow X$

is a function, where X is a set to which the elements of a sequence must belong. For any $u_0 \in X$ **this defines a unique sequence with** u_0 **as its first element, called the initial value.**

RECURRENCE RELATIONS

Recall

Factorial: defined by the recurrence relation

$n! = n(n-1)!$ for $n > 0$, and the initial condition $0! = 1$

Logistic map: An example of a recurrence relation is the

$x_{n+1} = r x_n (1 - x_n)$, with a given constant r; given the initial term x_0 each subsequent term is determined by this relation.

Fibonacci numbers: a type of a homogeneous linear recurrence relation with constant coefficients, see below.

$F_n = F_{n-1} + F_{n-2}$ with initial conditions (seed values) $F_0 = 0$,

$F_1 = 1$.

Solving a recurrence relation means obtaining a : a non-recursive function of n.

RECURRENCE RELATIONS

A little maths

Definition again:

A recurrence relation is an equation that recursively defines a sequence where the next term is a function of the previous terms (Expressing F_n as some combination of F_i with $i<n$).

Examples:

Fibonacci series – $F_n=F_{n-1}+F_{n-2}$;with $F_0 = 0$; $F_1=1$

Tower of Hanoi – $F_n=2F_{n-1}+1$

RECURRENCE RELATIONS

A little maths

Linear Recurrence Relations

A linear recurrence equation of degree k or order k is a recurrence equation which is in the format:

$$x_n = A_1 x_{n-1} + A_2 x_{n-1} + A_3 x_{n-1} + \ldots A_k x_{n-k}$$

(A_n is a constant and $A_k \neq 0$) on a sequence of numbers as a first-degree polynomial.

RECURRENCE RELATIONS

A little maths

How to solve linear recurrence relation

Suppose, a two ordered linear recurrence relation is:

$F_n = AF_{n-1} + BF_{n-2}$ **, where A and B are real numbers.**

Rearrange:

$F_n - AF_{n-1} - BF_{n-2} = 0$

Get characteristic Equation:

The characteristic equation for the above recurrence relation is : $x^n - Ax^{n-1} - Bx^{n-2} = 0$**; dividing everything by** x^{n-2} **we get:** $x^2 - Ax - B = 0$**; a quadratic equation we can solve**

Possibilities: same roots; distinct roots; complex roots

Distinct roots: $(x-x_1)(x-x_2)=0$**, so** $F_n = ax_1^n + bx_2^n$ **is the solution**

Same roots: $(x-x_1)^2 = 0$**, so** $F_n = ax_1^n + bx_1^n$ **is the solution**

RECURRENCE RELATIONS

Back to complexity

Steps of recurrence relation

Basic step: also called initial or base condition; one or more constants that terminate recurrence

Recursive steps: generate new terms from earlier terms; get next sequence from preceding k values, ie $f_{n-1}, f_{n-2}, f_{n-3}, \ldots f_{n-k}$.

For Fibonacci sequence we have: $F_0, F_1, F_2, \ldots,$

$$F_n = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ F_{n-1} + F_{n-2} & \text{if } n>=2 \end{cases}$$

Similarly for the factorial:

$$n! = \begin{cases} 1 & \text{if } n=1 \\ n.(n-1)! & \text{if } n>1 \end{cases}$$

RECURRENCE RELATIONS

Solving recurrence relations

METHODS

There are four methods that can be used to solve the recurrence equation:

1. The Substitution Method (Guess the solution & verify by Induction)

2. Iteration Method (unrolling and summing)

3. The Recursion-tree method

4. Master method

RECURRENCE RELATIONS

Solving recurrence relations

The Substitution Method

In this method one guesses a bound and applies mathematical induction to prove that the guess is correct.

Steps

Step1: Guess the form of the Solution.

Step2: Use Mathematical Induction to prove the correctness of the guess.

Example 1

Solve the following recurrence by using substitution method.

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

RECURRENCE RELATIONS

Solving recurrence relations

The Substitution Method

Example 1 continues

Solve the following recurrence by using substitution method.

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

Step1- guess

Due to n/2 it is suggestive of nlog, so guess T(n) = O(nlogn)

ie. T(n) <= c*nlogn

Step2- mathematical induction

Apply mathematical Induction to prove the guess.

Base cases:

Let n=1: Given that T(1) = 1, we find that T(1) <= c*1.0= 0 that leads to a contradiction;

RECURRENCE RELATIONS

Solving recurrence relations

The Substitution Method

Example 1 continues

Solve the following recurrence by using substitution method.

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

Step2- mathematical induction

Base cases:

Let n=1: Given that T(1) = 1, we find that T(1) <= c*1.0= 0 that leads to a contradiction;

Let n=2; T(2) <= c.2log2=c.2;

from the equation T(2) = T(2/2)+2 = T(1)+2= 0+2 =2 <= c.2 from above.

Induction step

Assume true for n = n/2; so T(n/2) <= c.(n/2)log(n/2) holds

RECURRENCE RELATIONS

Solving recurrence relations

The Substitution Method

Example 1 continues

Solve the following recurrence by using substitution method.

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

Step2- mathematical induction

Induction step: **Assume true for n = n/2; so T(n/2) <= c.(n/2)log(n/2) holds**

Prove that it holds for n: that is T(n) <= c.nlogn

But T(n) <= 2T(fr(n/2)) + n <= 2(c)(fr(n/2))log(fr(n/2))) + n

<= cnlog(fr(n/2)) + n <= cnlogn – cnlog2 + n <= cnlogn – cn + n

<= cnlogn for every c>=1; So by induction T(n) = O(nlogn)

Drawback of the method: coming up with the correct guess is not generally easy

RECURRENCE RELATIONS

Solving recurrence relations : METHODS- 2

ITERATION METHOD

The given recurrence is substituted back to itself several times

Steps

- **Expend the recurrence through substitution**
- **Express the expansion as a summation by plugging the recurrence back into itself seeking a pattern.**
- **Work out the total sum based on arithmetic or geometric series.**

- **Example 2.1: T(n) = b, if n =1, else T(n) = c+T(n-1) if n>2**
- **Solution**
- **T(1) = b as given and T(n) = c+T(n-1), also given**
- **At n-1 we have T(n-1) = c + (c+T(n-2))= 2c + T(n-2)**
- **At n-2 we have T(n-2) = 2c +c +T(n-3) = 3c + T(n-3) ……….**
- **At n-k we have T(n-k) = c.k + T(n-k) = c.k + T(1)= nc-c+b = O(n) where for k=n-1**

RECURRENCE RELATIONS

Solving recurrence relations : METHODS- 2

ITERATION METHOD

Example 2.2: $T(n) = a$, if $n = 1$, else $T(n) = T(n/2) + n$

- Solution
- $T(n) = n + T(n/2)$
- $T(n/2)$: we have $n+n/2+T(n/4)$
- $T(n/4)$: we have $n+n/2+n/4+T(n/8)$ ……….
- $T(n/k)$: we have $n+n/2+n/4+n/8 + \ldots + n/(2^{k-1}) + T(n/(2^k)$
- At the end: $T(n/(2^k) = T(1)$, so $n/(2^k) = 1$, $k= \log_2 n$
- We have geometric series:
- $n+n/2+n/4+n/8+\ldots..+n/(2^{k-1}) + T(1) = n+n/2+n/4+n/8+\ldots..+n/(2^{k-1}) + b$
- $= n(1-(1/2)^{\log 2n})/(1-(1/2)) = 2n(1-n^{\log 1-\log 2}) = 2n(1-n^{0-1}) = 2n(1-(1/n))$
- $=2n - 2 = O(n)$

RECURRENCE RELATIONS

Solving recurrence relations

METHODS-3: The Recursion-tree method

A tree is used to trace the steps iteratively and visually; it is very convenient. Reccurence is examined until boundary conditions are reached.

General: T(n) = aT(n/b) + f(n); place f(n) at the root, spread T(n/b) a times are children

Example 1: solve T(n) = 2T(n/2)+n

Place the n at the root; for simplicity replace T(n/2) by n/2

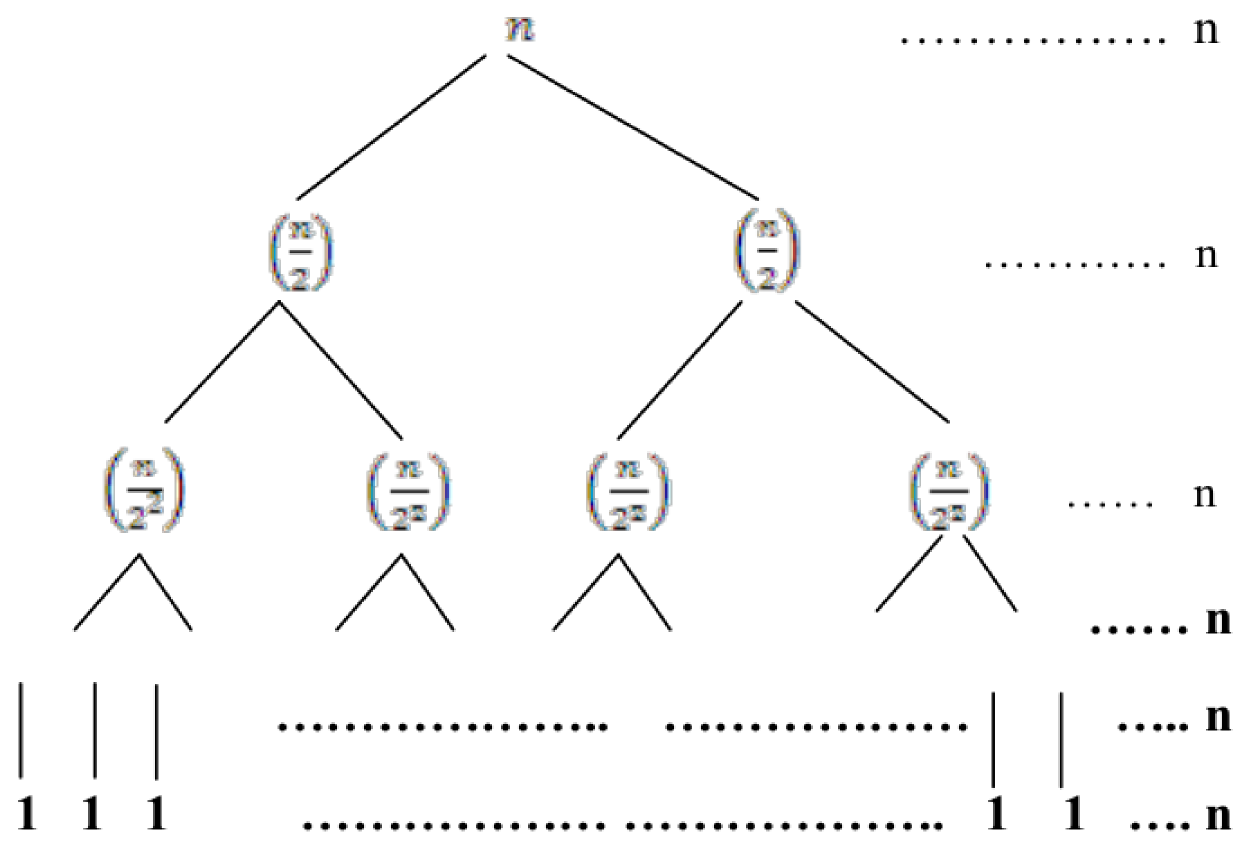
RECURRENCE RELATIONS

Solving recurrence relations

METHODS-3: The Recursion-tree method

Example 1: solve T(n) = 2T(n/2)+n

Place the n at the root; for simplicity replace T(n/2) by n/2



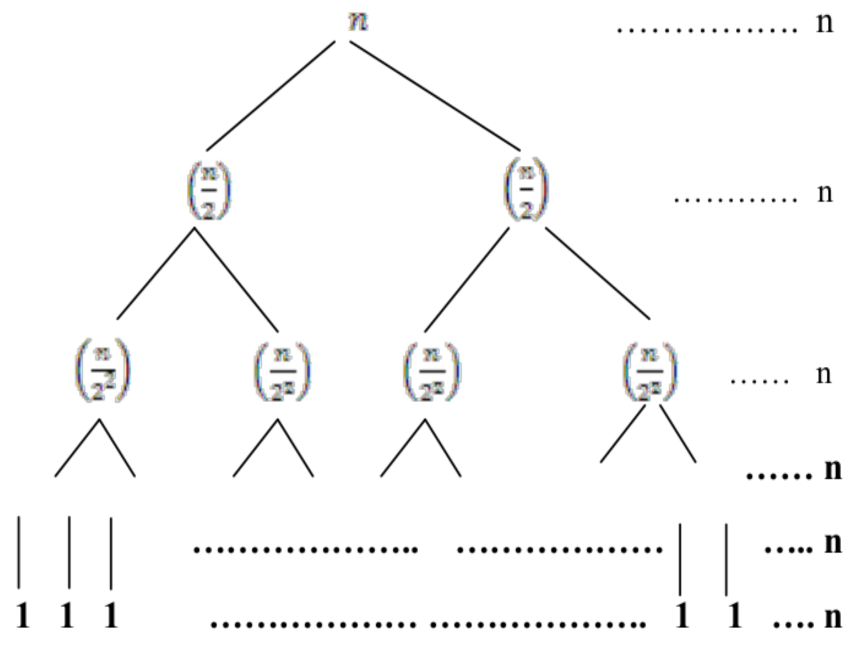
RECURRENCE RELATIONS

Solving recurrence relations

METHODS-3: The Recursion-tree method

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

The level costs each add to n; total cost is therefore n+n+….+n



The sequence:
$n, n/2, n/(2^2), n/(2^3), \ldots\ldots, n/(2^k)$
Last level = 1, so $n/(2^k) = 1$
So $n=2^k$, so $k=\log_2 n$

Total time requirement estimate:
$n + n + n + \ldots = nk$ terms
$n + n + n + \ldots = n(\log_2 n)$ terms

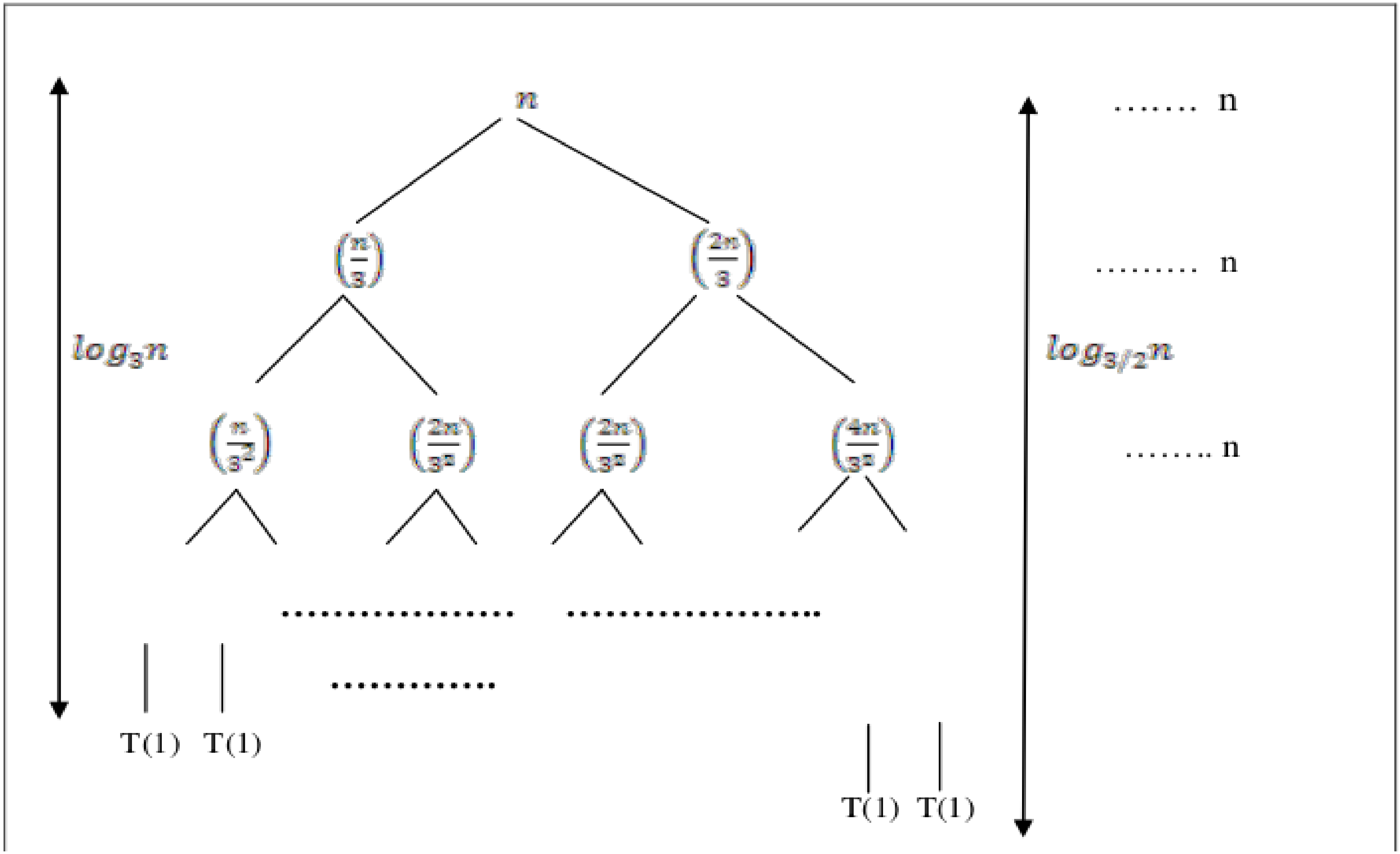
So $T(n) = O(n\log_2 n)$

RECURRENCE RELATIONS

Solving recurrence relations

METHODS-3: The Recursion-tree method

Example 2: solve T(n) = T(n/3) +T(2n/3) +n



RECURRENCE RELATIONS

Solving recurrence relations

METHODS-3: The Recursion-tree method

Example 2: solve T(n) = T(n/3) +T(2n/3) +n

The sequence:

$n, (2/3)n, (2/3)^2 n, (2/3)^3 n, \ldots\ldots, 1$

So $(2/3)^k = 1$, so $k = \log_{(3/2)} n$, k is the height of the tree

Total time estimate:

$n+n+n+\ldots+n = n(k \text{ times}) = n(\log_{(3/2)} n \text{ times})$

But $n(\log_{(3/2)} n) = (n\log_2 n)/(\log_2(3/2)) = c.n\log_2 n$

So $T(n) = O(n\log_2 n)$

For best case, take shortest path:

$n, n/3, n/3^2, n/3^3, \ldots\ldots, n/3^k$

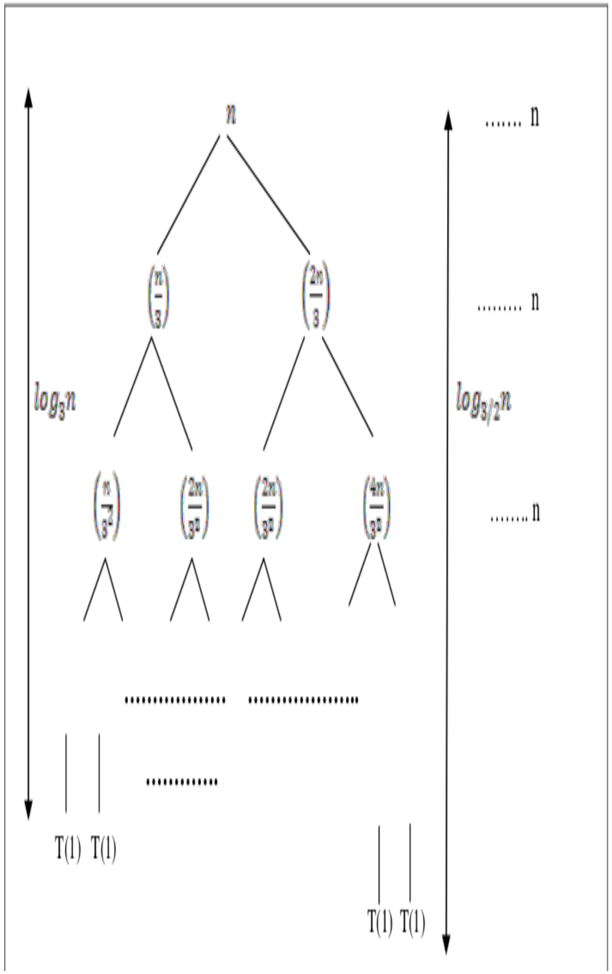
So $n/3^k = 1$, $k = \log_3 n$ which is the tree height

Estimate:

$n+n+n+ \ldots\ldots + n = n(k \text{ times}) = n(\log_3 n \text{ times})$

$\log_3 n = (\log_2 n)/(\log_2 3)$, so $T(n) = \Omega(n\log_2 n)$

So $T(n) = \Theta(n\log_2 n)$, since it both O and Ω for the same order.

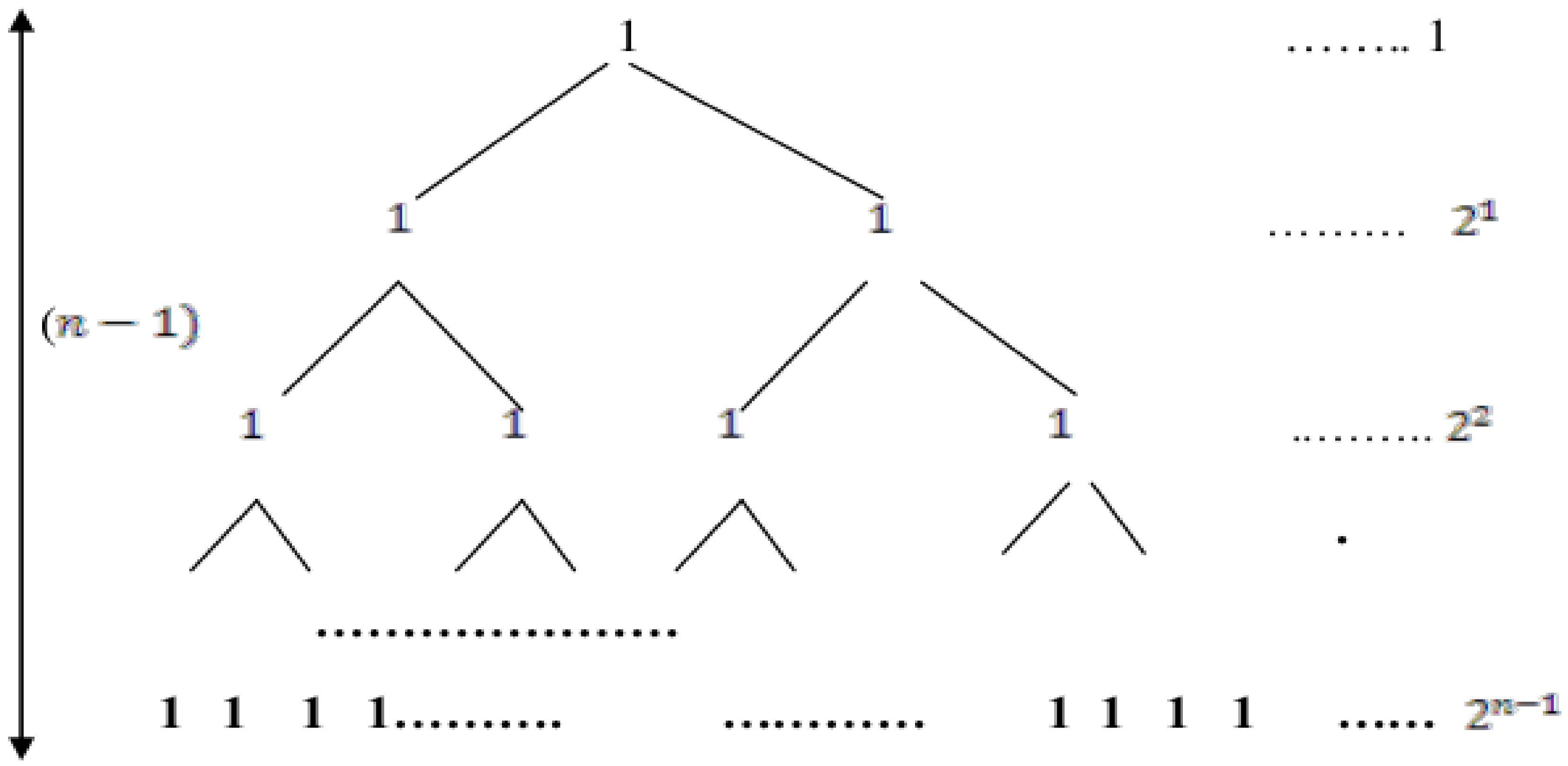


RECURRENCE RELATIONS

Solving recurrence relations

METHODS-3: The Recursion-tree method

Example 3: **solve T(n) = 2T(n-1) +1; T(1)=1; T(2)=3; Tower of Hanoi**



RECURRENCE RELATIONS

Solving recurrence relations

METHODS-3: The Recursion-tree method

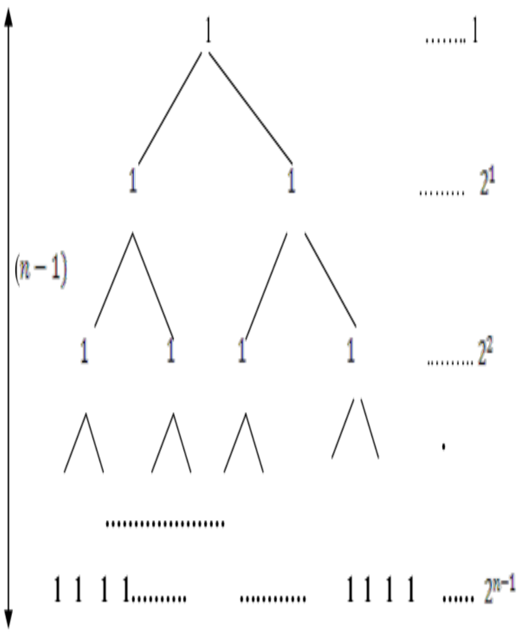
Example 3: solve $T(n) = 2T(n-1) + 1$; $T(1)=1$; $T(2)=3$; Tower of Hanoi

Last level: $n-(n-1) = 1$; also corresponds to heigh ot tree

Total cost:

$T(n) = 1+2^1+2^2,+2^3+\ldots\ldots 2^{n-1} = 1(2^n-1)/(2-1) = 2^n - 1$

$T(n) = O(2^n)$



Exercises- solve the following recurrence relations

1) $T(n)= 3T(n/2) + 1$; use iteration method
2) $T(n) = 4T(n/2) + n$; use recursion tree method
3) $T(n) = 3T(n/2) + n$; use recursion tree method
4) $T(n) = 2T(n/2) + n^2$; use recursion tree method
5) $T(n) = T(n/2) + T(n/4) + T(n/8) + n$; ; use recursion tree method
6) Write programs implementing factorial, fibonacci sequence and Tower of Hanoi and benchmark times for n=5,10,15.

RECURRENCE RELATIONS

Solving recurrence relations

MASTER METHOD

Asymptotically positive function: f(n) for which there is some $n_{0,}$ such that f(n)>0 for all $n>n_0$

Types of problems solved:

T(n) = aT(n/b) + f(n), where a, b are constants and a>=1 and b > 1, f(n) is asymptotically positive function

Note that there are a subproblems, each of size n/b

Each a subproblem takes T(n/b) and is solved recursively

The function f(n) provides the cost of dividing and combining the subproblems

n/b should be an integer, otherwise take the ceiling or the floor

a, and b are natural numbers.

RECURRENCE RELATIONS

Solving recurrence relations

MASTER METHOD

It is therefore a utility method for analyzing recurrence equations

It is used in many cases for divide and conquer algorithms

Format for recurence relations:

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

Where:

- **a, b are constants and a>=1 and b > 1,**
- **n is the size of the curent problem**
- **a is the number of subproblems in the recursion**
- **n/b is the size of the subproblems; n/b should be an integer, otherwise take the ceiling or the floor**
- **f(n) is the the cost of work done outside recursive calls such has dividing and combining the subproblems**

RECURRENCE RELATIONS

Solving recurrence relations

MASTER METHOD

MASTER THEOREM

Let $T(n) = aT(n/b) + f(n)$, where a, b are constants and $a>=1$ and $b > 1$, $f(n)$ is asymptotically bounded function and b/n is a positive integer, otherwise its ceiling or floor is taken.

Then $T(n)$ can be bounded asymptotically as follows:-

There are following three cases:

1. **If $f(n) = \Theta(n^c)$ where $c < \log_b a$ then $T(n) = \Theta(n^{\log_b a})$**

2. **If $f(n) = \Theta(n^c)$ where $c = \log_b a$ then $T(n) = \Theta(n^c \log n)$**

3. **If $f(n) = \Theta(n^c)$ where $c > \log_b a$ then $T(n) = \Theta(f(n))$**

RECURRENCE RELATIONS

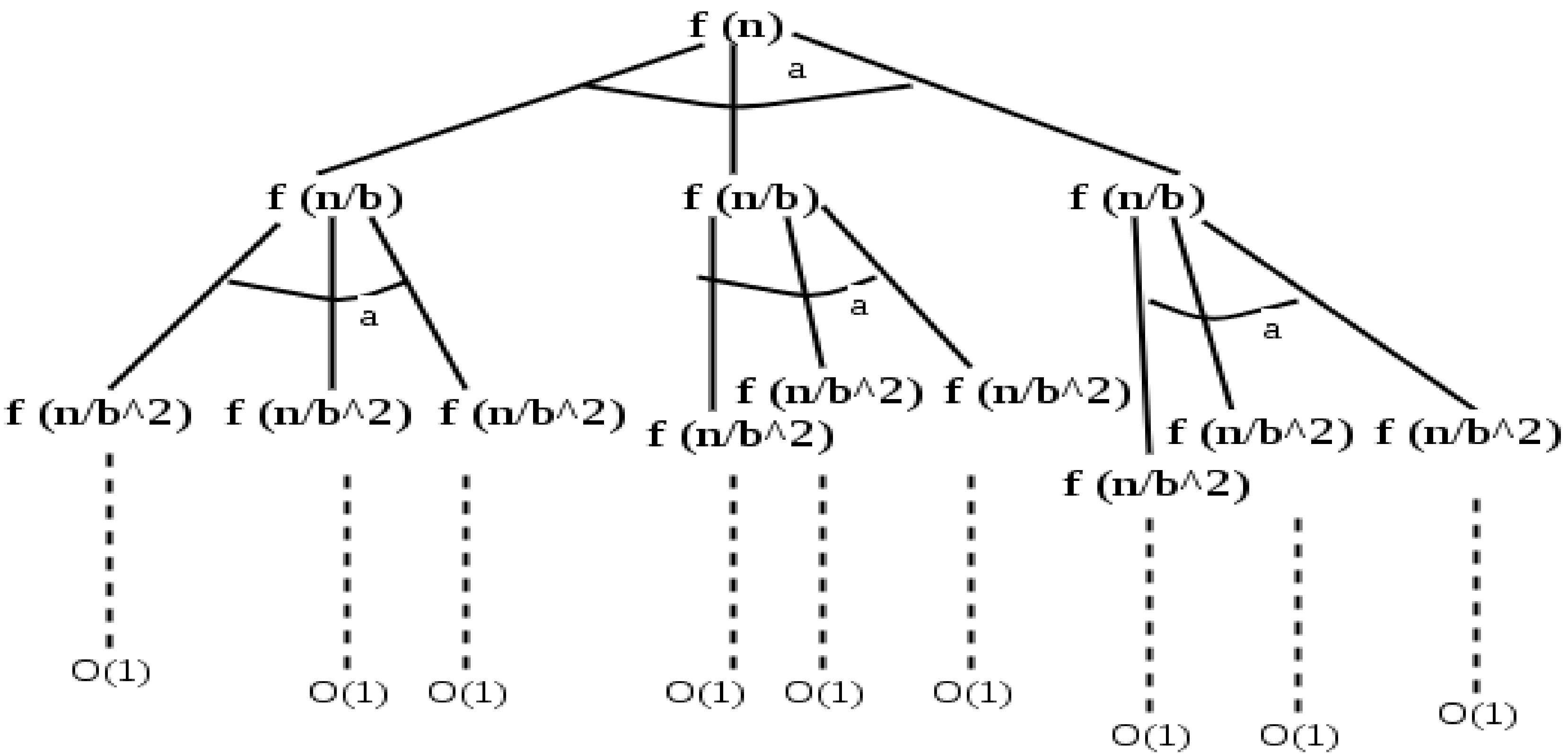
Solving recurrence relations

MASTER THEOREM - three cases:

1. If $f(n) = \Theta(n^c)$ where $c < \log_b a$ then $T(n) = \Theta(n^{\mathrm{Log}_b a})$

2. If $f(n) = \Theta(n^c)$ where $c = \log_b a$ then $T(n) = \Theta(n^c \log n)$

3. If $f(n) = \Theta(n^c)$ where $c > \log_b a$ then $T(n) = \Theta(f(n))$



RECURRENCE RELATIONS

Solving recurrence relations: MASTER THEOREM

T(n) = aT(n/b) + f(n), a>=1 and b > 1, f(n) is extra cost ; n/b is a positive integer (or floor or ceiling), (Version 2):-

There are 3 cases:

Case 1. The running time is dominated by the cost at the leaves:

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

for an $\varepsilon > 0$

Case 2. The running time is evenly distributed throughout the tree:

If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n) = \Theta(n^{\log_b(a)} \log(n))$

Case 3. The running time is dominated by the cost at the root:

If $f(n) = \Omega(n^{\log_b(a) + \varepsilon})$, then $T(n) = \Theta(f(n))$

for an $\varepsilon > 0$

If f(n) satisfies the regularity condition:
af(n/b) <= cf(n) where c < 1 (this always holds for polynomials)
Because of this condition, the Master Method cannot solve every recurrence of the given form.

RECURRENCE RELATIONS

Solving recurrence relations

MASTER METHOD

MASTER THEOREM: Hint on Applying Master Theorem

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

1. Extract a, b and f(n) from the given recurrence equation
2. Use values of a, b to evaluate the value of $n^{(\log_b (a))}$
3. Compare f(n) and what you got in 2 above ie $n^{(\log_b (a))}$
4. Identify appropriate case for Master Theorem:

   ie $f(n) > n^{(\log_b (a))}$ for case 1,

   $f(n) = n^{(\log_b (a))}$ for case 2 OR

   $f(n) < n^{(\log_b (a))}$ for case 3 provided $af(n/b) < kf(n)$ for some $k < 1$

RECURRENCE RELATIONS

Solving recurrence relations : MASTER METHOD: Version 2 cont

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, 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 constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

RECURRENCE RELATIONS

Solving recurrence relations

MASTER METHOD

MASTER THEOREM

Example: Let $T(n) = 3T(n/4) + n\log n$;

Then $a=3$, $b=4$; $f(n) = n\log n = n^{c>1}$

But $w=\log_4 3 \approx 0.79$; so $n^w = n^{0.79}$ this showns that $c > \log_4 3$

So $f(n) = n\log n = \Omega(n^{w+e})$, where $e \approx 0.21$;

apply case 3 [If $f(n) = \Theta(n^c)$ where $c > \log_b a$ then $T(n) = \Theta(f(n))$];

So $T(n) = \Theta(n\log n)$

Exercise: try for $T(n) = 2T(n/2) + n\log n$ (*****)

RECURRENCE RELATIONS

Solving recurrence relations

MASTER METHOD

MASTER THEOREM

Examples

Given: $T(n) = 2T(n/2) + n$

1. Extract: $a=2$, $b=2$ and $f(n) = n=n^1$, so $c=1$

2. Evaluate the value of $n^{(\log_b(a))}$ we have $n^{(\log_2(2))} = n^1 = n$

3. Compare $f(n)$ and what you got in 2 above ie $n^{(\log_2(2))} = n^1 = n$

4. Identify appropriate case for Master Theorem: Same so Case 2

$f(n) = n^{(\log_b(a))}$ for case 2

Applying case 2 [If $f(n) = \Theta(n^y)$ where $y = \log_b a$ then $T(n) = \Theta(n^y \log n)$] we have $T(n) = \Theta(n^1 \log n) = \Theta(n \log n)$

RECURRENCE RELATIONS

Solving recurrence relations

MASTER THEOREM: Examples

Given: $T(n) = 9T(n/3) + n$

1. Extract: $a=9$, $b=3$ and $f(n) = n$

2. Evaluate the value of $n^{(\log_b(a))}$ we have $n^{(\log_3(9))} = n^2$

3. Compare f(n) and what you got in 2 above ie $n^{(\log_2(2))} = n^2$

4. Identify appropriate case for Master Theorem: f(n) is less so Case 1

Applying case 1 [If $f(n) = \Theta(n^c)$ where $c < \log_b a$ then $T(n) = \Theta(n^{\log_b a})$] we have $T(n) = \Theta(n^2)$

RECURRENCE RELATIONS

Solving recurrence relations

MASTER THEOREM: Examples

Given: $T(n) = 3T(n/4) + n\log n$

1. Extract: $a=3$, $b=4$ and $f(n) = n\log n$

2. Evaluate the value of $n^{(\log_b(a))}$ we have $n^{(\log_4(3))} = n^{x<1}$

3. Compare $f(n)=n\log n$ with $n^{(\log_4(3))} = n^{x<1}$

4. Identify appropriate case for Master Theorem: f(n) is larger so Case 3

Applying case 3 [If $f(n) = \Theta(n^c)$ where $c > \log_b a$ then $T(n) = \Theta(f(n))$] check $af(n/b) \le kf(n)$ for $k<1$; ie $3(n/4)\log(n/4) \le kf(n)$ that is true for $k=3/4$; so we apply case 3 and have $T(n) = \Theta(n\log n)$

RECURRENCE RELATIONS

Solving recurrence relations

MASTER METHOD

MASTER THEOREM **Example:** Let $T(n) = 9T(n/3) + n$;

Then $a=9$, $b=3$; $f(n) = n=n^1$; $c=1$; $w=\log_3 9 = 2$; $n^w = n^2$

So $f(n) = O(n^{w-e})$, where $e=1$;

By case 1 [If $f(n) = \Theta(n^c)$ where $c < \log_b a$ then $T(n) = \Theta(n^{\mathrm{Log}_b a})$], $T(n)= \Theta(n^2)$

Exercise: try for $T(n) = 8T(n/2) + 1000n^2$

Example

Let $T(n) = T(2n/3) + 1$; $a=1$; $b=3/2$; $f(n) =1 =n^c$, $c=0$; where $w=\log_{3/2}1$

$\mathrm{Log}_b a= \log_{(3/2)}1 = 0$; Case 2 [If $f(n) = \Theta(n^c)$ where $c = \log_b a$ then $T(n) = \Theta(n^c \log n)$]
But $f(n) = \Theta(n^w)$, where $w=\log_3 2$. So we have $T(n) = \Theta(n^w.\log n)$, so
so $T(n) = \Theta(n^0.\log n)$, as $w=0$; then now $T(n) = \Theta(\log n)$

RECURRENCE RELATIONS

Solving recurrence relations

MASTER THEOREM

Example Case 1 Confirm the following

$T(n) = 2T(n/2) + 1;$	$T(n) = \Theta(n^1)$
$T(n) = 4T(n/2) + 1;$	$T(n) = \Theta(n^2)$
$T(n) = 4T(n/2) + n^1;$	$T(n) = \Theta(n^2)$
$T(n) = 8T(n/2) + n^2;$	$T(n) = \Theta(n^3)$
$T(n) = 16T(n/2) + n^2;$	$T(n) = \Theta(n^4)$

RECURRENCE RELATIONS

Solving recurrence relations

MASTER THEOREM

Example Case 2 Confirm the following

$T(n) = T(n/2) + 1;$	$T(n) = \Theta(\log n)$
$T(n) = 2T(n/2) + n;$	$T(n) = \Theta(n\log n)$
$T(n) = 2T(n/2) + n\log n;$	$T(n) = \Theta(n\log^2 n)$
$T(n) = 4T(n/2) + n^2;$	$T(n) = \Theta((n^2\log n)$
$T(n) = 4T(n/2) + (n\log n)^2;$	$T(n) = \Theta((n\log n)^2\log n)$
$T(n) = 2T(n/2) +n^1/(\log n);$	$T(n) = \Theta(n\log\log n)$
$T(n) = 2T(n/2) +n^1/(\log^2 n);$	$T(n) = \Theta(n)$

RECURRENCE RELATIONS

Solving recurrence relations

MASTER THEOREM

Example Case 3 Confirm the following

$T(n) = T(n/2) + n^1$;	$T(n) = \Theta(n^1)$
$T(n) = 2T(n/2) + n^2$;	$T(n) = \Theta(n^2)$
$T(n) = 2T(n/2) + n^2 \log n$;	$T(n) = \Theta(n^2 \log n)$
$T(n) = 4T(n/2) + n^3 \log^2 n$;	$T(n) = \Theta(n^3 \log^2 n)$
$T(n) = 2T(n/2) + n^2(\log n)$;	$T(n) = \Theta(n^2)$

RECURRENCE RELATIONS

Solving recurrence relations

MASTER THEOREM

Inadmissible equations

The following equations cannot be solved using the master theorem:[2]

- $$T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$$

  a is not a constant; the number of subproblems should be fixed

- $$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$$

  non-polynomial difference between f(n) and $n^{\log_b a}$ (see below)

- $$T(n) = 0.5T\left(\frac{n}{2}\right) + n$$

  $a<1$ cannot have less than one sub problem

- $$T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$$

  f(n) which is the combination time is not positive

- $$T(n) = T\left(\frac{n}{2}\right) + n(2 - \cos n)$$ Case 3 regularity violation

RECURRENCE RELATIONS

Solving recurrence relations

MASTER THEOREM EXERCISES

Use Master's Method to solve the following:

a. $T(n) = 4T\left(\frac{n}{2}\right) + n$

b. $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

c. $T(n) = 4T\left(\frac{n}{2}\right) + n^3$

d. $T(n) = 2T\left(\frac{n}{2}\right) + n\sqrt{n}$

e. $T(n) = 4T\left(\frac{n}{3}\right) + n^2$

f. $T(n) = 8T\left(\frac{n}{2}\right) + 3n^2$

RECURRENCE RELATIONS

Solving recurrence relations

Solve the following recurence relations

$T(n) = T(n-1) + 5$, $n>1$, $T(1)=0$

$T(n) = 3T(n-1)$ $n>1$, $T(1)=4$

$T(n) = T(n-1) + n$, $n>0$, $T(0)=0$

$T(n) = T(n/2) + n$, $n>1$, $T(1)=1$ (solve for $n=2^k$)

$T(n) = T(n/3) + n$, $n>1$, $T(1)=1$ (solve for $n=3^k$)

Set up a recursive algorithm based on $2^n = 2^{n-1} + 2^{n-1}$

SORTING

Selection Sort

Process

- Scan the entire list to find its smallest element;
- Exchange it with the first element, putting the smallest element in its final position in the sorted list.
- Then we scan the list, starting with the second element,
- to find the smallest among the last $n - 1$ elements and exchange it with the second element, putting the second smallest element in its final position.
- Repeat until all the elements are in their correct places.

SORTING

Selection Sort

Process

| 89 45 68 90 29 34 **17**

17 | 45 68 90 **29** 34 89

17 29 | 68 90 45 **34** 89

17 29 34 | 90 **45** 68 89

17 29 34 45 | 90 **68** 89

17 29 34 45 68 | 90 **89**

17 29 34 45 68 89 | 90

SORTING

Selection Sort

Algorithm

```
SelectionSort(A[0..n − 1])
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
    for i ← 0 to n − 2 do
        min ← i
        for j ← i + 1 to n − 1 do
            if A[j ] < A[min] min ← j
                Swap A[i] and A[min]
```

SORTING

Selection Sort

Complexity

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

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

The time complexity of Selection sort is Θ(n^2), but key swaps is Θ(n)

Exercise: Write a program that implements and times selection sort runs. Keep experimental data on this

SORTING

QuickSort

- **An important sorting algorithm that is based on the divide-and-conquer algorithmic approach.**

- **It divides element according to their value, creating partitions.**

- **A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it.**

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are } \geq A[s]}$$

SORTING

QuickSort

- **An important sorting algorithm that is based on the divide-and-conquer algorithmic approach.**

- **It divides element according to their value, creating partitions.**

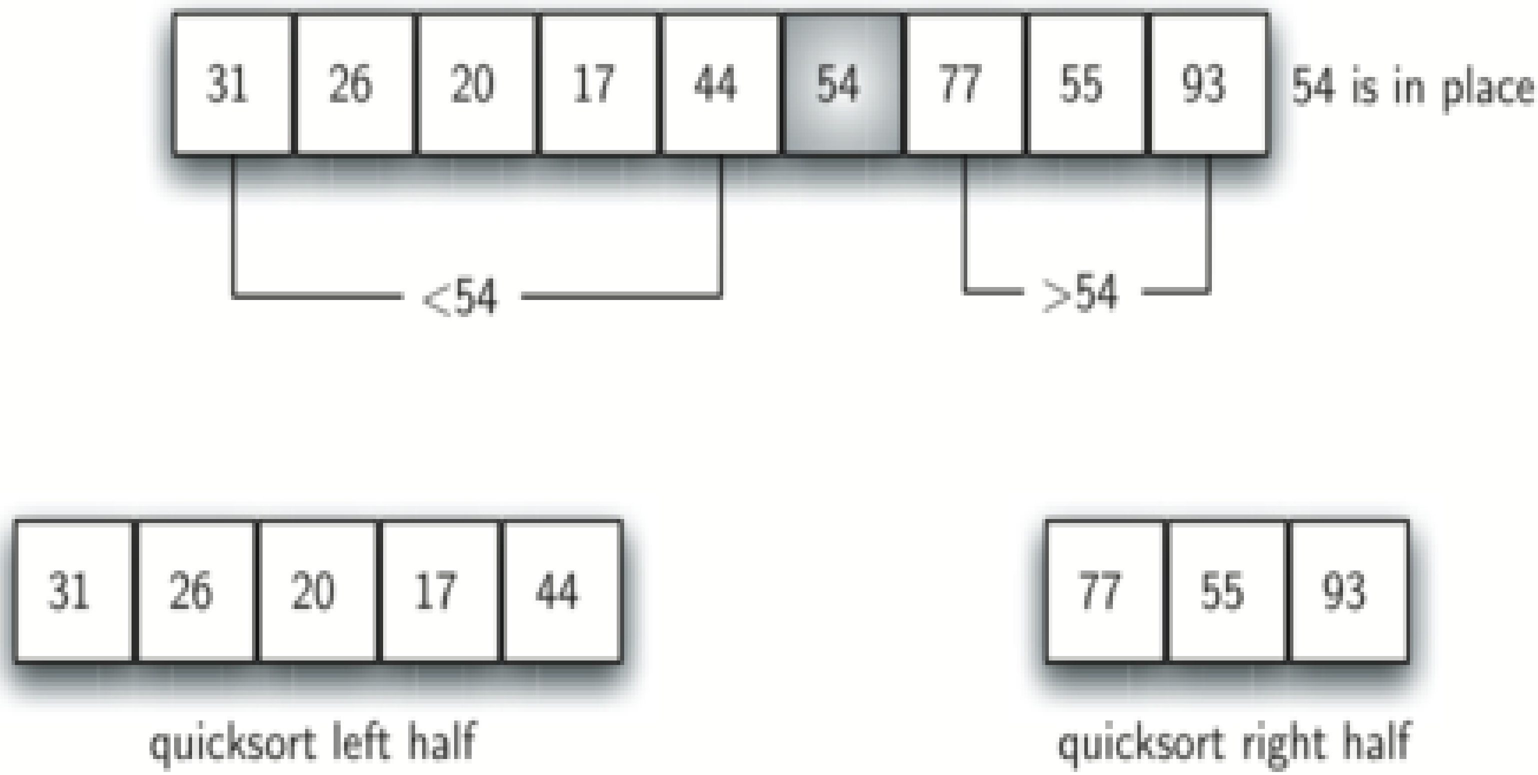
- **A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it.**

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are } \ge A[s]}$$

SORTING

QuickSort Process

$$\underbrace{A[0]\ldots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1]\ldots A[n-1]}_{\text{all are } \ge A[s]}$$



SORTING

QuickSort Process

$$\underbrace{A[0]\ldots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1]\ldots A[n-1]}_{\text{all are} \ge A[s]}$$

44	75	23	43	55	12	64	77	33

Original array—pivot is 44

23	12	43	33	44	75	55	64	77

Elements are sorted into less-than pivot values and greater-than pivot values

23	12	43	33	44	75	55	64	77

Split array into subarrays on either side of pivot value—pivot values are 23 and 75

23	12	43	33	75	55	64	77

Split subarrays and sort on pivot value

12	23	33	43	55	75	64	77

Sorted subarrays

12	23	33	43	55	75	64	77

Array after concentrating right side to left side

SORTING

QuickSort Process

Divide

Partition (rearrange) the array A[p .. r] into two (possibly empty) subarrays A[p .. q - 1] and A[q+1 .. r] such that each element of A[p .. q - 1 is less than or equal to A[q], which is, in turn, less than or equal to each element of A[q - 1 .. r]. Compute the index q as part of this partitioning procedure.

Conquer

Sort the two subarrays A[p .. q- 1] and A[q+1 .. r] by recursive calls to quicksort

Combine

Because the subarrays are already sorted, no work is needed to combine them: the entire array A[p .. r] is now sorted.

SORTING

QuickSort Algorithm

```
Quicksort(A[l..r]): //Sorts a subarray by quicksort
//Input: Subarray of array A[0..n − 1], defined by its left and right indices l and r
//Output: Subarray A[l..r] sorted in nondecreasing order
if l < r
        s ←Partition(A[l..r]) //s is a split position
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])
```

```
ALGORITHM Partition(A[l..r])
//Partitions by Hoare's algorithm, using the first element as a pivot
//Input: Subarray of array A[0..n − 1], defined by its left and right indices l and
r (l < r)
//Output: Partition of A[l..r], split position returned as this function's value
p ← A[l]
i ← l; j ← r + 1
repeat
        repeat i ← i + 1 until A[i] ≥ p
        repeat j ← j − 1 until A[j ] ≤ p
            swap(A[i], A[j ])
        until i ≥ j
        swap(A[i], A[j ]) //undo last swap when i ≥ j
        swap(A[l], A[j ])
return j
```

SORTING

QuickSort Algorithm

```
procedure quickSort(left, right)

   if right-left <= 0
      return
   else
      pivot = A[right]
      partition = partitionFunc(left, right, pivot)
      quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if

end procedure
```

SORTING

QuickSort Algorithm Complexity

Best

$T(n) = 2T(n/2) + n$, for $n > 1$, $T(1) = 0$

Using the Master Theorem, $T(n) \in (n \log_2 n)$;

Solving it exactly for $n = 2^k$ gives $T(n) = n \log_2 n$.

Worst

$T(n) = (n + 1) + n + \ldots + 3 = [((n + 1)(n + 2))/2]-3 \in \Theta(n^2)$

Average

$$T(n) = (1/n) \sum_{s=0}^{n-1} [(n + 1) + C_{avg}(s) + C_{avg}(n - 1 - s)] \text{ for } n > 1,$$

$$T(0) = 0, T(1) = 0$$

$T(n) \approx 2n \ln n \approx 1.39 n \log_2 n = \Theta(n \log_2 n)$

SORTING

QuickSort Algorithm Complexity

Best

$T(n) = 2T(n/2) + n$, for $n > 1$, $T(1) = 0$

Using the Master Theorem, $T(n) \in (n \log_2 n)$;

Solving it exactly for $n = 2^k$ gives $T(n) = n \log_2 n$.

Worst

$T(n) = (n + 1) + n + \ldots + 3 = [((n + 1)(n + 2))/2]-3 \in \Theta(n^2)$

Average

$$T(n) = (1/n) \sum_{s=0}^{n-1} [(n + 1) + C_{avg}(s) + C_{avg}(n - 1 - s)] \text{ for } n > 1,$$

$$T(0) = 0,\ T(1) = 0$$

$T(n) \approx 2n \ln n \approx 1.39 n \log_2 n = \Theta(n \log_2 n)$

Exercise: implement QuickSort and experiment on the timing with different input sets with numbers from 10, 50 and 500.

SORTING

MergeSort

- **This is a good example of a successful application of the divide-and-conquer technique.**

- **It sorts a given array A[0 .. n−1] by dividing it into two halves A[0.. $\lfloor n/2 \rfloor$) − 1] and A[$\lfloor (n/2 \rfloor$).. n − 1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.**

- **Mergesort is the method of choice for sorting linked lists and is therefore frequently used in functional and logical programming languages that have lists as their primary data structure.**

- **mergesort is basically optimal as far as the number of comparisons is concerned; so it is also a good choice if comparisons are expensive.**

SORTING

MergeSort

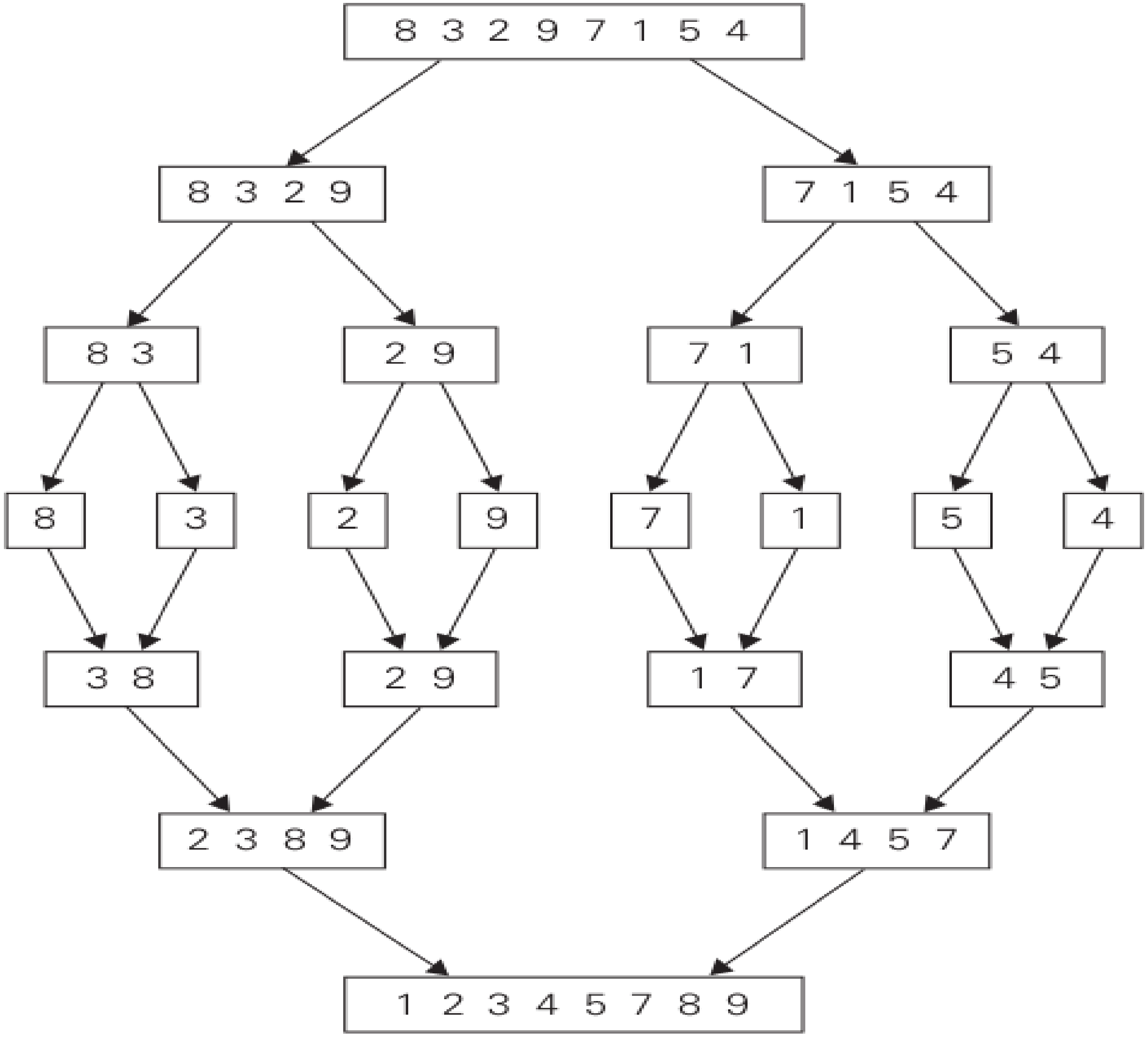
- **Divide**
  Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.

- **Conquer**
  Sort the two subsequences recursively using merge sort.

  Combine
  Merge the two sorted subsequences to produce the sorted answer.

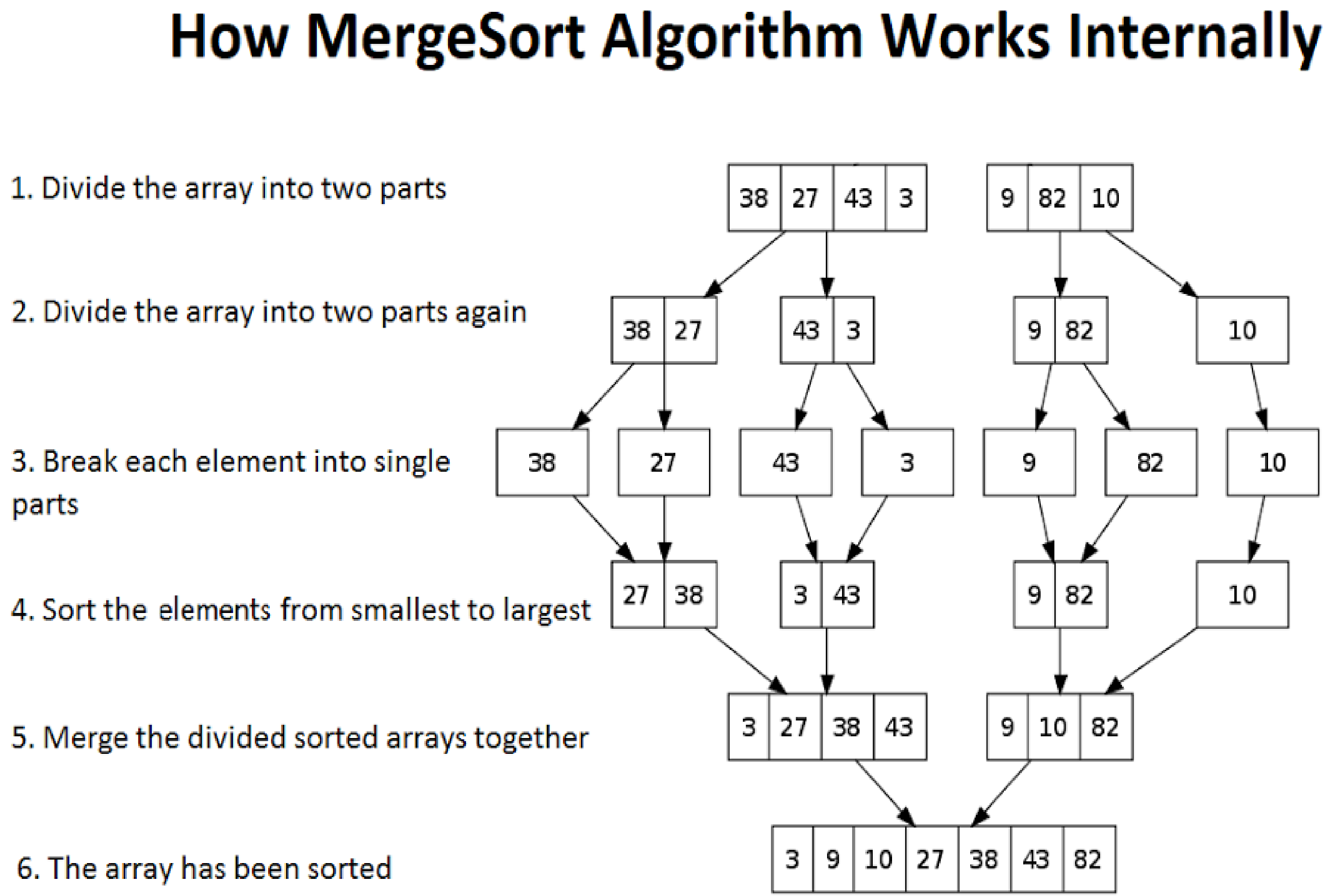
SORTING

MergeSort Process



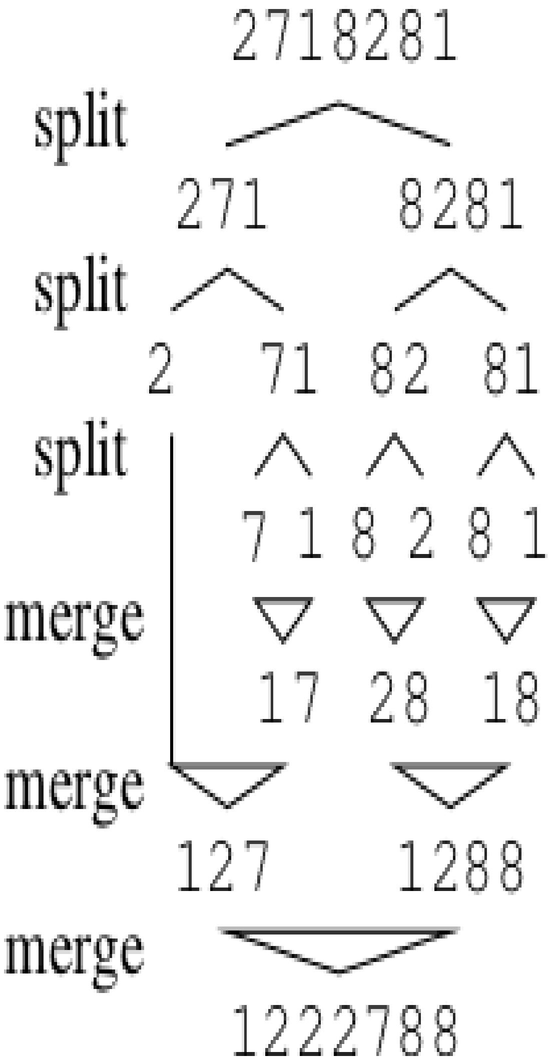
SORTING

MergeSort Process



SORTING

MergeSort Process



a	b	c	operation
$\langle 1,2,7\rangle$	$\langle 1,2,8,8\rangle$	$\langle\rangle$	move a
$\langle 2,7\rangle$	$\langle 1,2,8,8\rangle$	$\langle 1\rangle$	move b
$\langle 2,7\rangle$	$\langle 2,8,8\rangle$	$\langle 1,1\rangle$	move a
$\langle 7\rangle$	$\langle 2,8,8\rangle$	$\langle 1,1,2\rangle$	move b
$\langle 7\rangle$	$\langle 8,8\rangle$	$\langle 1,1,2,2\rangle$	move a
$\langle\rangle$	$\langle 8,8\rangle$	$\langle 1,1,2,2,7\rangle$	move a
$\langle\rangle$	$\langle\rangle$	$\langle 1,1,2,2,7,8,8\rangle$	concat b

SORTING

MergeSort Algorithm

Function $\boxed{mergeSort}(\langle e_1, \ldots, e_n \rangle)$: *Sequence* **of** *Element*

  if $n = 1$ **then return** $\langle e_1 \rangle$

  else return $merge(\boxed{mergeSort}(e_1, \ldots, e_{\lfloor n/2 \rfloor}), \boxed{mergeSort}(e_{\lfloor n/2 \rfloor + 1}, \ldots, e_n))$

// merging two sequences represented as lists

Function *merge*(a, b : *Sequence* **of** *Element*) : *Sequence* **of** *Element*

  $c := \langle \rangle$

  loop

    invariant a, b, and c are sorted and $\forall e \in c, e' \in a \cup b : e \leq e'$

    if $a.isEmpty$ **then** $c.concat(b)$; **return** c

    if $b.isEmpty$ **then** $c.concat(a)$; **return** c

    if $a.first \leq b.first$ **then** $c.moveToBack(a.first)$

    else $c.moveToBack(b.first)$

SORTING

MergeSort Algorithm

```
ALGORITHM Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B
and C
i ← 0; j ← 0; k ← 0
    while i < p and j < q do
        if B[i] ≤ C[j ]
            A[k] ← B[i]; i ← i + 1
            else A[k] ← C[j ]; j ← j + 1
                k ← k + 1
        if i = p
            copy C[j..q − 1] to A[k..p + q − 1]
            else copy B[i..p − 1] to A[k..p + q − 1]
```

SORTING

MergeSort Algorithm

TIME COMPLEXITY

Divide

The divide step just computes the middle of the subarray, which takes constant time. Thus D(n) = Θ(1).

Conquer: **We recursively solve two subproblems, each of size n/2, which contributes 2T(n/2) to the running time.**

Combine: **We have already noted that the MERGE procedure on an n-element subarray takes time Θ(n) and so C(n) = Θ(n)**

The combine recurrence:

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

SORTING

MergeSort Algorithm

TIME COMPLEXITY

The combine recurrence:

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

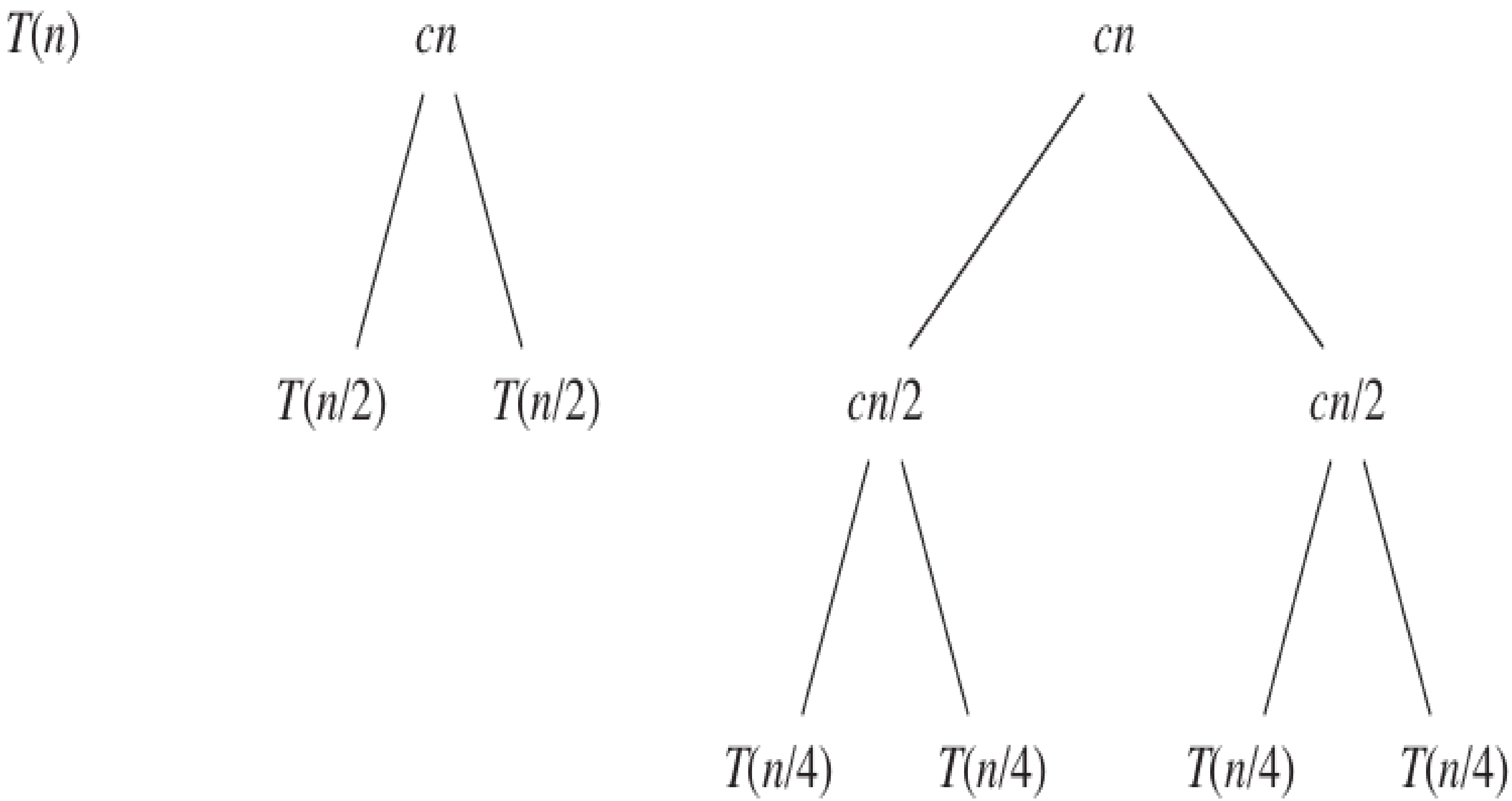
Rewriting we have:

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

This can be solve using the recursion tree as shown below.

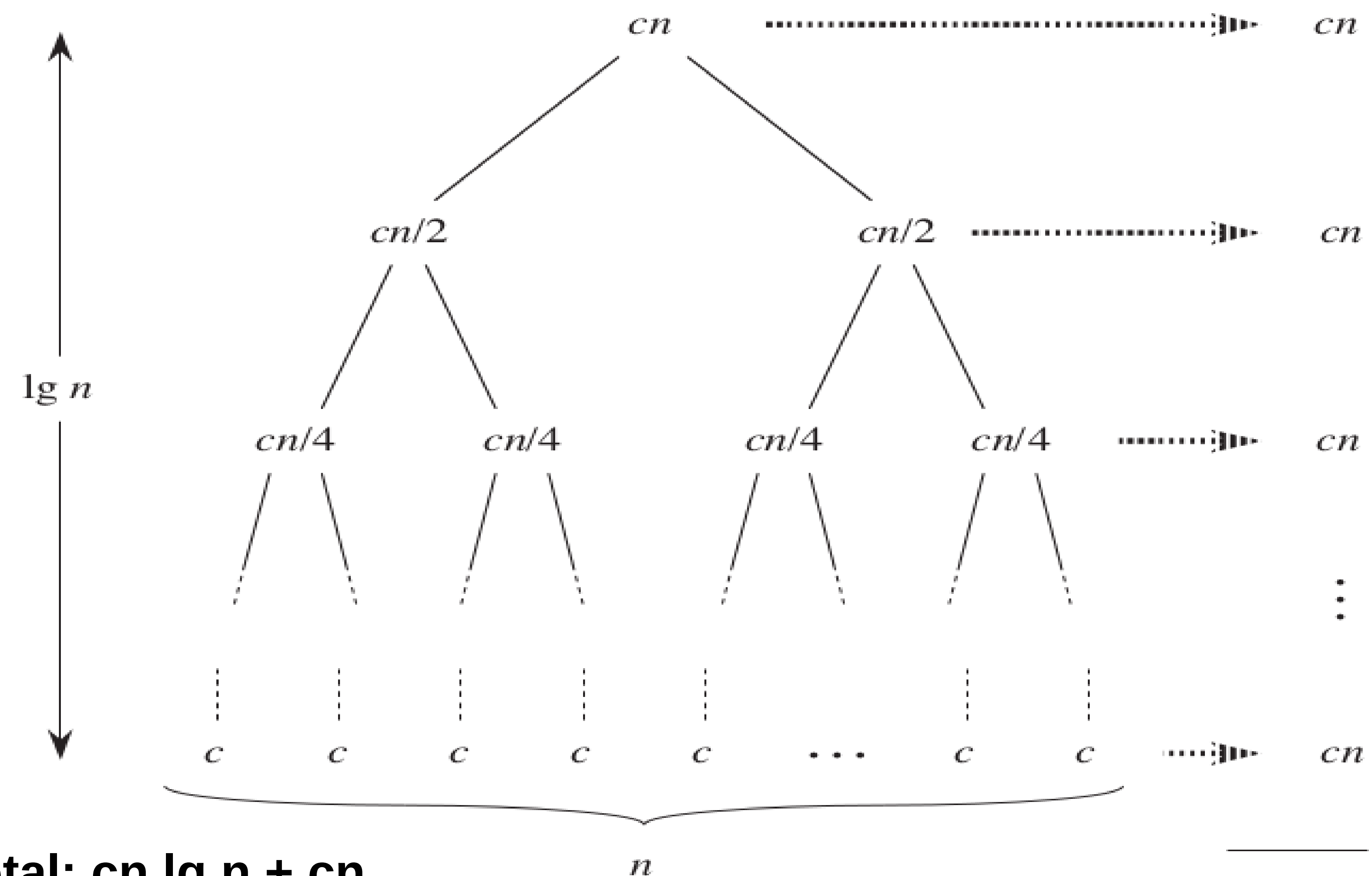
SORTING

MergeSort Algorithm



SORTING

MergeSort Algorithm



Total: cn lg n + cn
O(nlogn)

SELECTION

SELECTION ALGORITHM

This is an algorithm for finding the k^{th} smallest number in a list or array;

Such a number is called the k^{th} order statistic.

This includes the cases of finding the minimum, maximum, and median elements.

Selection problems are easily reduced to sorting,however they do not require the full power of sorting.

SELECTION

SELECTION ALGORITHM- DETERMINISTIC AND RANDOMIZED

$A[1..n]$.

$min = 1$;
for $j = 2$ to n do
  if $A[j] < A[min]$ then $min = j$ endif
endfor.

int RSELECT(int ℓ, r, i)
  $q = \text{RSPLIT}(\ell, r)$; $m = q - \ell + 1$;
  if $i < m$ then return RSELECT$(\ell, q-1, i)$
    elseif $i = m$ then return q
    else return RSELECT$(q+1, r, i-m)$
  endif.

SELECTION

SELECTION ALGORITHM

Let s = <e_1 , . . . , e_n> be a sequence
and let s' = <e_1' , . . . , e_n'> be the sorted version of it.

- Selection of the smallest element requires determining e_1' , selection of the smallest and the largest requires determining e_1' and e_n' ;
- The selection of the k-th largest requires determining e_k'.
- Selection of the median refers to selecting the $\lfloor n/2 \rfloor$ -th largest element.
- Selection of the median and also quartiles is a basic problem in statistics.
- It is easy to determine the smallest or the smallest and the largest element by a single scan of a sequence in linear time.
- k-th largest element can be determined in linear time.

SELECTION

SELECTION ALGORITHM

1. Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining n mod 5 elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups by first insertion-sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements.
3. Use SELECT recursively to find the median x of the $\lceil n/5 \rceil$ medians found in step 2. (If there are an even number of medians, then by our convention, x is the lower median.)
4. Partition the input array around the median-of-medians x using the modified version of PARTITION . Let k be one more than the number of elements on the low side of the partition, so that x is the k^{th} smallest element and there are k-n elements on the high side of the partition.
5. If i = k, then return x. Otherwise, use SELECT recursively to find the i^{th} smallest element on the low side if i < k, or the i - kth smallest element on the high side if i > k.

SELECTION

SELECTION ALGORITHM-RANDOMIZED

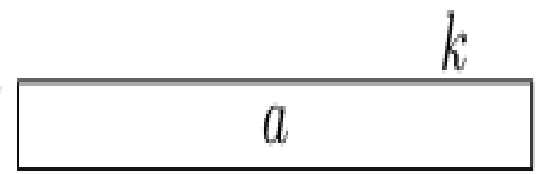
// Find an element with rank k

Function $select(s : Sequence$ **of** $Element; k : \mathbb{N}) : Element$

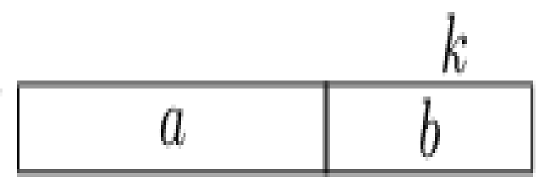
  assert $|s| \geq k$

  pick $p \in s$ uniformly at random // pivot key

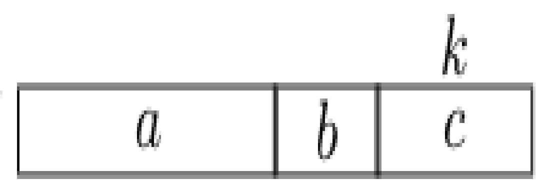
  $a := \langle e \in s : e < p \rangle$

  if $|a| \geq k$ **then return** $select(a, k)$ //


  $b := \langle e \in s : e = p \rangle$

  if $|a| + |b| \geq k$ **then return** p //


  $c := \langle e \in s : e > p \rangle$

  return $select(c, k - |a| - |b|)$ //


SELECTION

SELECTION ALGORITHM

s	k	p	a	b	c
$\langle 3,1,4,5,9,\mathbf{2},6,5,3,5,8\rangle$	6	2	$\langle 1\rangle$	$\langle 2\rangle$	$\langle 3,4,5,9,6,5,3,5,8\rangle$
$\langle 3,4,5,9,\mathbf{6},5,3,5,8\rangle$	4	6	$\langle 3,4,5,5,3,4\rangle$	$\langle 6\rangle$	$\langle 9,8\rangle$
$\langle 3,4,\mathbf{5},5,3,5\rangle$	4	5	$\langle 3,4,3\rangle$	$\langle 5,5,5\rangle$	$\langle\rangle$

SELECTION

SELECTION ALGORITHM- COMPLEXITY

Worst-case running time

For simplicity, assume that n is a multiple of 5 and ignore ceiling and floor functions.
The number of items less than or equal to the median of medians is at least 3n/10 in this context.
These are the first three items in the sets with medians less than or equal to the median of medians. I
Symmetrically, the number of items greater than or equal to the median of medians is at least 3n/10 .
The first recursion works on a set of n/5 medians, and the second recursion works on a set of at most 7n/ 10 items.
We have:
$T(n) <= n + T(n/5) + T(7n/10)$, that is $O(n)$

SELECTION

SELECTION ALGORITHM- COMPLEXITY

Worst-case running time

We have:

$T(n) <= n + T(n/5) + T(7n/10)$, that is $O(n)$ that can be proved using induction

Assume $T(m) \leq c \cdot m$ for $m < n$ and c a large enough constant;

$T(n) <= n + (c/5).n + (7c/10).n = (1+9c/10).n$

Tacking c values >=10, we have $T(n) <= c.n$

EXERCISES

(1)Estimate the running time of a program that has 2000 lines of sequential code of a procedural language.

(2) Estimate the running of a program that scans the input two times.

(3)Estimate the running time of a program that adds two nxn matrices.

(4)Estiamte the running time of a program that multiplies two nxn matrices.

(5)Estimate the running time of a program that uses binary search to locate an item from an unsorted array of size n.

(6) Estimate the running time of a program that requests n integers and displays their squares.

(7)Estimate the running time of a program that uses bubble sort technique to sort an array of n unsorted numbers.

(8)Estimate the running time of a program controlled by the loop: for (x=1; i<n;i++)

(9)Estimate the running time of a program controlled by the loop: for (x=1; i<n;i--)

EXERCISES

(1)Estimate the running time of a program controlled by the loop: for (x=1; i<n;i=I*2).

(2)Estimate the running time of a program controlled by the loop: for (x=1; i<n;i=i*4).

(3)Estimate the running time of a program controlled by the loop: for (x=1; i<n;i=i/2).

(4) Define recurrence relation and give an example.

(5)State the recurrence relations for Fibonacci series and Tower of Hanoi.

(6) Give a general formula of a linear recurrence equation.

(7)Work out a characteristic equation for the recurrence relation:

(1)$R_n = AR_{n-1} + BR$, for A, B being real numbers

(8)Give the steps of a recurrence relation

(9)Discuss the methods used to solve recurrence relations giving examples in each case.

(10)Describe the substitution method

(11)Describe the iteration method

(12)Describe the recursion tree method

EXERCISES

(1)Describe the master method
(2)Solve $T(n) = 9T(n/3)$ using Master Theorem
(3)Solve $T(n) = T(2n/3) + 1$ using Master Theorem
(4)Solve $T(n) = 8T(n/2) + 1000n^2$ using Master Theorem
(5)Solve $T(n) = n^2T(n/2) + n^2$ using Master Theorem
(6)Solve $T(n) = 64T(n/8) - n^2 \log n$ using Master Theorem
(7)Solve $T(n) = 4T(n/3) + n^2$ using Master Theorem
(8)Set up recursive algorithm based on $2^n = 2^{n-1} + 2^{n-2}$
(9)Describe selection sort
(10)Implement selection sort
(11)Discuss the complexity of selection sort
(12)Describe quicksort
(13)Implement quicksort
(14)Discuss the complexity of quicksort

EXERCISES

(1)Describe MergeSort
(2)Implement MergeSort
(3)Discuss the complexity of MergeSort
(4)Describe Selection algorithm
(5)Implement Selection algorithm
(6)Discuss the complexity of Selection algorithm