COMPREHENSIVE Conventional Algorithms Analysis

[Chapter 7 intends to provide a link between the conventional analysis and the probabilistic analysis of algorithms besides providing basic exposure to use of randomization as a tool to design efficient algorithms, and use of probabilistic analysis in analysis and design of algorithms]

1. Elements of Algorithm Design and Analysis

Understanding of algorithms essentially involves focusing on two concerns:

1. Design of algorithms
2. Analysis of Algorithms

1.1. **Understanding Systems and Problems**

I) System's Entities and their Properties – encoded as Mathematical function

a. Continuous functions – e.g. analog signals F= a*X^2+ b*Y^3; X, Y continuous {most of the properties/ parameters occurring in real life}
b. Discrete functions – F= a*X^2+ b*Y^3; X, Y discrete {all properties/ parameters recorded in digital domain}

II) Algorithm's Specification

a. Type – Search, Sorting, Optimization
b. Abstract Data Type-
   i. Data Organization – System Parameters and their Universe/ Domain
   ii. Process Specification – I/O Parameters , PreCondition, PostCondition, Algorithm
c. Solution implementation approach - Iterative/ Recursive
d. Effectiveness- Time, Space, Programming effort, Correctness….

IV) Programs

a. Functional
b. Object oriented

V) Optimization Problem

a. Elements of definition
   i. Optimization Criterion =
   ii. Constraints
b. Example = 0/1 knapsack criterion = **profit;** constraint= **knapsack's capacity**
c. Types of Algorithms for Optimization Problems
   i. Deterministic (Hard computing)- Greedy, Dynamic Programming, Branch and Bound, Backtracking
   ii. Approximate-
      1. Probabilistic/ Randomized/ Stochastic Methods
      2. Soft Computing Algorithms – Neural Networks, Evolutionary Algorithms, etc.

1.2.**Properties of an Algorithm**

Definition 1.1 [Algorithm]: An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. **Input.** Zero or more quantities are externally supplied.
2. **Output.** At least one quantity is produced.
3. **Definiteness.** Each instruction is clear and unambiguous.
4. **Finiteness.** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness.** Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible. □

VI) Design of Algorithms

Design of an algorithm to solve a particular problem is guided by understanding of basic algorithm design techniques (approaches) with regard to

a. Characteristics of the problem to which the technique is applicable
b. Formulation of the problem in terms of the technique
c. Basic steps of solving the problem according to the technique
d. Typical (example) problems to which that technique is applicable
e. Limitations of the technique

Some Algorithm design techniques are

I) Divide and Conquer: e.g. Binary Search, QuickSort, MergeSort etc.
II) Greedy Technique: e.g. Knapsack problem, Min. Spanning Tree etc.
III) Back Tracking: e.g. N-Queen Problem etc.
IV) Branch & Bound: e.g. 15-puzzle problem etc.
V) Dynamic Programming: e.g. 0/1 knapsack (technique: generating profit values for all (element, remaining capacity) pair), TSP problem etc.
VI) Evolutionary Algorithms

VII) Analysis of Algorithms

We need algebraic and logical ability to estimate the run-time performance of algorithms, so that we can compare two or more algorithms available for performing the same task. We need to analyze an algorithm about:

a. Time Complexity
b. Space Complexity
c. Correctness
d. Effect of data characteristics on the performance of algorithm: Best-case, Worst-case, Average-case performance.

2. Analyzing Time complexity of Iterative Algorithms

2.1 General Rules

In analyzing the time complexity of an algorithm, we normally ignore actual CPU-cycles of each instruction execution. Rather we aim to approximately estimate the growth of the run-time of the algorithm with respect to input data size, n. For this, we just calculate the step count.

Some thumb rules we may use for estimating time complexity of an iterative algorithm involving the major language constructs can be observed as suggested

I) for a statement block t (n) = count of the number of statements.

II) for each single level loop involved: t (n) = no. of iterations of the loop.

III) for each nested loop: t (n) = (no. of iterations of the outer loop) * (no. of iterations of the inner loop).

IV) for control structure: t (n) = larger of the counts in the '***if***'' and the '***else***' part statement blocks.

We can refer to the following examples to make the idea clear.

```
i)    Algo1 (n)                              Step Count
         1.    P:= 1                             1
         2.    for I = 1 to 2n do
         3.          P:= P * I                   2n
                                               =========
```

Estimating time complexity: f(n) = 2n + 1.

```
ii)   Algo2 (m,n)                            Step Count
         1.    P:= 1                             1
         2.    for I = 1 to m do
         3.          for J = 1 to n do
         4.                P:= P * I             m*n
                                               =========
```

Estimating time complexity: f(n) = m*n + 1.

```
iii)  Algo3 (n)                                  Step Count
         1.    P:= 1                                  1
         2.    if (<cond1>) then
         3.          for J = 1 to n do
         4.                P:= P * I                        [… n …]
         5.    else
         6.          for J = 1 to 2n do
         7.                P:= P * I                        [… 2n …]
         8.    end if                                 2n [since 2n > n]
                                          =========
```

Estimating time complexity: f(n) = 2n + 1.

But the basic rules are not always helpful. **Sometimes we need to apply reasoning** to estimate the step counts.

For example, we consider the following algorithm.

```
iv)   Algo4 (n)                                  Step Count
         1.    P:= 1                                  1
         2.    for I = 1 to n do
         3.          for J = 1 to I do
         4.                P:= P * I                ???
                                                 ========
```

Here, we need to apply reasoning. We observe that step counts of (P:=P*I): it executes 1 time for I = 1; 2 times for I = 2; and so on

So, estimating time complexity, we get

$$f(n) = 1 + 2 + 3 + \ldots + n + 1 = \sum_{1}^{n} i + 1 = \frac{n(n+1)}{2} = \tfrac{1}{2} n^2 + \tfrac{1}{2} n + 1.$$

Further, we observe that such algorithms may take other forms also.

```
v)    Algo5 (n)                                  Step Count
         1.    P:= 1                                  1
         2.    for I = 1 to n do
         3.          for J = 1 to I² do
         4.                P:= P * I          1² + 2² + 3² + ... + n².
```

Estimating time complexity: $f(n) = \sum_{1}^{n} i^2 + 1$

```
vi)   Algo6 (n)                                  Step Count
         1.    P:= 1                                  1
         2.    for I = 1 to n do
         3.          for J = 1 to log(I) do
         4.                P:= P * I          log(1)+log(2)+...+log(n).
```

Estimating time complexity: $f(n) = \sum_{1}^{n} \log(i) + 1$

In the due course, we see that we need to know the algebraic series and summations. A quick go through *section 2.2* and solving the exercises in *section 2.3* may refresh the related concepts.

2.2 Algebraic Series and Other Relevant Mathematical Results Review

Arithmetic Series

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{1}{2} n(n+1) \qquad \textbf{--- [I]}$$

$$\sum_{k=1}^{n} k^2 = 1^2 + 2^2 + \ldots + n^2 = \frac{1}{6} n(n+1)(2n+1) \qquad \textbf{--- [II]}$$

$$\sum_{k=1}^{n} k^3 = 1^3 + 2^3 + \ldots + n^3 = \frac{1}{4} n^2(n+1)^2 \qquad \textbf{--- [III]}$$

Geometric Series

For real $x \neq 1$, $|x| < 1$,

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} \qquad \textbf{--- [IV]}$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \qquad \textbf{--- [V]}$$

For real $x \neq 1$, $|x| > 1$,

$S_n = a_1 + a_1x + a_1x^2 + \ldots + a_1x^{n-1} = a_1(1-x^n)/(1-x)$

Harmonic Series

For positive integers n,

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k} = \ln(n) + c, \qquad \textbf{--- [VI]}$$

where *c is positive integer constant.*

Additional formulae can be gained by integrating or differentiating the formulae above. e.g. by differentiating both sides of infinite geometric series [v], we get:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1 - x)^2}, \text{ for } |x| < 1 \qquad \textbf{--- [VII]}$$

Telescopic Series

For any sequence $a_0, a_1, \ldots, a_n$,

$$\sum_{k=0}^{n} (a_k - a_{k-1}) = a_n - a_0 \qquad \textbf{--- [VIII]}$$

since, each of the terms $a_0, a_1, \ldots, a_{n-1}$ is added in exactly once and subtracted once. We say that the sum telescopes.

Similarly,

$$\sum_{k=0}^{n} (a_k - a_{k+1}) = a_0 - a_n \qquad \textbf{--- [IX]}$$

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} = \sum_{k=1}^{n-1} \left(\frac{1}{k} - \frac{1}{k+1}\right) = 1 - \frac{1}{n} \qquad \textbf{--- [X]}$$

Some More Results

Exponentials

$a^0 = 1$, $a^1 = a$, $a^{-1} = 1/a$, $(a^m)^n = a^{mn}$, $(a^m)^n = (a^n)^m$, $a^m a^n = a^{m+n}$

For all real a,b, s.t. a>1, $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$

--- [XI]

Logarithms

$lg\ n = log_2\ n$, $ln\ n = log_e\ n$, $lg^k n = (lg\ n)^k$, $lg\ lg\ n = lg\ (lg\ n)$,

For all real a, b, c >0

$$a = b^{\log_b a}, \log_c(ab) = \log_c a + \log_c b, \log_b a^n = n \log_b a,$$

$$\log_b a = \frac{\log_c a}{\log_c b}, \log_b(1/a) = -\log_b a, \log_b a = \frac{1}{\log_a b}$$

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

--- [XII]

Factorials

$$n! = \begin{cases} 0, & n = 0 \\ n.(n-1), & n > 0 \end{cases}$$

Stirling's Approximation

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right) \quad \textbf{--- [XIII]}$$

$for \;\; n \geq 1$,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha_n}, \; \frac{1}{12n+1} < \alpha_n < \frac{1}{12n} \quad \textbf{--- [IX]}$$

Use: Applying Stirling's approximation, we get ***lg*(n!) = *Θ*(n *lg* n)** *[the Θ-notation to be introduced later]*

L-Hospital Rule for Solving Limits

When, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{\infty}{\infty} \; or \; \frac{\infty}{0}$ form,

Then, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{d(f(n)/dn}{d(g(n)/dn}$ **--- [X]**

2.3 Relevant exercises on solving arithmetic series

1. Evaluate the following

   a) $\sum_{k=1}^{n} (2k - 1)$ b) $\sum_{k=1}^{n} (2k + 1)x^{2k}$ c) $\prod_{k=1}^{n} 2.4^k$ d) $\prod_{k=2}^{n} (1 - \frac{1}{k^2})$

   e) $\sum_{1}^{n} \log(i)$

2. Verify the following:

   a) $\sum_{k=1}^{n} \frac{1}{2k-1} = \ln(\sqrt{n}) + c$ b) $\sum_{k=0}^{\infty} k^2x^k = \frac{x(1+x)}{(1-x)^3}, \forall 0 < |x| < 1$

   c) $\sum_{k=0}^{\infty} \frac{(k-1)}{2^k} = 0$

3. Analyzing Time complexity of Recursive Algorithms

3.1. Recursive Algorithms and Recurrences

A recursive algorithm and its corresponding step count can be considered to have typically the following scheme.

Single Recursive Call

```
RecAlgo1 (n)                                  Step Count
      1.    program termination condition      {program terminates for n=n0}
      2.       corresponding statements        f1(n) {gives basic case count}
      3.    [ statements ]                     f2(n)
      4.    RecAlgo1(n')                       t(n'){general recurrence for
                                               recursive step count}
      5.    [ statements ]                     c3
```

i.e. an algorithm having a mix of statement blocks having constant counts and those having count as linear function of the input size *n*.

Representing time complexity: $T(n) = f1(n)$, for $n = n_0$;

$= T(n') + f(n)$, for $n > n0$;

Here, $f(n) = f2(n) + c3$, $f1(n)$ may give a constant

$n' = f(n)$ = modified parameter as function of n.

Alternative representation as follows.

$T(n_0) = f1(n)$, $T(n) = T(n') + f(n)$.

Example 3.2. Factorial

```
iFactorial (n) { /*if n==0 return 1;*/ f=1; while(n>0) {f=f*n; n=n-1;}}
rFactorial( n )
      1.    if  ( n==0 )                          1          T(0)=1
      2.                return 1
      3.    return n* rFactorial (n-1)            T(n-1)+1
```

T(n) = T(n-1) + 2; T(0) = 1

Q - Recursive function's execution implicitly involves system stack.<= Overhead of ~

rFact(4) = 4*rFact(3) = 4*3*rFact(2) = 4*3*2*rFact(1) =4*3*2*1*rFact(0) = 4*3*2*1*1= 4*3*2 = 4*6= 24

Q- What are the steps involved during a function-call-return => recursion has overhead

Q- Why recursion is useful even when it has overhead? => programmers' convenience for originally recursive definitions. Q. Write iterative program for Fibonacci, IN_ORD_BST

Q- Write recursive function to sum the values of array, linked_list

Example 3.2. Binary Search

```
Bin_Srch( Ar [], l, u, v )
      1.    if  ( l == u )                                    T(1)=2
      2.                if ( Ar[l] == v ) return l;
      4.          else            return false;
      5.    m = (l+u) /2;
      6.    if  (v == Ar[m])      return m;
      7.    if  (v < Ar[m])       Bin_Srch(Ar, l, m-1, v);  T(n/2)
      8.    if  (v > Ar[m])       Bin_Srch(Ar, m+1, u, v);  T(n/2)
```

Estimating time complexity for Bin_Srch(n):

Here we see that Bin_Srch() recursive calls are made over half (n/2) of the portion (n) of input list. Moreover, only one of the recursive calls is executed at any call. Therefore, the recurrence for the time complexity representation becomes,

$T(n) = T(n/2) + C$; where C = no. of supplementary steps other than the recursive calls = max(4 (step 7) or 5(step8)) = 5.

So, the complete recurrence representing time complexity of Bin_Srch algorithm is:

$T(n) = 2$, *for* $n = 1$

$= T(n/2) + 5$ *for* $n > 1$

Alternative, $T(n) = T(n/2) + 5$, with $T(1) = 2$.

Multiple Recursive Calls

```
RecAlgo1 (n)                                  Step Count
      1.    program termination condition     {program terminates for n=n0}
      2.          corresponding statements    f1(n) {gives base case count}
      3.    [ statements ]                     c3
      4.    RecAlgo1(n1)                      t(n1){general recurrence for
                                              recursive step count}
      5.    [ statements ]                     c4
      6.    RecAlgo1(n2)                      t(n2)
      .
      .
      .     RecAlgo1(nn)                      t(nn)
      k.    [ statements ]                     cn
```

Representing time complexity:

$T(n) = f1(n)$, for $n = n_0$;

$= \sum_{i=1}^{n} T(n_i) + f(n)$, for $n > n_0$;

Here, *f1(n)* may be constant;

f(n) = function representing contribution of non-recursive calls

$n_i = f_i(n)$, $1 <= i <= n$. i.e. all recurrences with different modified parameters are summed up.

Multiple Recursive Calls with Same Parameter Modification Function

If all the recursive calls use the same function, say n' =f(n), for modifying the parameter for the recursive call, and let *m* such calls are there, then the recurrence derived above can be written as:

$T(n) = f1(n)$, for $n = n_0$;

$= m\,T(n') + f(n)$, for $n > n0$;

Moreover, ***if each n' = n/b***, for some positive constant *b*, then

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

Example 3.3. Merge Sort

```
M_Sort( Ar[], l, u )                    Step Count
      1.    if  ( l == u )                  1  {base case: n0 = 1; T(1) = 1}
      2.                return;
      3.    m = (l+u)/2;                    1
      4.    M_Sort (Ar, l, m-1);            T(n/2) { size(l to m-1) = n/2 }
      5.    M_Sort (Ar, m+1, u);            T(n/2) { similarly  }
      6.    Merge  (Ar, l, m, m+1, u);      cn     { Merge is an iterative
                                 algorithm and its step count is
                              given by a linear function of n
                                            (... ... verify!!)}
```

Estimating time complexity for M_Sort(n):
Here we see that both the steps 4 and 5 are executed on each recursive call of the function. So, we observe the recurrence to be

$T(n) = 1$, for $n = 1$

$= 2\,T(n/2) + f(n) + 2$, for $n > 1$, where $f(n)$ is a linear function of n.

Review Exercise:

1. Write the iterative algorithm for Merge() function in the M_Sort algorithm. Verify its time complexity.

2. Write the recursive Fibonacci () function. Determine the recurrence representing its time complexity.

3. Write the Recursive Inorder() function for traversing a binary tree. Determine its recurrence for a complete binary tree.

4. Write the Quicksort() algorithm. Determine its recurrence assuming that in each recursive call, the list is divided into two equal parts.

5. Devise a binary search algorithm that splits the set not into two sets of (almost) equal sizes but into two sets, one of which is twice the size of the other. Determine its recurrence.

3.2. Solving Recurrences-I

We need to solve the recurrence obtained for a recursive algorithm. In the following, we see solutions to some typical recurrences.

3.2.1 Substitution Method

I. The following recurrence arises for a recursive program that loops through the input to estimate one item:

T(N) = T(N-1) + N for N > 1 with T(1) = 1.

Solution: {T(3) = T(2)+3 = T(1)+3+2= 1+ Sum(2+3)
T(4) = T(3)+4= T(2)+3+4 = T(1)+2+3+4 = 1+ Sum(2+3+4) }

T(N) = T(N-1) + N [// N+T(N-1)]
= T(N-2) + (N-1) + N
= T(N-3) + (N-2) + (N-1) + N
…
=T(N-k) + (N-(k-1)) +….+(N-1) + N
//(For k=N-1 :) = T(1) + 2 + … + (N-2) + (N-1) + N
= 1 + 2 + … + (N-2) + (N-1) + N
$= \frac{N(N+1)}{2}$
$= \frac{N^2}{2} + \frac{N}{2}$

II. The following recurrence arises for a recursive program that halves the input in one step.

T(N) = T(N/2) + C for N > 1 with T(1) = C1 ;

C, C1 being positive constants.

Solution:

Let N = 2^k i.e. k = log_2 N.
T(N) = T(N/2) + C
= T($N/2^2$) + C+ C [(N / 2) /2 = N/(2*2) = $N/2^2$]

$= T(N/2^3) + C + 2C$
…
$= T(N/2^k) + C + (k-1)C$
$= T(1) + k\,C$ [since $N = 2^k$, hence, $N/2^k = 1$]
$= k\,C + C1$ [since $T(1) = 1$]
$= C.\boldsymbol{log_2 N + C1}$

[***Note:*** *The recurrence of Bin_Srch() conforms to this recurrence model, with C = 5, C1 = 2*]

III. The following recurrence arises for a recursive program that halves the input for the next call, but alongside must examine every item in the input.

T(N) = T(N/2) + N for N > 1 with T(1) = 0.

Solution:

Let $N = 2^k$ i.e. $k = log_2 N$.
$T(N) = T(N/2) + N$
$= T(N/2^2) + N/2 + N$ [$(N / 2) /2 = N/(2*2) = N/2^2$]
$= T(N/2^3) + N/4 + N/2 + N$
…
$= T(N/2^k) + N/2^{k-1} + \ldots + N/4 + N/2 + N$
$= T(1) + 2 + \ldots + N/4 + N/2 + N$ [since $N = 2^k$, hence, $N/2^k = 1$ and $N/2^{k-1} = 2 * N/2^k = 2$]
$= 0 + 2 + \ldots + N/4 + N/2 + N$
$= N + N/2 + N/4 + \ldots + N/2^k$
$= N\,(1 + ½ + (½)^2 + \ldots + (½)^k)$
$= N \dfrac{\left(\frac{1}{2}\right)^{k+1} - 1}{\left(\frac{1}{2}\right) - 1}$

The result above is a bit clumsy to interpret. So, it can be approximated to estimate the order of growth of the resultant function. For this, we look for the result of summation for the case, $k \to \infty$, i.e. above series becomes a infinite geometric series.

Hence, $T(N) \cong N\,(1 + ½ + (½)^2 + \ldots \ldots \infty$ terms)

$= N.\dfrac{1}{1 - \frac{1}{2}}$
$= \mathbf{2N}$

IV. The following recurrence arises for a recursive program that has to make a linear pass through the input, before, during, or after splitting that input into two equal halves.

T(N) =2T(N/2) + N for N > 1 with T(1) = *C* ;
C being positive constant.

Solution:

Let N $= 2^k$ i.e. $k = log_2 N$.
T(N) $= 2\,T(N/2) + N$
$= 2\,[2\,T(N/2^2) + N/2\,] + N = 2^2\,T(N/2^2) + 2 * (N/2) + N$
$= 2^2\,[2\,T(N/2^3) + (N/2^2)] + 2N = 2^3\,T(N/2^3) + 2^2 * (N/2^2) + 2N$
$= 2^3\,T(N/2^3) + 3N$
…
$= 2^k\,T(N/2^k) + kN$
$= \mathbf{2}^{\log_2 N}\,T(1) + N.\,log_2 N$
$= N\,T(1) + N.\,log_2 N$ [since $\mathbf{2}^{\log_2 N} = N^{\log_2 2} = N$, ***Ref. XII***]
$= N * C + N.\,log_2 N$
$= \mathbf{N.\,log_2 N + C.N}$

[***Note:*** *The recurrence of M_Sort() conforms to this recurrence model, as f(n)+2 being a linear function of N can be approximated as N; C = 1.*]

T(N) =2T(N/2) + C for N > 1 with T(1) = *D* ; C, D being positive constant.

Solution:

Let N $= 2^k$ i.e. $k = log_2 N$.
T(N) $= 2\,T(N/2) + C$
$= 2\,[2\,T(N/2^2)\,C\,] + C = 2^2\,T(N/2^2) + C(1+2)$
$= 2^2\,[2\,T(N/2^3) + C] + C(1+2) = 2^3\,T(N/2^3) + C(1+2+2^2)$
…
$= 2^k\,T(N/2^k) + C(1+2+2^2+\ldots 2^{k-1})$ []
[since, $S_n = a_1 + a_1x + a_1x^2 + \ldots + a_1x^{n-1} = a_1(1-x^n)/(1-x)$]

$= \mathbf{2}^{\log_2 N}\,T(1) + C(\mathbf{2}^{\log_2 N} - 1)$
$= N\,T(1) + C.(N-1)$ [since $\mathbf{2}^{\log_2 N} = N^{\log_2 2} = N$, ***Ref. XII***]
$= N * D + C.N - C$
$= \mathbf{(C+D)N-C}$

Review Exercise:

1. Solve following recurrences using the induction method:

i) T(N) = T(N/2) + 1 for N > 1 with T(1) = *C* ; C being positive constant.

ii) T(N) = T(N-1) + N, T(1) = 1

iii) T(N) = T(N/2) + N, T(1) = 0

iv) T(N) = 2 T(N/2) + C, T(1) = 0, C being positive constant.

v) T(N) = 2 T(N/2) + N, T(1) = 1

2.

3.2.2 Substitution Method

3.2.3 Change of Variable Method

1. **T(N) = 2 T(√N) + log N, T(1) = 1**

Solution: With change of variable assuming $N = 2^m$ i.e. m=logN yields

$T(2^m) = 2T(2^{m/2})+m$

Renaming $S(m) = T(2^m)$ the recurrence can be rewritten as

$S(m) = 2S(m/2) + m$

Standard solutions of such model recurrence gives $S(m) = m\log m + cm$

Changing back from S(m) to **T(N) we get**

T(N) = T$(2^m) = S(m) = \Theta(m\log m) = \Theta(\log N \log\log N)$

2. **T(N) = 2 T(N/2) + Nlog N, T(1) = 1**

Let us take $n = 2^m$. Then we have the recurrence

$$T(2^m) = 2T(2^{m-1}) + 2^m \log_2(2^m) = 2T(2^{m-1}) + m2^m$$

Calling $T(2^m)$ as $f(m)$, we get that

$$\begin{aligned} f(m) &= 2f(m-1) + m2^m \\ &= 2(2f(m-2) + (m-1)2^{m-1}) + m2^m \\ &= 4f(m-2) + (m-1)2^m + m2^m \\ &= 4(2f(m-3) + (m-2)2^{m-2}) + (m-1)2^m + m2^m \\ &= 8f(m-3) + (m-2)2^m + (m-1)2^m + m2^m \end{aligned}$$

Proceeding on these lines, we get that

$$\begin{aligned} f(m) &= 2^m f(0) + 2^m(1+2+3+\cdots+m) = 2^m f(0) + \frac{m(m+1)}{2}2^m \\ &= 2^m f(0) + m(m+1)2^{m-1} \end{aligned}$$

Hence, $T(n) = nT(1) + n\left(\frac{\log_2(n)(1+\log_2(n))}{2}\right) = \Theta(n\log^2 n)$.

4. Asymptotic Analysis

4.1 Asymptotic Notations: Basic definitions and Relevance

Our purpose is to estimate the order of growth of run time of an algorithm. The concept of asymptotic notations helps in estimating and representing the order of growth of functions. We usually try to compare the representative function - of the algorithm's input size N - with respect to some common asymptotic functions. For example,

1 (or any integer) – *Constant; log* (N) – *Logarithmic*; $C_1.N + C_2$ – *Linear*; N *log* (N) - N *log*(N); $C_1.N^2 + C_2$ – *Quadratic*; $C_1.N^3 + C_2$ – *Cubic*; in general, $C_1.N^p + C_2$ – *Polynomial*; $C_1.C_2^N$ – *Exponential*; N! – *Factorial*.

Definitions

Big- Oh Notation

The Big-Oh notation expresses an ***upper bound*** for the function *f(N)*.

$$f(N) = O(g(N)) \quad iff \quad f(N) <= c.\ g(N)\ ; \qquad \forall N \geq n_0; 0 \leq c, n_0 \leq \infty$$

Omega Notation

The *Big-Omega*-notation expresses an ***lower bound*** for the function *f(N)*.

$$f(N) = \Omega(g(N)) \quad iff \quad f(N) >= c.\ g(N); \qquad \forall N \geq n_0; 0 \leq c, n_0 \leq \infty$$

Theta Notation

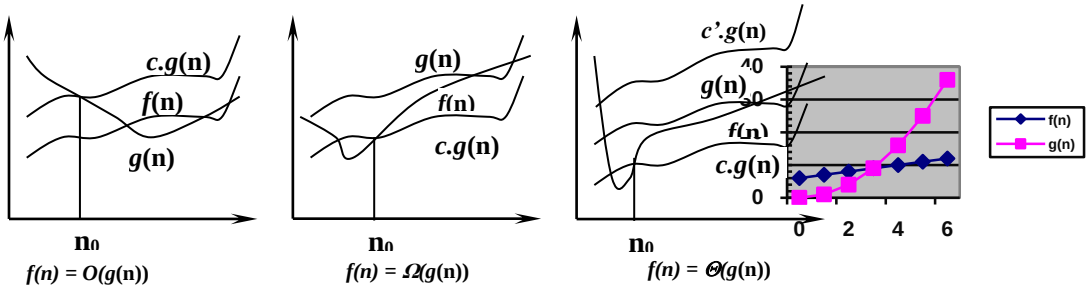
The *Theta*-notation expresses an ***exact bound*** (or *equivalence*) for the function *f(N)*.

$$f(N) = \Theta(g(N)) \quad iff \quad c_1.g(N) <= f(N) <= c_2\ .\ g(N);$$

$$\forall N \geq n_0; 0 \leq c_1, c_2, n_0 \leq \infty$$

We may also say that ***f(N) = Θ(g(N)) iff f(N) = O(g(N)) and f(N) = Ω(g(N)).***

Interpreting meanings of *upper – lower bounds* with graphical representation



The relevance of the value n_0 can be understood by the above plot (rightmost) for $f(n) = n+6$, $g(n) = n^2$. We observe, **$n_0 = 3$**, since $f(n)<g(n)$ before N=3; $f(n)=g(n)$, at N=3, but $f(n)>g(n)$ after N=3.

4.2 Estimating order of growth of functions

Comparison of Order of Growth of Some Standard Reference Functions

$$\textit{Constant} <. \textit{log}\,(N) <. N <. N\,\textit{log}\,(N) <. N^2 <. N^3 <. N^p <. C_2^N$$

Here, the relation **<.** shows ascending order of growth of the corresponding functions.

To estimate the order of growth of a given function *f(n)*, we first need to *guess* a function *g(n)* to compare with. As our choice, we guess a standard function nearest in comparison (*upper or lower bound*) to a major element in *f(n)*.

e.g. Consider the function $f(n) = 2n^3+5n+4$, we may choose to compare it with $g(n) = n^3$. In another function $f(n) = 2n^2+5n\log n+4n$, we may choose to compare it with $g(n) = n^2$, or with $g(n) = n\log n$.

Sometimes, we are provided the functions *f(n)* and *g(n)*, are required to compare the two.

e.g. We may be categorically asked to compare the function $f(n) = 2n^2+5n+4$ with $g(n) = n\log_2 n$, or with $g(n) = n$.

Particularly, when it concerns to the main purpose of all these endeavors, that is comparison of performance of two given algorithms (for the same task), the functions representing time complexities of the two algorithms simply work as *f(n)* and *g(n)*.

e.g. Suppose two algorithms *A* and *B* have time-complexity functions as I) $5n\log n+2n$, and II) n^2+10n; then we can have either $f(n) = 5n\log n+2n$, $g(n) = n^2+10n$, or $f(n) = n^2+10n$, $g(n) = 5n\log n+2n$.

4.2.1 Tabular Method

As a crude method to determine (if exist) the values for $\boldsymbol{c}$, and $\boldsymbol{n_0}$ so that the $\boldsymbol{O}$, $\boldsymbol{\Omega}$ or $\boldsymbol{\Theta}$ relations between two given functions *f(n) and g(n)* could be verified, we can use a trial and error approach. The trial values can be tabulated; exemplified as below.

Example 4.1.

Let $f(n) = 5n^2+2$, $g(n) = n^2$.

The table given below suggests that

$f(n) <= c.\ g(n)$, for $n_0 = 1$, and $c = 7$

So, $f(n) = 5n^2+2 = \boldsymbol{O}(n^2)$ for $n_0 = 1$, and $c = 7$. [4.1.a]

n	***f(n)*** = $5n^2+2$	***g (n)*** = n^2	***C***	***c.g(n)***	***f(n)<= c.g(n)?***
0	2	0	1	0	No
1	7	1	1	1	no (→ set *c* =7 to get ***f(n)= c.g(n)***)
1	7	1	7	7	yes (→check further for same *c* value)
2	22	4	7	28	yes (→verify more)
3	45	9	7	63	yes (→seems verified,….Stop)

Similarly, the table given below suggests that

f(n) >= c. g(n), for $n_0 = 0$, and $c = 1$;

So, *f(n)* = $5n^2+2$ = $\boldsymbol{\Omega}(n^2)$ for $n_0 = 1$, and $c = 1$. [4.1.b]

n	***f(n)***	***g(n)***	***C***	***c.g(n)***	***f(n)>= c.g(n)?***
0	2	0	1	0	yes
1	7	1	1	1	yes (→verify more)
2	22	4	1	4	yes (→verify more)
3	45	9	1	9	yes (→seems verified,….Stop)

From observations 4.1.a, and 4.1.b above, we may conclude that

c_1. *g(n)* <= *f(n)* <= c_2. *g(n),* for $n_0 = 1$, $c_1=1$, and $c_2 = 7$.

Hence, by definition of the *Theta*-notation

$5n^2+2 = \boldsymbol{\Theta}(n^2)$ for $n_0 = 1$, $c_1=1$, and $c_2 = 7$. [4.1.c]

In short, we may also say that since $5n^2+2 = \boldsymbol{O}(n^2)$, and $5n^2+2 = \boldsymbol{\Omega}(n^2)$ hence ,

$5n^2+2 = \boldsymbol{\Theta}(n^2)$.

Review Exercise:

0. Value of C to prove f(n)=$\boldsymbol{\Theta}$(g(n)) for f(n) = n^2+50n+10 g(n) = n^2 is ___

1. Find some f(n) and g(n), such that f(n)$\neq$ $\boldsymbol{\Theta}$(g(n)). Use the tabular method to verify.

2. Find some f(n) and g(n), such that f(n)=O(g(n), but f(n)$\neq$ $\boldsymbol{\Omega}$ (g(n)). Use the tabular method to verify.

3. Find some f(n) and g(n), such that f(n)= $\boldsymbol{\Omega}$ (g(n)), but f(n)$\neq$ O(g(n). Use the tabular method to verify.

4. Find some f(n) and g(n), such that f(n)=O(g(n), but g(n)$\neq$ O(f(n). Use the tabular method to verify.

4.2.2 Ratio Theorem

Let *f(n)* and *g(n)* be some asymptotically non negative functions of *n*. Then the following properties are defined.

Big- Oh Notation (Loose upper boud) ~ f(N)<=g(N)

f(N) = O(g(N)) ***iff*** $\lim_{n\to\infty} \frac{f(n)}{g(n)} \le c$, for some finite constant c.

Big- Omega Notation (Loose lower boud) ~ f(N)>=g(N)

f(N) = Ω(g(N)) ***iff*** $\lim_{n\to\infty} \frac{g(n)}{f(n)} \le c$, for some finite constant c.

Theta Notation (Equivalent) ~ f(N)=g(N)

f(N) = Θ(g(N)) ***iff*** lim (n->Inf) f(n)/ g(n) =c [$\lim_{n\to\infty} \frac{f(n)}{g(n)} \le c$ ***and*** $\lim_{n\to\infty} \frac{g(n)}{f(n)} \le c$, for some finite constant c.]

Small- Oh Notation (Tight upper boud) ~ f(N)<g(N)

f(N) = o(g(N)) ***iff*** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Small- Omega Notation (Tight lower boud) ~ f(N)>g(N)

f(N) = ω(g(N)) ***iff*** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Example 4.2.

Let *f(n)* = 3n + 2, *g(n)* = n.

Then, $\lim_{n\to\infty} \frac{(3n + 2)}{n} = 3$, ➔ 3n + 2 = ***O*** (n)

and $\lim_{n\to\infty} \frac{n}{(3n + 2)} = \frac{1}{3} < 3$ ➔ 3n + 2 = **Ω** (n)

Thus, with c = 3, we determine that 3n + 2 = ***Θ*** (n)

Example 4.3. [use of L-Hospital Rule – *Ref. X*]

Consider *f(n)* = $5n^2+3n+2$, *g(n)* = n^2.

Here, $\lim_{n\to\infty} \frac{(5n^2 + 3n + 2)}{n^2} \approx \frac{\infty}{\infty}$ *form* , so we apply L-Hospital's rule.

Then, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{d(5n^2 + 3n + 2) / dn}{d(n^2) / dn} = \lim_{n\to\infty} \frac{10n + 3 + 0}{2n} \approx \frac{\infty}{\infty}$ *form*

$$= \lim_{n\to\infty} \frac{d(10n + 3) / dn}{d(2n) / dn} = \frac{10 + 0}{2} = 5$$

Similarly, $\lim_{n\to\infty} \frac{n^2}{(5n^2 + 3n + 2)} \approx \frac{\infty}{\infty}\ form$, We get, $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \frac{1}{5} < 5$

Thus for c =5, we see, $f(n) = \boldsymbol{O}(n^2)$, $f(n) = \boldsymbol{\Omega}(n^2)$, and $f(n) = \boldsymbol{\Theta}(n^2)$.

Review Exercise:

1. Determine the asymptotic relations between following pairs of functions:

i) ($3n^2 + 100\ n + 6$; n^3) **ii) ($2n^3 + 4n^2 \log(n)$; n^3)** **iii) ($2n^3 + 4n^2\log(n)$; $n^3 \log(n)$)**

2. Determine the order of growth the function f(n) = 3 log n + log log n.

3. Estimate the lower bound on growth of the function f(n) = n3 log n. Verify.

4. Estimate time complexity of following:

```
int j = 0;
    for(int i = 0; i < n; ++i) {
        while(j < n && arr[i] < arr[j]) {
            j++;
        }
    }
```

4.3 Solving Recurrences-II

4.3.1 Recursion Tree Method

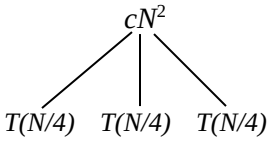
The recursion tree method is best to generate a good guess. It is useful when the recurrence describes the running time of a divide-and-conquer algorithm.

Example 4.4.

Let us consider the recurrence

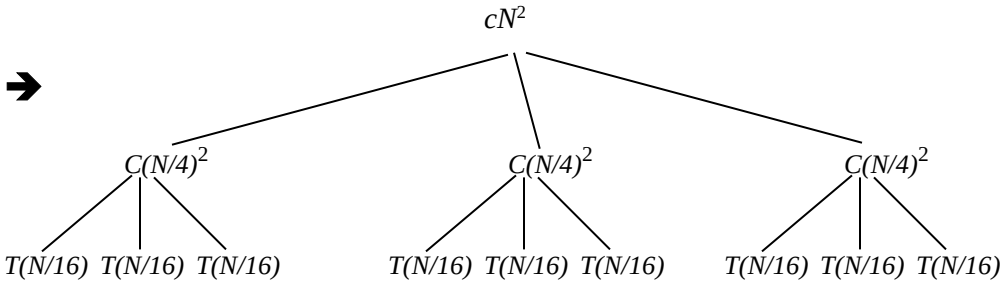
$T(N) = 3T(N/4) + \Theta(n^2)$, $T(1)=1$.

$T(N)$ ➔



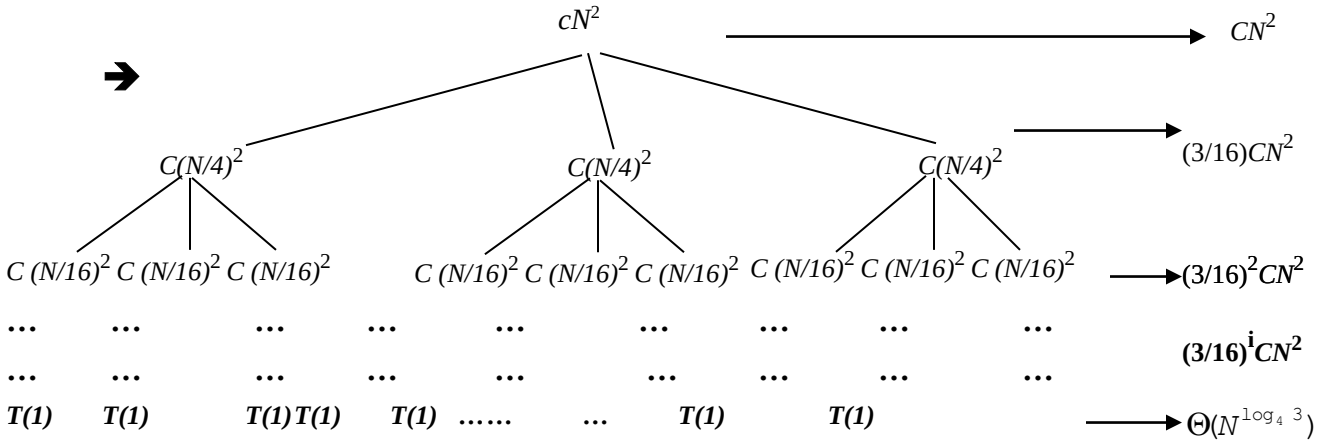
[since, T(N) expands as T(N) → T(N/4)+T(N/4)+T(N/4), as well as, contributes step count cN ^{2}].

➔



[since, each T(N/4) expands as T(N/4) → 3T(N/(4*4)) →3T(N/16) → T(N/16)+T(N/16)+T(N/16) as well as, contributes step count c(N/4) ^{2}].

➔



[for each branch **at level 3:** T(N/16*4)**= C* (N/16*4)^2 =C*N^2(1/ 16*16*4*4)= C*N^2(1/16)^3**]=> for all branches at level 3= > (3/16)^3CN^2

We observe,

Depth of Tree:-

The subproblem size for a node at depth i is $= (N/4)^i$.

Thus the subproblem size hits N=1 when $(N/4^i) = 1$ ➔ $i = log_4 N$

Therefore, No. of levels (Depth) of Tree = $log_4 N + 1$ *[i.e. 0, 1, 2, …, $log_4 N$]*

Step Count at last level:-

At the last level, each node contributes T(1) i.e. 1 step,

Since No. of nodes at level i is 3^i .

Hence, the last level at depth $log_4 N$ has a cost of $3^{log_4 N} = N^{log_4 3}$ [*ref. XII*]

Thence, summing up the cost of all the levels:

$$T(N) = cN^2 + (3/16)\, cN^2 + (3/16)^2\, cN^2 + \ldots + (3/16)^{log_4 N - 1}\, cN^2 + \Theta(N^{\log_4 3}).$$

$$= \sum_{i=0}^{\log_4 N - 1} \left(\frac{3}{16}\right)^i cN^2 + \Theta(N^{\log_4 3})$$

$$= \frac{\left(\frac{3}{16}\right)^{\log_4 N} - 1}{\left(\frac{3}{16}\right) - 1} . cN^2 + \Theta(N^{\log_4 3}).$$

This result is a bit messy. So, we can make an approximate about the order of growth of the function using an infinite decreasing geometric series as an upper bound.

i.e. $$T(N) = \sum_{i=0}^{\log_4 N - 1} \left(\frac{3}{16}\right)^i cN^2 + \Theta(N^{\log_4 3})$$

$$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cN^2 + \Theta(N^{\log_4 3})$$

$$= \frac{1}{1 - \left(\frac{3}{16}\right)} . cN^2 + \Theta(N^{\log_4 3}) \qquad [\textit{ ref. V }]$$

$$= \frac{16}{13} . cN^2 + \Theta(N^{\log_4 3}) = O(N^2) \qquad [\textit{ Justify!! }]$$

Thus the solution for the given recurrence using the recursion tree method is obtained to be

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

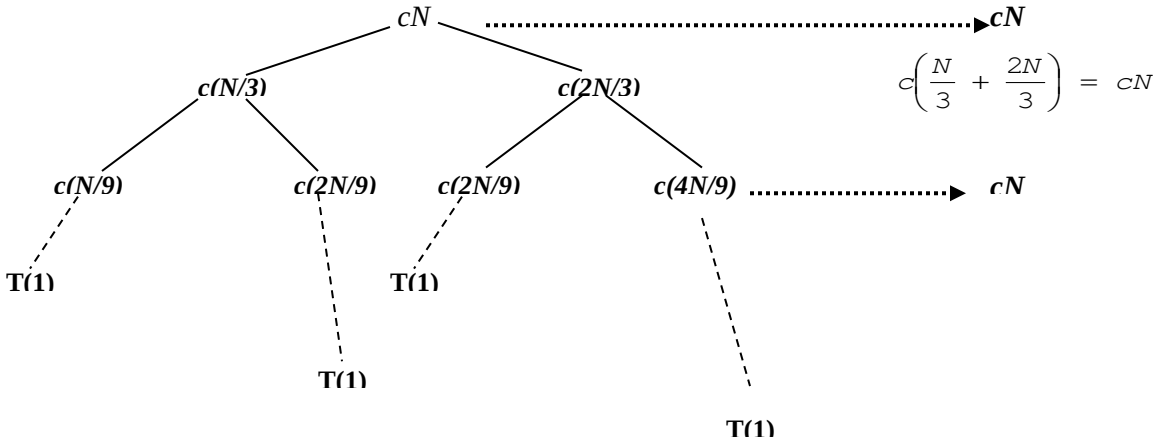
Solving Recurrences when the recursion tree is unbalanced

Such derivation is helpful when more than one recursive calls are involved, each of them working on unequal portions of input data (list) set.

e.g. Consider the recurrence

$T(N) = T(N/3) + T(2N/3) + O(N)$

The Recursion Tree



→Let the constant factor in *O(N)* be *c*.

→When we add the values across the levels of the recursion tree, we get the value of cost for each level = ***cN***.

→ ***Depth of Tree:-*** i.e. the number of levels in the tree.

The *longest path* from the root to *any leaf*, is across N -> (2/3)N -> $(2/3)^2$N -> … ->1.

Let the no. of steps in path (i.e. no. of levels in the tree) = k.

Then $\left(\frac{2}{3}\right)^k N = 1$ ➔ k = $log_{3/2}$N.

∴ The height of the tree = $log_{3/2}$N.

Thus for the recurrence yielding recursion tree of max level $log_{3/2}$N, we have to deal by doing an approximate estimation.

→ ***The Guess***

We expect the substitution to the recurrence to be at most the number of levels times the cost of each level.

Thus the solution for the given recurrence using the recursion tree method is obtained to be

T(N) = O(c N $log_{3/2}$N) = O(N logN)

4.3.2 Balancing

As we observe, the time complexity of a divide- and- conquer algorithm can be represented using a recursion tree. Further, an algorithm generating a balanced recursion tree at each level, achieves the best time complexity, of the order ***O***(NlogN), for example, as in Quicksort.

Further we observe in the BST, that a balanced BST obtains best search time of ***O***(logN) whereas an skewed BST consumes the worst search time of order ***O***(N). Applying this observation to achieve best efficiency in a tree based search algorithm, it is desirable to generate a tree which is balanced.

[Cormen pp 337] a class of balanced search trees called "AVL trees" in 1962. Balancing data structures can listed as below.

1. **"AVL trees"** – This tree was the first example of the idea of **balancing** a search tree, introduced by Adel'son-Vel'skiĭ and Landis in 1962.
2. **"2-3 trees"** – [J. E. Hopcroft 1970]. A 2-3-tree maintains balance by manipulating the degrees of nodes in the tree.
3. **"B-trees"** – [Bayer and McCreight 1972]. A generalization of 2-3 tree.
4. **"Red-black trees/ symmetric binary B-trees"** – [Bayer 1972]
5. **"Weight balanced trees"** – [Nievergelt and Reingold 19732],
6. **"k-neighbor trees"** – [Maurer 19].
7. **"Splay trees"** – [Sleator and Tarjan 1985], which are "self-adjusting" and maintain balance without any explicit balance condition such as color. Instead, "splay operations" (which involve rotations) are performed within the tree every time an access is made.

4.3.3 The Master Method

The Master method, in fact derived from the results of recursion tree approach to guess approximate solution for recurrences of a particular form, can be stated as below

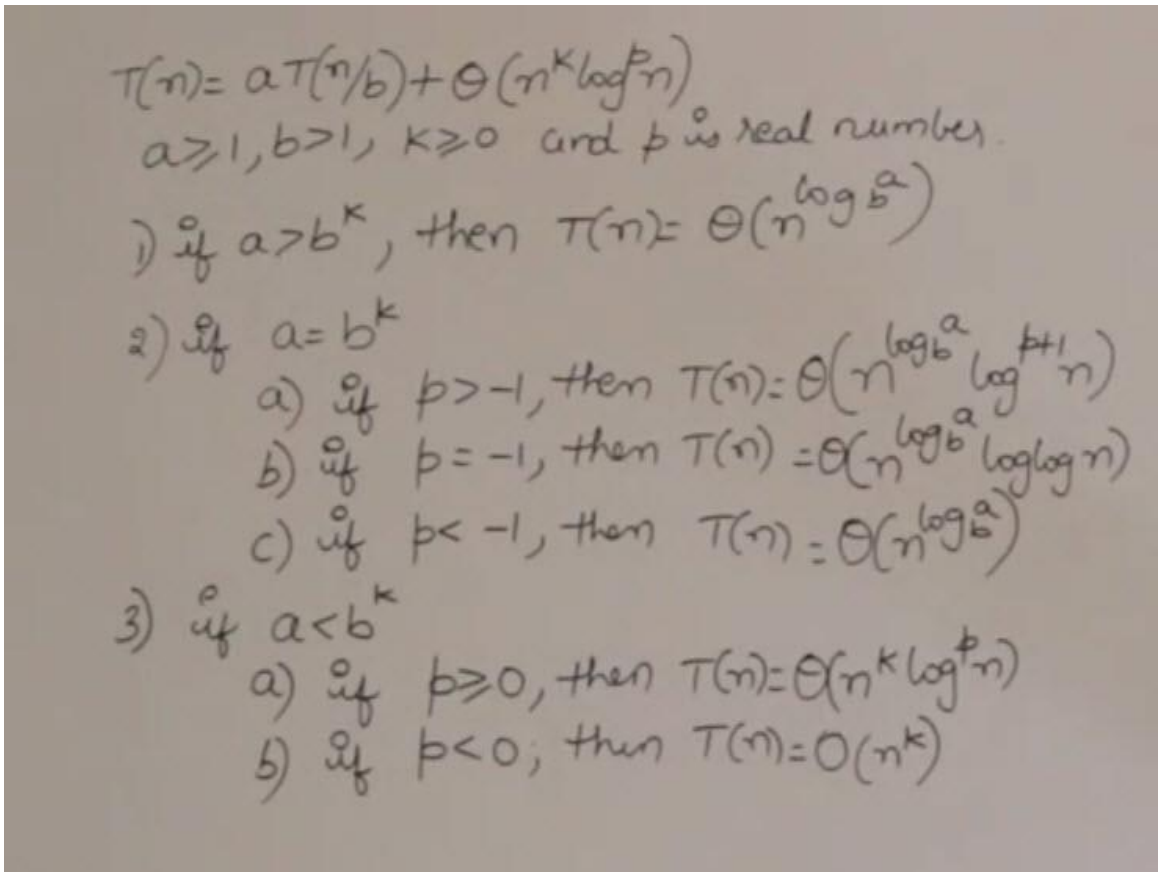
When $T(n) = aT\left(\frac{n}{b}\right) + f(n), a \geq 1, b > 1, being \quad positive-constants$, f(n) being asymptotically positive function.

Then,

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

II) $If \quad f(n) = \Theta(n^{\log_b a}) \quad then \quad T(n) = \Theta(n^{\log_b a} . \log_2 n)$

III $If \quad f(n) = \Omega(n^{\log_b a + \varepsilon}) \quad and \quad af\left(n/b\right) \leq cf(n), \quad \forall n > 0$ **, c being a positive integer,** $then \quad T(n) = \Theta(f(n))$.



Example 4.4.

Consider the recurrence $T(n) = 9T\left(\frac{n}{3}\right) + n^2$ $\frac{f(n)}{n^{\log_b a}} = \frac{n^2}{n^2} = 1 = n^0$

Here, we see that $f(n) = n^2$, and $n^2 = n^{\log_3 9} = n^{\log_b a}$ ➔ $f(n) = \Theta(n^{\log_b a})$

Hence rule II is applicable, So, $T(n) = \Theta(n^{\log_b a}.\log_2 n)$ ➔ $T(n) = \Theta(n^2.\log_2 n)$

Example 4.5.

Consider the recurrence $T(n) = 9T\left(\frac{n}{3}\right) + n$ $\frac{n}{n^2} = \frac{1}{n} = n^{-1}$

Here, we see that $f(n) = n$, and $n = n^{\log_3 9-1}$ ➔ $f(n) = O(n^{\log_b a-\varepsilon})$, $\varepsilon \leq 1$

Hence rule I is applicable, So, $T(n) = \Theta(n^{\log_b a})$ ➔ $T(n) = \Theta(n^2)$

Example 4.6.

Consider the recurrence $T(n) = 9T\left(\frac{n}{3}\right) + n^3$ $\frac{n^3}{n^2} = n$

Here, we see that $f(n) = n^3$, and $n^3 = n^{\log_3 9+1}$ ➔ $f(n) = \Omega(n^{\log_b a+\varepsilon})$, $\varepsilon \leq 1$

Moreover, $af\left(\frac{n}{b}\right)$ i.e. $9 * \left(\frac{n}{3}\right)^3 <= c * f(n)$ i.e. $c * n^3$, for $c = 1$, for all $n > 0$.

Hence rule III is applicable, So, $T(n) = \Theta(f(n))$ ➔ $T(n) = \Theta(n^3)$

Example 4.6.
T(n) = 4T(n/2) + nlogn
f(n) = nlogn =>k=1, p=1 $n^{\log_b a} = n^{\log_2 4} = n^2$ => $a>b^k$ (i.e. $4>2^1$) => Condition 1 applies
=> T(n) = $\theta(n^{\log_b a}) = \theta(n^{\log_2 4}) = \theta(n^2)$

Review Exercise:
1.Devise a binary search algorithm that splits the set not into two sets of (almost) equal sizes but into two sets, one of which is twice the size of the other. How does this algorithm compare with binary search?
2. Solve following recurrences using appropriate method:
i) T(n) = 2T(n/2) + n^3, T(1) =1 ii) T(n) = 8T(n/3)+n, T(1) = 0
iii) T(n) = 2T(n/4)+n, T(1) = 1 iv) T(n) = 4T(n/2) + nlogn
3. Solve following recurrences using any method, verify using recursion tree method.
i) T(n) = 2T(n/2)+n, T(1) =1 ii) T(n) = 2T(n/2)+n^2, T(1) =1

4.3.4 Justification of the Rules of Master Method

Reconsider the theorem. The three rules can be justified by intuition as below

RuleI.

$f(n) = 2f(n/2) + 1 = 4f(n/4) + 2 + 1 = \ldots$
$= n\,f(n/n) + n/2 + \ldots + 4 + 2 + 1$

The first term dominates

RuleII.

$f(n) = 2f(n/2) + n = 4f(n/4) + n + n = \ldots$
$= n\,f(n/n) + n + n + n + \ldots + n$

All terms are comparable

RuleIII.

$f(n) = f(n/2) + n = f(n/4) + n/2 + n = \ldots$
$= f(n/n) + 2 + 4 + \ldots + n/4 + n/2 + n$

The last term dominates

4.4 Some Properties of Asymptotic Notations

Little Oh, Big Oh, and Their Buddies

Notation		Growth rate	Example of use
$f(n) = o(g(n))$	$<$	strictly less than	$T(n) = cn^2 + o(n^2)$
$f(n) = O(g(n))$	$\leq$	no greater than	$T(n,m) = O(n \log n + m)$
$f(n) = \Theta(g(n))$	$=$	the same as	$T(n) = \Theta(n \log n)$
$f(n) = \Omega(g(n))$	$\geq$	no less than	$T(n) = \Omega(\sqrt{n})$
$f(n) = \omega(g(n))$	$>$	strictly greater than	$T(n) = \omega(\log n)$

Exercise 5.2-5 If $f(x) = x\sqrt{x+1}$, what can you say about the Big-Θ behavior of solutions to

$$T(n) = \begin{cases} 2T(\lceil n/3 \rceil) + f(n) & \text{if } n > 1 \\ d & \text{if } n = 1, \end{cases}$$

where n can be any positive integer, and the solutions to

$$T(n) = \begin{cases} 2T(n/3) + f(n) & \text{if } n > 1 \\ d & \text{if } n = 1, \end{cases}$$

where n is restricted to be a power of 3?

Since $f(x) = x\sqrt{x+1} \geq x\sqrt{x} = x^{3/2}$, we have that $x^{3/2} = O(f(x))$. Since

$$\sqrt{x+1} \leq \sqrt{x+x} = \sqrt{2x} = \sqrt{2}\sqrt{x}$$

for $x > 1$, we have $f(x) = x\sqrt{x+1} \leq \sqrt{2}x\sqrt{x} = \sqrt{2}x^{3/2} = O(x^{3/2})$. Thus the big-$\Theta$ behavior of the solutions to the two recurrences will be the same.

Exercise 5.2-6 What does the Master Theorem tell us about the solutions to the recurrence

$$T(n) = \begin{cases} 3T(n/2) + n\sqrt{n+1} & \text{if } n > 1 \\ 1 & \text{if } n = 1? \end{cases}$$

As we saw in our solution to Exercise 5.2-5 $x\sqrt{x+1} = \Theta(x^{3/2})$. Since $2^{3/2} = \sqrt{2^3} = \sqrt{8} < 3$, we have that $\log_2 3 > 3/2$. Then by conclusion 3 of version 2 of the Master Theorem, $T(n) = \Theta(n^{\log_2 3})$.

$$T(n) = 2T(\frac{n}{2}) + \frac{n}{\log n} = 4T(\frac{n}{4}) + \frac{n}{\log n} + \frac{n}{\log n/2} =$$

$$= \ldots = 2^k T\left(\frac{n}{2^k}\right) + n \cdot \sum_{i=0}^{k-1} \frac{1}{\log \frac{n}{2^i}} = 2^k T\left(\frac{n}{2^k}\right) + n \cdot \sum_{i=0}^{k-1} \frac{1}{\log n - i}$$

$$2^k T\left(\frac{n}{2^k}\right) + n \cdot \sum_{i=0}^{k-1} \frac{1}{\log n - i} = n + n \cdot \sum_{i=0}^{k-1} \frac{1}{k - i} = O(n \cdot \log\log n)$$

$T(n) = 4T(n/2) + n \lg n$

There are 4^i nodes at each level i, each with value $(n/2^i)\lg(n/2^i) = (n/2^i)(\lg n - i)$; again, the depth of the tree is at most $\lg n - 1$. We have the following summation:

$$T(n) = \sum_{i=0}^{\lg n - 1} n2^i(\lg n - i)$$

We can simplify this sum by substituting $j = \lg n - i$:

$$T(n) = \sum_{j=i}^{\lg n} n2^{\lg n - j} j = \sum_{j=i}^{\lg n} \frac{n^2 j}{2^j} = n^2 \sum_{j=i}^{\lg n} \frac{j}{2^j} = \mathbf{\Theta(n^2)}$$

$$T(n) = 3T(n/4) + n \lg n ,$$

we have $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = \Omega(n^{\log_4 3+\epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, we have that $af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n \lg n = cf(n)$ for $c = 3/4$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

The master method does not apply to the recurrence

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

even though it appears to have the proper form: $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n$. You might mistakenly think that case 3 should apply, since $f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$. The problem is that it is not *polynomially* larger. The ratio $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ is asymptotically less than n^{ϵ} for any positive constant ϵ. Consequently, the recurrence falls into the gap between case 2 and case 3. (See Exercise 4.6-2 for a solution.)

Extended Master Theorem

[Cormen exercise]

4.6-2 ⋆

Show that if $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \ge 0$, then the master recurrence has solution $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. For simplicity, confine your analysis to exact powers of b.

$T(n) = 2T(\frac{n}{2}) + \log n \Rightarrow \frac{f(n)}{n^{\log_b a}} = \frac{\log n}{n}$ not polynomially related

$\Rightarrow$ basic Master theorem not applicable

Conditional Asymptotic Notation

Many algorithms are easier to analyse if initially we only consider instances whose size satisfies a certain condition, such as being a power of 2. Conditional asymptotic notation handles this situation. [ref: Page 46 of Brassard and Bartley, Algorithmics]

*** Problem 2.1.20.** Let $b \geq 2$ be any integer, let $f : \mathbb{N} \to \mathbb{R}^*$ be a smooth function, and let $t : \mathbb{N} \to \mathbb{R}^*$ be an eventually nondecreasing function such that $t(n) \in X(f(n) \mid n$ is a power of $b)$, where X stands for one of O, Ω, or Θ. Prove that $t(n) \in X(f(n))$. Furthermore, if $t(n) \in \Theta(f(n))$, prove that $t(n)$ is also smooth. Give two specific examples to illustrate that the conditions "$t(n)$ is eventually nondecreasing" and "$f(bn) \in O(f(n))$" are both necessary to obtain these results. □

We illustrate this principle using an example suggested by the algorithm for merge sorting given in Section 4.4.

Example 2.1.2. Let $t(n)$ be defined by the following equation:

$$t(n) = \begin{cases} a & \text{if } n = 1 \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + bn & \text{otherwise,} \end{cases}$$

where a and b are arbitrary real positive constants. The presence of floors and ceilings makes this equation hard to analyse exactly. However, if we only consider the cases when n is a power of 2, the equation becomes

$$t(n) = \begin{cases} a & \text{if } n = 1 \\ 2t(n/2) + bn & \text{if } n > 1 \text{ is a power of 2.} \end{cases}$$

The techniques discussed in Section 2.3, in particular Problem 2.3.6, allow us to infer immediately that $t(n) \in \Theta(n \log n \mid n$ is a power of 2). In order to apply the result of the previous problem to conclude that $t(n) \in \Theta(n \log n)$, we need only show that $t(n)$ is an eventually nondecreasing function and that $n \log n$ is smooth.

The proof that $(\forall n \geq 1)[t(n) \leq t(n+1)]$ is by mathematical induction. First, note that $t(1) = a \leq 2(a+b) = t(2)$. Let n be greater than 1. By the induction hypothesis, assume that $(\forall m < n)[t(m) \leq t(m+1)]$. In particular, $t(\lfloor n/2 \rfloor) \leq t(\lfloor (n+1)/2 \rfloor)$ and $t(\lceil n/2 \rceil) \leq t(\lceil (n+1)/2 \rceil)$. Therefore

$$t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + bn \leq t(\lfloor (n+1)/2 \rfloor) + t(\lceil (n+1)/2 \rceil) + b(n+1) = t(n+1).$$

Asymptotic Notation with Several Parameters

It may happen when we analyze an algorithm that its execution time depends simultaneously on more than one parameter of the instance in question. This situation is typical of certain algorithms for problems involving graphs, where the time depends on both the number of vertices and the number of edges. In such cases the notion of the "size of the instance" that we have used so far may lose much of its meaning. For this reason, the asymptotic notation is generalized in a natural way to functions of several variables.

Let $f : \mathbb{N}\times\mathbb{N} \to \mathbb{R}^*$ be an arbitrary function. We define

$$O(f(m,n)) = \{\, t : \mathbb{N}\times\mathbb{N} \to \mathbb{R}^* \mid (\exists c \in \mathbb{R}^+)(\exists m_0, n_0 \in \mathbb{N}) \\ (\forall n \geq n_0)(\forall m \geq m_0)[t(m,n) \leq cf(m,n)] \,\}$$

There is nevertheless an essential difference between an asymptotic notation with only one parameter and one with several: it can happen that the thresholds m_0 and n_0 are indispensable. This is explained by the fact that while there are never more than a finite number of values of n_0 such that $n \geq$ no is not true, there are in general an infinite number of pairs < m , n > such that $m \geq 0$ and $n \geq 0$ yet such that $m \geq m_0$ and $n \geq n_0$ are not both true.

Time complexity of Graph Algorithms

Example Prim's Algorithm

```
MST-PRIM(G, w, r)                          Binary-Heap     Fib-Heap
1  for each u ∈ G.V                              V
2       u.key ← ∞                                V
3       u.π ← NIL                                V
4   r.key ← 0                                    V
5   Q ← G.V                                      1
6   while Q ≠ ∅                                  V
7       u ← EXTRACT-MIN(Q)                 O(Vlog V)
8       for each v ∈ G.Adj[u]                    2E
9           if v ∈ Q and w(u,v) < v.key           2E
10              v.π ← u                           2E
11              v.key ← w(u, v)             O(log V)
```

Operations on Asymptotic Notation

Let f and g be arbitrary functions from ***N*** into ***R****. Then

i. $\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n)) = \Theta(\max(f(n), g(n)))$
$= \max(\Theta(f(n)), \Theta(g(n)))$;

ii. $\Theta([f(n)]^2) = [\Theta(f(n))]^2 = \Theta(f(n)) \times \Theta(f(n))$;

iii. $[1 + \Theta(1)]^n = 2^{\Theta(n)}$, but $[\Theta(1)]^n \neq 2^{\Theta(n)}$; and

iv. $f(n) \in \prod_{i=0}^{n} \Theta(1)$

5. Best-case, Worst-case, Average Case Analysis

The actual performance of an algorithm largely depends on the data set instance. Depending on the data characteristics, *the best*, *the worst*, and an *average* performance of an algorithm can be determined. We can understand the notion going through following examples.

Example 5.1. Linear Search

```
#include <stdio.h>

// Linearly search x in arr[].  If x is present then return the index,

// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
        if (arr[i] == x)
         return i;
    }

    return -1;
}
```

Worst Case Performance

The worst case happens **when the element to be searched (x in the above code) is not present in the array**. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be Θ(n).

Best Case Performance

The best case occurs **when x is present at the first location**. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1).

Average Case Performance

In average case analysis, we take **all possible inputs and calculate computing time for all of the inputs**. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (**including the case of x not being present in array**). So we sum all the cases and divide the sum by (**n+1**). Hence

$$\text{Average Case Time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \frac{\theta((n+1)*(n+2)/2)}{(n+1)}$$

$$= \Theta(n)$$

Example 5.2. Insertion Sort

```
1.   for i = 0 to n
2.        key = A[i]
3.        j = i - 1
4.        while j >= 0 and A[j] > key
5.            A[j + 1] = A[j]
6.            j = j - 1
7.        end while
8.     A[j + 1] = key
9.     end for
```

COST OF LINE	C1	C2	C3	C4	C5	C6	C8
NO. OF TIMES IT IS RUN	n	n-1	n-1	$\sum_{j=1}^{n-1}(t_j)$	$\sum_{j=1}^{n-1}(t_{j-1})$	$\sum_{j=1}^{n-1}(t_{j-1})$	n-1

Total Running Time of Insertion sort (T(n)) = $C_1 * n + (C_2 + C_3) * (n - 1) + C_4 * \sum_{j=1}^{n-1}(t_j) + (C_5 + C_6) * \sum_{j=1}^{n-1}(t_j) + C_8 * (n - 1)$

WORST CASE TIME COMPLEXITY: Θ(N^2)
AVERAGE CASE TIME COMPLEXITY: Θ(N^2)
BEST CASE TIME COMPLEXITY: Θ(N)
SPACE COMPLEXITY: Θ(1)

Best Case Analysis [Θ(N)**] -** In Best Case i.e., ***when the array is already sorted***, **tj = 1**
Therefore, $T(n) = C1 * n + (C2 + C3) * (n - 1) + C4 * (n - 1) + (C5 + C6) * (n - 2) + C8 * (n - 1)$
which when further simplified has dominating factor of **n** and gives T(n) = C * (n) or O(n)

Worst Case Analysis [Θ(N^2)] - In Worst Case i.e., ***when the array is reversly sorted*** (in descending order), **tj = j**
Therefore, T(n) = C1 * n + (C2 + C3) * (n - 1) + C4 * (n - 1) (n) / 2 + (C5 + C6) * ((n - 1) (n) / 2 - 1) + C8 * (n - 1)

Average Case Analysis [Θ(N^2)] - Let's ***assume that tj = (j-1)/2*** to calculate the average case
Therefore, T(n) = C1 * n + (C2 + C3) * (n - 1) + C4/2 * (n - 1) (n) / 2 + (C5 + C6)/2 * ((n - 1) (n) / 2 - 1) + C8 * (n - 1)
which when further simplified has dominating factor of n^2 and gives T(n) = C*(n^2) or O(n^2)

[Cormen pg 28] Suppose that we randomly choose n numbers and apply insertion sort.
How long does it take to determine where in subarray A[1, .., j-1] to insert element A[j]? => On average, half the elements in A[1, .., j-1] are less than A[j], and half the elements are greater.
[e.g. for [1,2,3,4]: <below:-[0,1,2,3]-> avg=6/4~2(which is n/2 for n=4)
and >above:-[3,2,1,0]]-> avg=6/4~2(which is n/2 for n=4)

On average, therefore, we check half of the subarray A[1, .., j-1], and so tj is about j=2. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

Can we optimize it further?
Searching for the correct position of an element and **Swapping** are two main operations included in the Algorithm.
We can optimize the searching by using Binary Search, which will improve the searching complexity from O(n) to O(log n) for one element and to n * O(log n) or O(n log n) for n elements.
But since it will take O(n) for one element to be placed at its correct position, n elements will take n * O(n) or O(n2) time for being placed at their right places. Hence, the overall complexity remains **O(n2)**.
We can optimize the swapping by using Doubly Linked list instead of array, that will improve the complexity of swapping from O(n) to O(1) as we can insert an element in a linked list by changing pointers (without shifting the rest of elements).
But since the complexity to search remains O(n2) as we cannot use binary search in linked list. Hence, The overall complexity remains **O(n2)**.
Therefore, we can conclude that we cannot reduce the worst case time complexity of insertion sort from **O(n2)** .

Example 5.3. Quick Sort

The running time of the *QuickSort* algorithm is equal to the running time of the two recursive calls plus the linear time spent in the *Partition* step; the pivot selection takes only constant time.

Typical Quick_Sort Algorithm

```
Q_Sort(A[],l, u)
      1.    if (l<=u) return;
      2.    p = select_pivot_element();
      3.    m = Partition (A, l, m, u) ;//determines correct position
                                 //for p in the list.
      4.    if (l<m-1) Q_Sort(A, l, m-1);
      5.    if (m+1<u) Q_Sort(A, m+1, u);
```

This gives the basic *QuickSort* recurrence as

$T(N) = T(i) + T(N - i - 1) + cN$; Take $T(0) = T(1) = 1$. Where,

$i = |S_1|$ is the number of elements in S_1, the first partition. -- [5.1.a]

Worst Case Performance

When the pivot is the smallest element while sorting in ascending order (typically when the input list is already sorted in desired order), all the time, then, $i = 0$.

Moreover, if we ignore T(0) = 1, which is insignificant with respect to large N, then, the resulting recurrence becomes

$T(N) = T(N-1) + cN, \quad N>1$ -- [5.1.b]

Solving this recurrence, we get

$T(N) = T(1) + c.\sum_{i=2} i = \quad O(N^2)$ -- [5.1.c]

Hence, for *QuickSort* the ***Worst Case Time Complexity = O(N ^{2}).***

Best Case Performance

Best case of *QuickSort* occurs in ideal sequence of input data elements, when pivot is always (in each partition) found in the middle.

Then, $T(N) = 2T(N/2) + cN, \quad N>1$ -- [5.1.d]

Solving this recurrence, we get the solution as:

$T(N) = O(NlgN)$ -- [5.1.e]

Hence, for *QuickSort* the ***Best Case Time Complexity = O(NlgN).***

Average Case Performance

For, average case, we assume that each of the sizes for S_1 is equally likely, and hence the probability for each size is $1/N$. Then we can analyze as following for all partitioning strategies which preserve randomness.

As per the assumptions above,

The average value of T(i), and hence of T(N – i – 1), is $\left(1/N\right)\sum_{j=0}^{N-1} T(j)$. -- [5.1.f]

Then equation 5.1.a becomes

$$T(N) = \left(2/N\right)\sum_{j=0}^{N-1} T(j) + cN \qquad \text{-- [5.1.g]}$$

[$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-1-i) + cn)$

$= \frac{2}{n}(T(0) + T(1) + \ldots + T(n-2) + T(n-1)) + cn$]

Solve *eqn. 5.1.g* :

$$N.T(N) = 2.\sum_{j=0}^{N-1} T(j) + cN^2 \qquad \text{[Multiply both sides by } N\text{]} \qquad \text{-- [5.1.h]}$$

➔ $$(N\text{-}1).T(N\text{-}1) = 2.\sum_{j=0}^{N-2} T(j) + c(N-1)^2 \qquad \text{-- [5.1.i]}$$

If we subtract eqn. 5.1.i from eqn 5.1.h, we get,

$$N.T(N) - (N\text{-}1).T(N\text{-}1) = 2T(N\text{-}1) + 2cN - c \qquad \textit{[Applying telescopic sums]}$$

We rearrange the terms and drop the insignificant $-c$ on the right.

$$NT(N) = (N+1)T(N\text{-}1)+2cN \qquad \text{-- [5.1.j]}$$

Solve *eqn. 5.1.j* :

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2c}{N+1} \qquad \text{-- [5.1.k]}$$

Similarly,

$$\frac{T(N-1)}{N} = \frac{T(N)}{N-1} + \frac{2c}{N} \qquad \text{-- [5.1.l]}$$

$$\frac{T(N-2)}{N-1} = \frac{T(N-1)}{N-2} + \frac{2c}{N-1} \qquad \text{-- [5.1.m]}$$

…

(N-2)th steps $\frac{T(1)}{2} = \frac{T(0)}{1} + \frac{2c}{2}$ -- [5.1.n]

Adding eqns. 5.1.k to 5.1.n, we get

$$\frac{T(N)}{N+1} = \frac{T(0)}{1} + 2\sum_{i=2}^{N+1}\frac{1}{i}.c \qquad \text{-- [5.1.o]}$$

$$\frac{T(N)}{N+1} = O(\log N)$$ [since the sum on the right is approximately

$log_e(N+1)+\gamma$ - 3/2 ; $\gamma \approx 0.577$ *is Euler Constant*.]

Thus, for *QuickSort* the ***Average Case Time Complexity = O(NlgN).***

Partitioning Algorithms for QuickSort

1. Hoare Partition- Uses two indices (usual partition scheme) which start at the two ends and move in opposite directions to find the correct location of the pivot. It does three times lesser swaps than Lomuto's partition

2. Lomuto's partition scheme is easy to implement as compared to Hoare scheme. This has inferior performance to Hoare's QuickSort.

Lomuto's Partition Scheme:

```
partition(arr[], lo, hi)
    pivot = arr[hi]
    i = lo     // place for swapping
    for j := lo to hi – 1 do
        if arr[j] <= pivot then
            swap arr[i] with arr[j]
            i = i + 1
    swap arr[i] with arr[hi]
    return i
```

Example 5.2. Merge_Sort

In a typical Merge_Sort algorithm, [*ref. example 3.2*] in each pass (or in each recursive call in a recursive version) regardless of what the data is, the list is divided in two equal parts, and each portion is processed in similar fashion. Moreover, in each call of the Merge_Sort, after completion of two calls over two (N/2)-length portions, a Merge routine processes the whole

length (i.e. N) to combine the results. So, in all cases, whatever the data characteristic is, the recurrence (we consider for the recursive version), is the same, i.e.,

$T(N) = 2T(N/2)+N$ -- [5.2.a]

We, know the above recurrence has the solution

$T(N) = O(NlogN)$

Thus, for *MergeSort* the ***Best, Worst, Average Case Time Complexity = O(NlgN).***

6. Improving Time complexity: Some Algorithm Design Examples

→***Sometimes applying some reasoning helps:***

Example 6.1. Computing X^n

```
Comp_X_n_ Algo_I (X,n)
      1.    power := x
      2.    for i := 1 to n-1 do
      3.          power := power * X;
```

time complexity - Θ(n).

```
Comp_X_n_ Algo_II (X,n)  // applicable particularly if  n = 2^k, k being
                         // some positive integer+r.
      1.    power := x
      2.    for i := 1 to log2n do
      3.          power := power * power;
```

time complexity - Θ(log$_2$n).

```
Comp_X_n_ Algo_III (X,n)  // applicable even if  n != 2^k, k being some
                          // positive integer.
      1.    power := 1, z:=X              n=13
      2.    for m := n to 1 do
      3.          while ((m mod 2) == 0) do
      4.                m:= floor(m/2), z:= z*z; //X.X.X.X.X.X.X.X
      5.          m:= m-1
      6.          power := power * z; //X.X.X.X.X.X.X.X.X.X.X.X.X
      7.    return power
```

time complexity - Θ(log$_2$n).

→***Sometimes divide and conquer technique helps:***

Example 6.2. Processing a bit stream

"Given a bit stream such that all 1's precede any 0, we are required to find the position of the last 1".

```
Bit_Proc1_Algo_I (A[], n) //n is the length of the string.
      //The last one can be found simply by performing a sequential
      //search. [try!! –write algo]
```

time complexity - Θ(n).

```
Bit_Proc1_Algo_II (A[], l, u) // u-l+1 = n
      1.    if (l == u)
      2.          if (A[l] == 1) return l
      3.          else          return -1 // -1 indicates "1 not found"
      4.    m:= (l+u)/2
      5.    if (A[m]==1)
      6.          if (A[m+1]==0)return m else Bit_Proc1_Algo_I(A, m+1, u)
      7.    else  // i.e. if (A[m]==0)
      8.          if (A[m-1]==1)return m else Bit_Proc1_Algo_I(A, l, m-1)
```

time complexity - $\Theta(\log_2 n)$. [*try!! –verify*]

Review Exercise:

1. Suppose that each row of an n x n array A consists of 1's and 0's such that, in any row I of A, all the 1's come before any 0's in that row. Suppose further that the number of 1's in row I is at least the number in row i+1, for i= 0,1, …,n-2. Assuming A is already in memory, describe a method running in O(nlogn) time (not O(n^2) time) for finding the row of A that contains the most 1's. [1111000000]

2. Write a recursive algorithm for finding both the maximum and the minimum elements in an array A of n elements. Determine its time complexity. Try to revise the algorithm to improve time complexity.

3. Suppose that each node in a BST also has the field leftsize. Design an algorithm to insert an element x into such a BST. The complexity of your algorithm should be O(h), where h is the height of the search tree. Verify it.

4. An array A contains n-1 unique integers in the range [0, n-1], that is there is one number from this range that is not in A. Design an O(n)-time algorithm for finding that number. You are allowed to use only O(1) additional space besides the array itself.

5. Let x[1:n] and y[1:n] contain two sets of integers, each sorted in nondecreasing order. Write an algorithm that finds the median of te 2n combined elements. What is the time complexity of your algorithm?

6. The k-th quantiles of an n-element set are the k-1 elements from the set that divide the sorted set into k equal-sized sets. Give an O(n logk) time algorithm to list the kth quantiles of a set.

7. Given two vectors X=(x1,…, xn) and Y=(y1,…, yn), X<Y if there exists an i, 1<=i<=n, such that xj = yj for 1<=j<I and xi<yi. Given m vectors each of size n, write an algorithm that determines the minimum vector. Analyze time complexity

7. Correctness Proof

I. Insertion Sort

The algorithm

```
for j <- 2 to length(A) do
// Invariant 1:  A[1..j-1] is a sorted permutation of the original A[1..j-1]
   key <- A[j]
   i <- j-1
   while (i > 0 and A[i] > key) do
// Invariant 2:  A[i .. j] are each >= key
       A[i+1] <- A[i]
       i <- i -1
   A[i+1] <- key
```

Correctness proof

We will prove the correctness of this sorting algorithm by proving that the loop invariants hold and then drawing conclusions from what this implies upon termination of the loops.

Invariant 1:

- ⇨ Invariant 1 is true initially because in the first iteration of the loop j = 2 so A[1..j-1] is A[1..1] and a single element is always a sorted list.
- ⇨ (Invariant 1 is maintained by the loop and true during the next iteration, we must examine the body of the loop) - It must be true that after the last line of the outer loop (A[i+1] <- key), A[1..j] is a sorted permutation of the original A[1..j]. **We will show this is true by examining Invariant 2** and reasoning from it.
    - **Invariant 2** is true upon initialization (the first iteration of the inner loop) because i = j-1, A[i] was explicitly tested and know to be > key, and A[j] == key.
    - **The inner loop maintains this invariant** because the statement A[i+1] <- A[i] moves a value in A[i], known to be > key, into A[i+1] which also held a value >= key. Thus this statement does not change the validity of the invariant. (If after setting i <- i-1 the invariant does not hold, the loop test of A[i] > key catches the fact and terminates the loop.)
    - We also note that the inner loop does not destroy any data because the first iteration copies a value over A[j], the value stored in key. As long as key is stored back into the

array, we maintain the statement that A[1..j] contains the first j elements of the original list.

- Upon termination of the inner loop, we know the following things about the array A:
  - A[1..i] are sorted and <= key (true by default if i == 0, true because A[1..i] is sorted and A[i] <= key if i > 0)
  - A[i+1 .. j] are sorted and >= key because the loop invariant held before i was decremented and the invariant said A[i .. j] >= key.
  - A[i+1]==A[i+2] if the loop executed at least once and A[i+1] == key if the loop did not execute at all.
- Given these facts, we see that A[i+1] <- key does not destroy any data and gives us A[1..j] is a sorted permutation of the original j elements of A.

Thus, Invariant 1 is maintained after an iteration of the loop and it remains to note that when the outer loop terminates, j = length(A) + 1 so A[1..j-1] is A[1..length], thus the entire array is sorted.

Amortized Analysis

[ref Cormen p-451]
In an ***amortized analysis***, we average the time required to perform a sequence of data-structure operations over all the operations performed. With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the *average performance of each operation in the worst case*.

Comparison to other analysis techniques

~Worst-case: As mentioned above, worst case analysis can give overly pessimistic bounds for sequences of operations, because such analysis ignores interactions among different operations on the same data structure (Tarjan 1985). Amortized analysis may lead to a more realistic worst case bound by taking these interactions into account. Note that the bound offered by amortized analysis is, in fact, a worstcase bound on the average time per operation; a single operation in a sequence may have cost worse than this bound, but the average cost over all operations in any valid sequence will always perform within the bound.

~ Average case: Amortized analysis is similar to average case analysis, in that it is concerned with the cost averaged over a sequence of operations. However, average case analysis relies on probabilistic assumptions about the data structures and operations in order to compute an expected running time of an algorithm. Its applicability is therefore dependent on certain assumptions about probability distributions on algorithm inputs, which means the analysis is invalid if these assumptions do not hold (or that probabilistic analysis cannot be used at all, if input distributions cannot be described!) (Cormen et al. 2001, 92–3). Amortized analysis needs no such assumptions. Also, it offers an upperbound on the worst case running time of a sequence of operations, and this bound will always hold. An average case bound, on the other hand, does not preclude the possibility

that one will get "unlucky" and encounter an input that requires much more than the expected computation time, even if the assumptions on the distribution of inputs are valid. These differences between probabilistic and amortized analysis therefore have important consequences for the interpretation and relevance of the resulting bounds.

~Competitive analysis: Amortized analysis is closely related to competitive analysis, which involves comparing the worstcase performance of an online algorithm to the performance of an optimal offline algorithm on the same data. Amortization is useful because competitive analysis's performance bounds must hold regardless of the particular input, which by definition is seen by the online algorithm in sequence rather than at the beginning of processing. Sleator and Tarjan (1985a) offer an example of using amortized analysis to perform competitive analysis.

The three approaches to amortized analysis

There exist three main approaches to amortized analysis: aggregate analysis, the accounting method, and the potential method (Cormen et al. 2001, p.405).

Aggregate analysis- in which we determine an upper bound $T(n)$ on the total cost of a sequence of n operations. The average cost per operation is then $T(n)/n$. We take the average cost as the amortized cost of each operation, so that all operations have the same amortized cost.

Accounting method- in which we determine an amortized cost of each operation. When there is more than one type of operation, each type of operation may have a different amortized cost. The accounting method overcharges some operations early in the sequence, storing the overcharge as "prepaid credit" on specific objects in the data structure. Later in the sequence, the credit pays for operations that are charged less than they actually cost.

Potential method- is like the accounting method in that we determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method maintains the credit as the "potential energy" of the data structure as a whole instead of associating the credit with individual objects within the data structure.

Examples
Aggregate Analysis
Consider following Stack operation

MULTIPOP(S, k)

1 **while** not STACK-EMPTY(S) and k > 0

2 POP(S)

3 k = k- 1

The worst-case cost of a MULTIPOP operation in the sequence is $O(n)$, since the stack size is at most n. The worst-case time of any stack operation is therefore $O(n)$, and hence a sequence of n operations costs $O(n^2)$, since we may have $O(n)$ MULTIPOP operations costing $O(n)$ each. Although this analysis is correct, the $O(n^2)$ result, which we obtained by considering the worst-case cost of each operation individually, is not tight.

Using aggregate analysis, we can obtain a better upper bound that considers the entire sequence of n operations. In fact, although a single MULTIPOP operation can be expensive, any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack can cost at most $O(n)$.

Reason ➔ We can pop each object from the stack at most once for each time we have pushed it onto the stack.

Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP
= at most the number of PUSH operations = at most n.

For any value of n, any sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time.

Therefore, the average cost of an operation is $O(n)/n = O(1)$.

8. Probabilistic Analysis and Randomized Algorithms

Probabilistic analysis is the use of probability in the analysis of problems. As such, this approach helps in assessing average performance of an algorithm. To do so, we need to know, or make assumptions about, the characteristic distribution of inputs. Then we analyze our algorithm, computing an expected running time. The expectation is taken over the distribution of the possible inputs. Thus, in effect, we estimate the *average-case behavior* of the algorithm, i.e. average running time over all possible inputs. Thus we may use probabilistic analysis as a technique in designing an efficient algorithm and as a means to gain an insight into a problem.

However, for some problems, we can not describe a reasonable input distribution, and in these cases we cannot use probabilistic analysis. In fact, here, we know very little about the distribution or we may not be able to model the distribution information computationally. Even in such situations, we can use probability and randomness as a tool for algorithm design and analysis, by implementing part of the algorithm using randomization. In these situations probabilistic analysis guides in the development of such ***randomized algorithms***.

A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits.

More generally, for a randomized algorithm, the performance behavior is determined not only by its input but also by values produced by a *random-number generator*.

Randomized algorithms are categorized into two types:

➔ **Las Vegas**: algorithms that use the random input so that they always terminate with the correct answer (**always the same output for the same input**), but where the expected running time is finite, e.g. QuickSort

➔ **Monte Carlo**: algorithms which have a chance of producing an incorrect result or fail to produce a result either by signaling a failure or failing to terminate, e.g. Primarity Testing

Randomized Algorithms and their Performance Estimation with Probabilistic Analysis

Example 8.1. Randomized QuickSort

```
RANDOM_QUICKSORT (A, lb, ub)
         3. if lb < ub
         4. then q<- RANDOM_PARTITION(A,lb,ub)
         5.       RANDOM_QUICKSORT (A, lb, q-1)
         6.       RANDOM_QUICKSORT (A, q+1, ub)
RANDOM_PARTITION (A, lb , ub)
         1. i <- Random (lb, ub)
         2. exchange A[ub], A[i]
         3. return PARTITION(A, lb, ub)
PARTITION (A, p,r)
   1.  x <- A[r]
   2.  I <- p-1
   3.  for j <- p to r-1
   4.       do if A[j] <= x
   5.          then  I <- i+1
   6.                exchange A[i], A[j]
   7.  exchange A[i+1], A[r]
   8.  return i+1
```

Expected Running Time

The running time of Quicksort is dominated by the time spent in the PARTITION. Each time PARTITION is called, a pivot element is selected and this element is never included in any future recursive calls to RANDOM_QUICKSORT and PARTITION. Thus there can be at most *n* calls to PARTITION over the entire execution of the quicksort algorithm. One call to PARTITION takes O(1) time plus some time to execute the **for** loop (line 3-6). Each iteration of this **for** loop performs a comparison in line 4.

Let X be the number of comparisons performed in line 4 of PARTITION over the entire execution of the algorithm on an n-element array. Then the running time of the sorting algorithm as whole would be $O(n + X)$.

So we need to compute *X*, the total number of comparisons performed in all calls to PARTITION. In fact, we derive an ***overall bound on the total number of comparisons***.

Let us rename the elements of A as $z_1, z_2, \ldots, z_n$, z_i being the i^{th} smallest element.

Let $Z_{ij} = \{ z_i, z_{i+1}, \ldots, z_j \}$ to be the set of elements between z_i and z_j inclusive.

Let X_{ij} = I { z_i is compared to z_j }, where we consider any occurrence of comparison during the execution of the algorithm, not just during one iteration or one call of PARTITION.

Since each element is compared at most once, we have:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

Taking expectations on both sides, and then using linearity of expectations and using the lemma

$X_A = I\{A\}$ ➔ $E[X_A] = Pr[A]$.

We get,i

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}. \qquad \text{-- eq. [A]}$$

Now,

$$\Pr\{z_i \text{ is compared to } z_j\} =$$

$$\Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\}$$

$$= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\}$$

$$+ \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\}$$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$ [since the two events are mutually exclusive.] --eq. [B]

Combining equations [A] and [B], we get,

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}.$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

$$= \sum_{i=1}^{n-1} O(\lg n)$$

$$= O(n \lg n).$$

Thus we conclude that using RANDOM_PARTITION, the expected running time of quicksort is $O(n \lg n)$.

Average case performance versus estimated time of randomized algorithm

The comparison between $O(n \lg n)$ as average case run-time of quicksort and $O(n \lg n)$ as expected run time of randomized quicksort is made clear by considering the fact that the average case performance estimation hinges on the assumption that all the input permutations are equally likely. However, we may ever encounter the worst case estimated $O(n^2)$ performance across several runs of ordinary quicksort.

Problem Solving with Randomized algorithms and probabilistic analysis

{ref: Ex.5-1 and 5-2 of Cormen --determining *expected time*.}

Example 8.3. Search in an unordered list.

```
//RANDOMIZED SEARCH ALGORITHM: pseudocode
R_SEARCH (A, l, u, val, k)
   1. count = 0
   2. while (count<u-l+1) do step 3-7
   3. i = random() % (u-l)+u
   4. if (A[i] == val) then do step 5,6
   5.       print i
   6.       exit
   7. count = count + 1
   8. print "value not found"
   9. end
```

Expected number of comparisons to find given value

I. If there is only single occurrence of *val*, i.e. k=1.

Here,

$$E[X_i] = \Pr[(Ar[i] \neq val)]$$
$$= 1 - \Pr[(Ar[i] = val)]$$

Since single value of *val* is present in A, so all the indices are equally likely to have it. So,

$$\Pr[(Ar[i] = val)] = \frac{1}{n}, \quad \text{and,}$$

$$E[X_i] = 1 - \frac{1}{n} = \frac{n-1}{n}, \text{ where, } n = u\text{-}l+1 = \text{size of A.}$$

Hence,

$$E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i]$$

$$= \sum_{i=1}^{n} \frac{n-1}{n} = \frac{n(n-1)}{n} = n - 1$$

II. If there are more than single occurrences of *val*, i.e. $k \geq 1$.

Generalizing for $k \geq 1$,

$$E[X_i] = \Pr[(Ar[i] \neq val)] = \frac{n-k}{n}$$

Therefore,

$$E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{n-k}{n} = \frac{n(n-k)}{n} = n - k$$

III. If there is no occurrence of *val*, i.e. k =0.

$$E[X_i] = \Pr[(Ar[i] \neq val)] = \frac{n}{n} = 1$$

So, $E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} 1 = n$

10. Approximation Algorithms

[ref: Horowitz & Sahni ch12. Brassard & Bratley]

No **NP-hard** problem can be solved in polynomial time. Yet, many NP-hard optimization problems have great practical importance and it is desirable to solve large instances of these problems in a reasonable amount of time.

The best known algorithms to solve NP-Hard problems have a worst-case time complexity which is **exponential** in input.

However some improvisations may provide algorithms for such problems in **subexponential time complexity e.g. $2^{n/2}$, $n^{\log n}$** etc.

Redefining "Solve" to achieve subexponential solutions-

➔ An **approximation algorithm** is an algorithm that generates an approximate solution to P.

An approximate solution to a given problem is a feasible solution with return value close to an optimal solution.

➔ **Probabilistically Good algorithm**- An approximation for problem P that almost always generates optimal solution.

Let *C* be the cost of a solution found for a problem of size *n* and *C** be the optimal solution for that problem.

Then we say an algorithm has an **approximation ratio of ρ(n)** *(that's "rho")* if

C/*C** ≤ ρ(n) for minimization problems: the factor by which the actual solution obtained is larger than the optimal solution.
*C**/*C* ≤ ρ(n) for maximization problems: the factor by which the optimal solution is larger than the solution obtained

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n)$$

The CLRS text says both of these at once in one expression shown to the right. The ratio is never less than 1 (perfect performance).

An algorithm that has an approximation ratio of ρ(n) is called a **ρ(*n*)-approximation algorithm**.

An **approximation scheme** is a parameterized approximation algorithm that takes an additional input ε > 0 and for any fixed ε is a (1+ε)-approximation algorithm.

An approximation scheme is a **polynomial approximation scheme** if for any fixed ε > 0 the scheme runs in time polynomial in input size *n*. (We will not be discussing approximation schemes today; just wanted you to be aware of the idea. See section 35.5)

A common strategy in approximation proofs: we don't know the size of the optimal solution, but we can set a lower bound on the optimal solution and relate the obtained solution to this lower bound.

Traveling Salesperson Problem- Approximation Algorithm

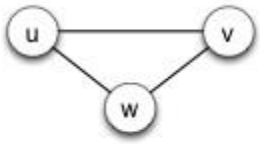
In the **Traveling Salesperson Problem** (TSP) we are given a complete undirected graph $G = (V, E)$ (representing, for example, routes between cities) that has a nonnegative integer cost $c(u, v)$ for each edge $\{u, v\}$ (representing distances between cities), and must find a Hamiltonian cycle or tour with minimum cost. We define the cost of such a cycle A to be the sum of the costs of edges:

$$c(A) = \sum_{(u,v)\in A} c(u, v)$$

The unrestricted TSP is very hard, so we'll start by looking at a common restriction.

Triangle Inequality TSP

In many applications (e.g., Euclidean distances on two dimensional surfaces), the TSP cost function satisfies the **triangle inequality**:



$c(u, v) \le c(u, w) + c(w, v), \forall\ u, v, w \in V.$

Essentially this means that it is no more costly to go directly from u to v than it would be to go between them via a third point w.

Approximate Tour for Triangle Inequality TSP

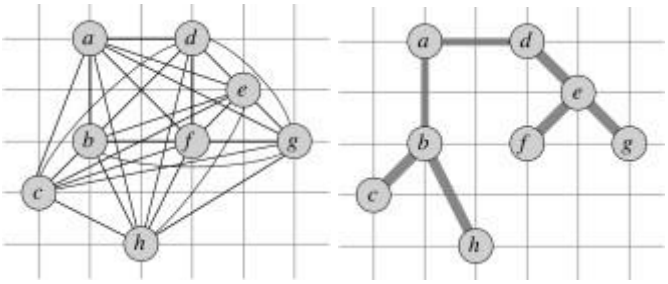
The triangle inequality TSP is still NP-Complete, but there is a 2-approximation algorithm for it. The algorithm finds a minimum spanning tree (Topic 17), and then converts this to a low cost tour:

```
APPROX-TSP-TOUR(G, c)
1  select a vertex r ∈ G.V to be a "root" vertex
2  compute a minimum spanning tree T for G from root r
       using MST-PRIM(G, c, r)
3  let H be a list of vertices, ordered according to when they
       are first visited in a preorder tree walk of T
4  return the hamiltonian cycle H
```

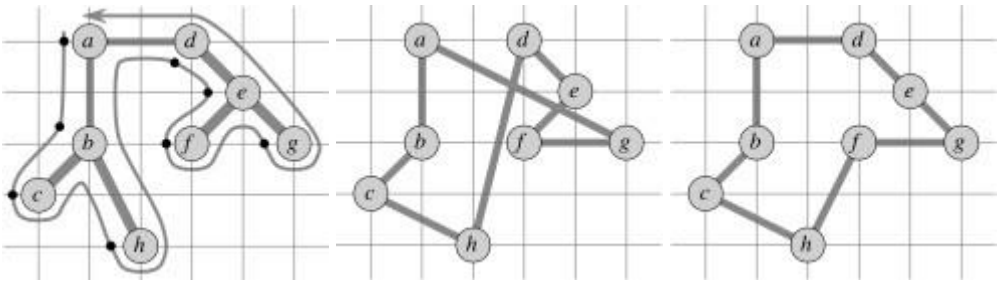
(Another MST algorithm might also work.)

Example

Suppose we are working on the graph shown below to the left. (Vertices are placed on a grid so you can compute distances if you wish.) The MST starting with vertex *a* is shown to the right.



Recall from early in the semester (or ICS 241) that a preorder walk of a tree visits a vertex before visiting its children. Starting with vertex *a*, the preorder walk visits vertices in order *a*, *b*, *c*, *h*, *d*, *e*, *f*, *g*. This is the basis for constructing the cycle in the center (cost 19.074). The optimal solution is shown to the right (cost 14.715).



Analysis of Approx-TSP-Tour

Theorem: **Approx-TSP-Tour is a polynomial time 2-approximation algorithm for TSP with triangle inequality.**

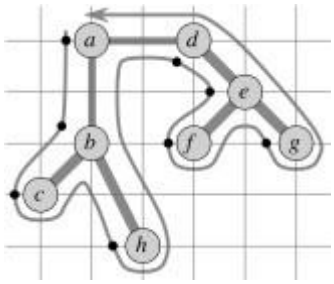
APPROX-TSP-TOUR(G, c)
1 select a vertex $r \in G.V$ to be a "root" vertex
2 compute a minimum spanning tree T for G from root r
using MST-PRIM(G, c, r)
3 let H be a list of vertices, ordered according to when they
are first visited in a preorder tree walk of T
4 **return** the hamiltonian cycle H

Proof: The algorithm is correct because it produces a Hamiltonian circuit.

The algorithm is polynomial time because the most expensive operation is `MST- Prim`, which can be computed in O(*E* lg *V*) (see Topic 17 notes).

For the approximation result, let *T* be the spanning tree found in line 2, *H* be the tour found and *H** be an optimal tour for a given problem.

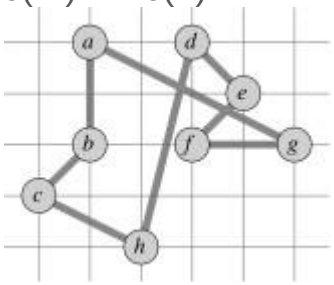
If we delete any edge from *H, we get a spanning tree that can be no cheaper than the _minimum_ spanning tree _T_, because _H_* has one more (nonegative cost) edge than *T*:



$c(T) \leq c(H^*)$

Consider the cost of the **full walk** W that traverses the edges of T exactly twice starting at the root. (For our example, W is ⟨{a, b}, {b, c}, {c, b}, {b, h}, {h, b}, {b, a}, {a, d}, … {d, a}⟩.) Since each edge in T is traversed twice in W:

$c(W) = 2\ c(T)$



This walk W is not a tour because it visits some vertices more than once, but we can skip the redundant visits to vertices once we have visited them, producing the same tour H as in line 3. (For example, instead of ⟨{a, b}, {b, c}, {c, b}, {b, h}, … ⟩, go direct: ⟨{a, b}, {b, c}, {c, h}, … ⟩.)

By the triangle inequality, which says it can't cost any more to go direct between two vertices,

$c(H) \leq c(W)$

Noting that H is the tour constructed by Approx-TSP-Tour, and putting all of these together:

$c(H) \leq c(W) = 2\ c(T) \leq 2\ c(H^*)$

So, $c(H) \leq 2\ c(H^*)$, and thus `Approx-TSP-Tour` is a 2-approximation algorithm for TSP.

Vertex Cover Approximations

Recall that a **vertex cover** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$ then $u \in V'$ or $v \in V'$ or both (there is a vertex in V' "covering" every edge in E).

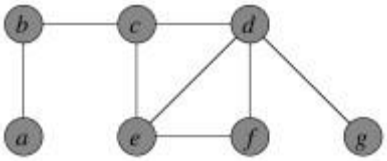
The optimization version of the **Vertex Cover Problem** is to find a vertex cover of minimum size in G.

We previously showed by reduction of CLIQUE to VERTEX-COVER that the corresponding decision problem is NP-Complete, so the optimization problem is NP-Hard.

Approx-Vertex-Cover

Vertex Cover can be approximated by the following surprisingly simple algorithm, which iterately chooses an edge that is not covered yet and covers it:

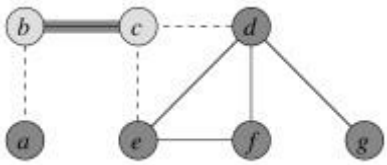
```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```



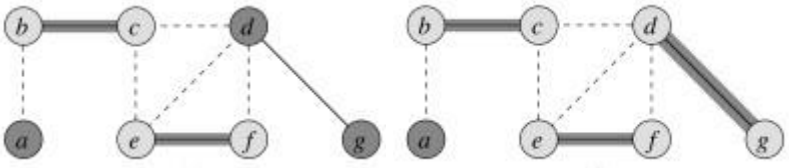
Example

Suppose we have this input graph:

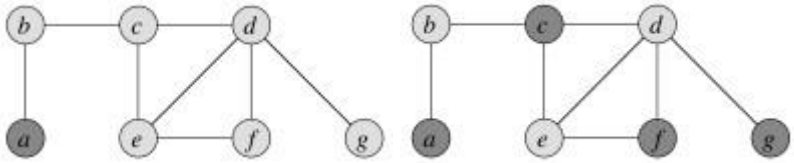
Suppose then that edge $\{b, c\}$ is chosen. The two incident vertices are added to the cover and all other incident edges are removed from consideration:



Iterating now for edges $\{e, f\}$ and then $\{d, g\}$:



The resulting vertex cover is shown on the left and the optimal vertex on the right:



(Would the approximation bound be tighter if we always chose an edge with the highest degree vertex remaining? Let's try it on this example. Would it be tighter in general? See 35.1-3.)

Analysis

How good is the approximation? We can show that the solution is within a factor of 2 of optimal.

Theorem: **Approx-Vertex-Cover is a polynomial time 2-approximation algorithm for Vertex Cover.**

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```

Proof: The algorithm is correct because it loops until every edge in *E* has been covered.

The algorithm has O(|*E*|) iterations of the loop, and (using aggregate analysis, Topic 15) across all loop iterations, O(|V|) vertices are added to *C*. Therefore it is O(*E* + *V*), so is polynomial.

It remains to be shown that the solution is no more than twice the size of the optimal cover. We'll do so by finding a lower bound on the optimal solution *C**.

Let *A* be the set of edges chosen in line 4 of the algorithm. Any vertex cover must cover at least one endpoint of every edge in *A*. No two edges in *A* share a vertex (see algorithm), so in order to cover *A*, the optimal solution *C** must have at least as many vertices:

$A \leq C^*$

Since each execution of line 4 picks an edge for which neither endpoint is yet in *C* and adds these two vertices to *C*, then we know that

$C = 2A$

Therefore:

$C \leq 2C^*$

That is, |*C*| cannot be larger than twice the optimal, so is a 2-approximation algorithm for Vertex Cover.

11. Divide and Conquer Algorithms

11.1. Essential Schematic

Given a function to compute on *N* inputs, the divide-and-conquer strategy suggests splitting the inputs into *k* distinct subsets, *1<k<=N* yielding *k* subproblems. These subproblems must be solved and then a method must be found to combine subsolutions into a solution of the whole.

11.2. Control Abstraction of Divide and Conquer Algorithms

Control abstraction refers to a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined.

```
1   Algorithm DAndC(P)
2   {
3       if Small(P) then return S(P);
4       else
5       {
6           divide P into smaller instances P_1, P_2, ..., P_k, k ≥ 1;
7           Apply DAndC to each of these subproblems;
8           return Combine(DAndC(P_1),DAndC(P_2),...,DAndC(P_k));
9       }
10  }
```

Algorithm 3.1 Control abstraction for divide-and-conquer

computing time of DAndC is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases} \tag{3.1}$$

where $T(n)$ is the time for DAndC on any input of size n and $g(n)$ is the time to compute the answer directly for small inputs. The function $f(n)$ is the time for dividing P and combining the solutions to subproblems. For divide-and-conquer-based algorithms that produce subproblems of the same type as the original problem, it is very natural to first describe such algorithms using recursion.

The complexity of many divide-and-conquer algorithms is given by recurrences of the form

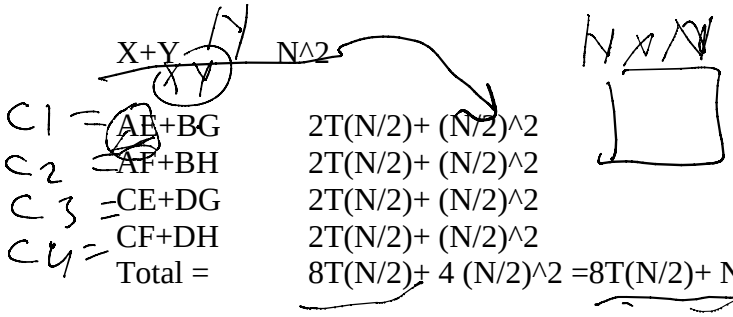
$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases} \tag{3.2}$$

where a and b are known constants. We assume that $T(1)$ is known and n is a power of b (i.e., $n = b^k$).

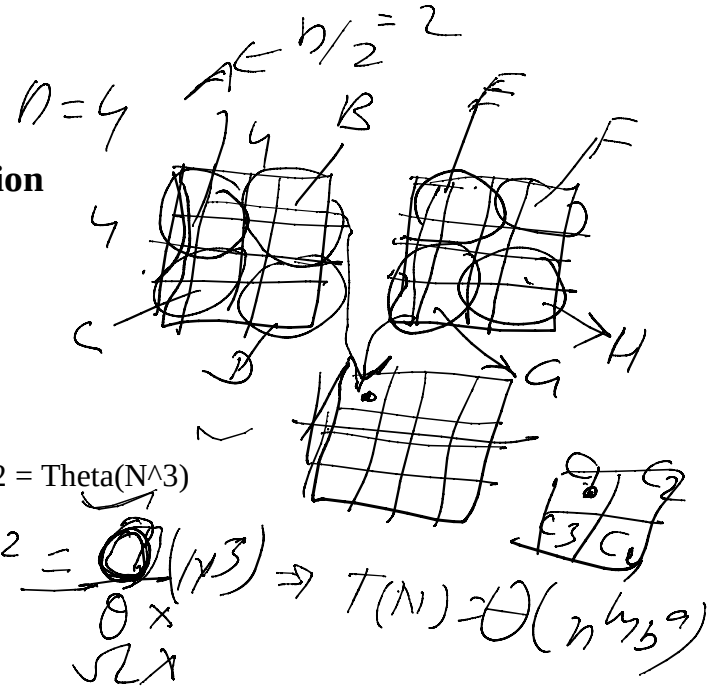
One of the methods for solving any such recurrence relation is called the *substitution method.* This method repeatedly makes substitution for each occurrence of the function T in the right-hand side until all such occurrences disappear.

11.3. Balancing
11.4. Min_Max Problem solution with Divide and Conquer

Divide and Conquer- Matrix Multiplication



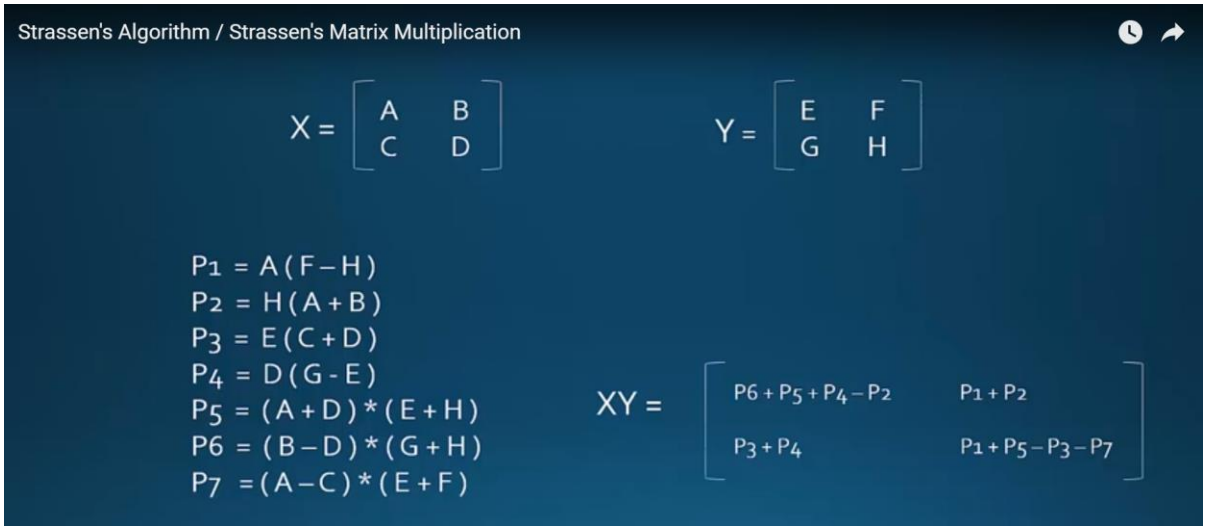
C1 =	AE+BG	2T(N/2)+ (N/2)^2
C2 =	AF+BH	2T(N/2)+ (N/2)^2
C3 =	CE+DG	2T(N/2)+ (N/2)^2
C4 =	CF+DH	2T(N/2)+ (N/2)^2
Total =		8T(N/2)+ 4 (N/2)^2 =8T(N/2)+ N^2 = Theta(N^3)

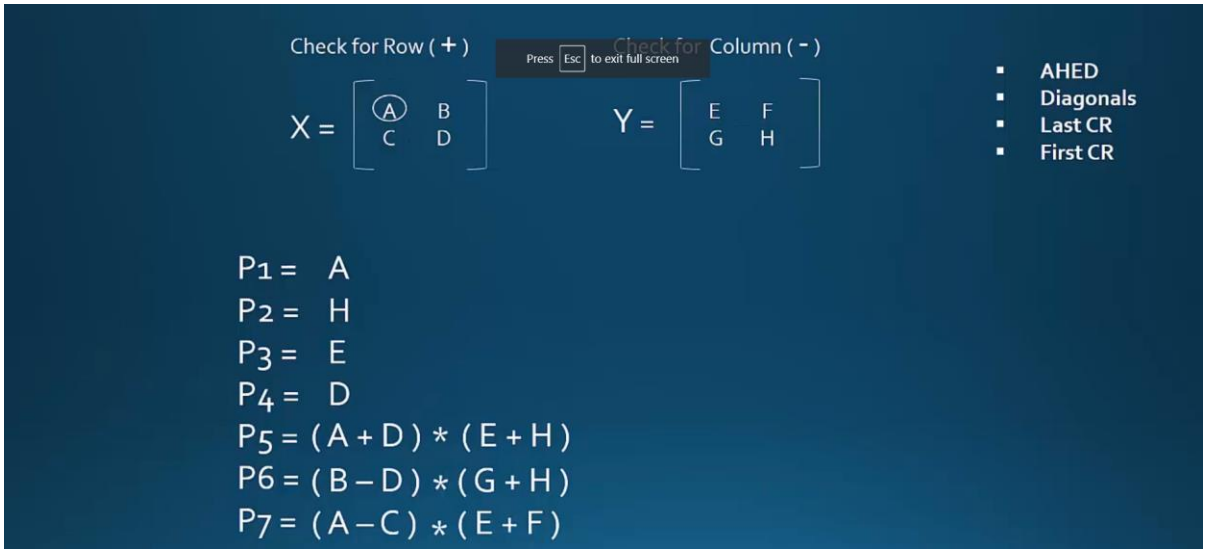


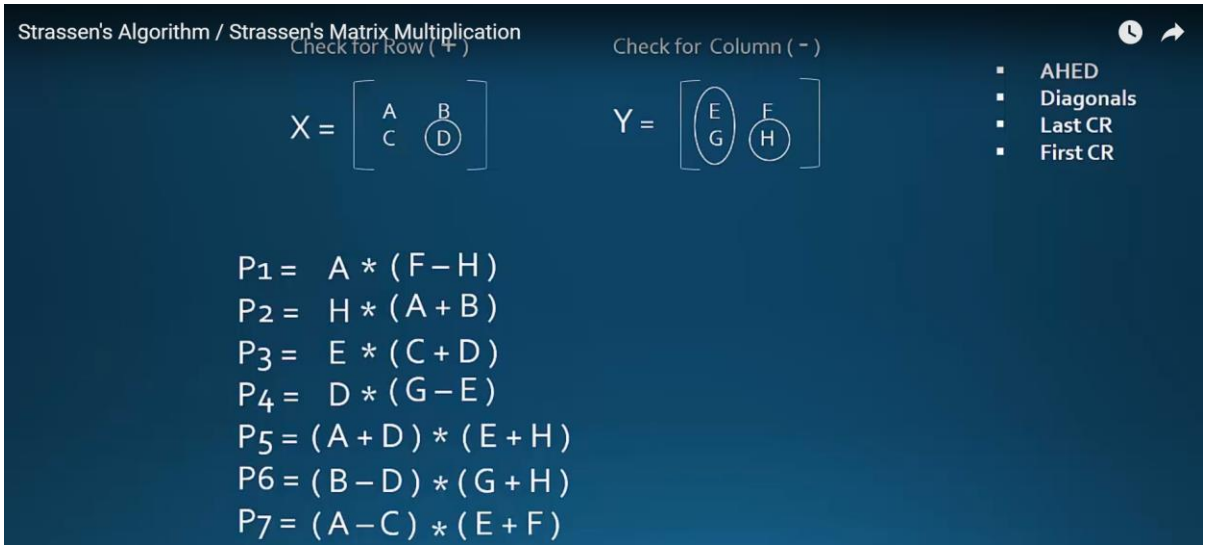
$N^2 = \Omega(N^3) \Rightarrow T(N) = \Theta(n^{\log_3 9})$

Ω x

Ω x


Strassen's Algorithm / Strassen's Matrix Multiplication
X = [A B; C D]
Y = [E F; G H]
P1 = A(F−H)
P2 = H(A+B)
P3 = E(C+D)
P4 = D(G-E)
P5 = (A+D)*(E+H)
P6 = (B−D)*(G+H)
P7 = (A−C)*(E+F)
XY = [P6+P5+P4−P2 P1+P2; P3+P4 P1+P5−P3−P7]


Check for Row (+)
Press Esc to exit full screen
Check for Column (-)
X = [A B; C D]
Y = [E F; G H]
AHED
Diagonals
Last CR
First CR
P1 = A
P2 = H
P3 = E
P4 = D
P5 = (A+D)*(E+H)
P6 = (B−D)*(G+H)
P7 = (A−C)*(E+F)


Strassen's Algorithm / Strassen's Matrix Multiplication
Check for Row (+)
Check for Column (-)
X = [A B; C D]
Y = [E F; G H]
AHED
Diagonals
Last CR
First CR
P1 = A*(F−H)
P2 = H*(A+B)
P3 = E*(C+D)
P4 = D*(G−E)
P5 = (A+D)*(E+H)
P6 = (B−D)*(G+H)
P7 = (A−C)*(E+F)

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

$$n^2 = \underline{\quad} \Rightarrow T(n) = \Theta(n^{\log_2 7})$$

$$\Rightarrow \Theta(n^{2.79})$$

Divide And conquer => Large Integers Multiplication

Multiplication of Large Integers :

When the size (the no. of computer words required, or the length of the representation in decimal or binary), of operand is too long to be held in a single word of the computer in use, even the the operations like addition, multiplication etc. may not be considered elementary.

e.g. For two integers of sizes m and n (no. of digits) respectively:-

The ordinary classical multiplication algorithm takes $m*n$ time.

More efficient algorithms :

⇒ Divide and Conquer takes time in the order of $n\,m^{\log 3/2} \approx n\,m^{.5}$

n = size of the larger operand, m = ---smaller

i.e. if $m=n \Rightarrow$ time $= n^{1.59}$ i.e. $< n^2$.

ref: p-4, p-220.

planning a divide and conquer based solution :

eg. 981×1234 can be performed as below.

$0981 \Rightarrow w = 09,\ x = 81$
$1234 \Rightarrow y = 12,\ z = 34$
$\Rightarrow 981 = 10^2 w + x$
$1234 = 10^2 y + z$

$$\Rightarrow 0981 \times 1234 = (10^2 w + x)(10^2 y + z)$$
$$= 10^4 wy + 10^2 (wz + xy) + xz$$
$$= 1080000 + 127800 + 2754 = 1210554$$

exact value as could have been obtained by direct multiplication.

Now:

Let us consider $r = (w+x)(y+z) = wy + (wz+xy) + xz$

Further let $p = wy = 9 \times 12 = 108$, $q = xz = 81 \times 34 = 2754$

$r = (w+x)(y+z) = 90 \times 46 = 4140$

$\Rightarrow 981 \times 1234 = 10^4 p + 10^2 (r - p - q) + q = 1080000 + 127800 + 2754 = 1210554$

$t(n) = 3\,t(n/2) + g(n)$ where $g(n)$ = time needed for additions, shifts (for multiplication with 10^4 or 10^2) and other overhead ops over n size numbers.

$\Rightarrow t(n) \in \Theta(n^{\log 3} | n = 2^k) = \Theta(n^{1.58})$

* Neelgagan improvement over $\Theta(n^2)$

Divide And conquer => O(n) time Selection/ Finding k-th Smallest item/ Finding Median [Ref: Cormen pg 220]

(4.b) O(n) time scheme for finding Median of array of integers

Begin SELECT (A, L, V)

1. IF n=1, then return A[L] as median
2. 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.
3. Find the median of each of the $\lceil n/5 \rceil$ groups by first ~~insertion~~ insertion sorting the elements of each group and then picking the median from the sorted list of group elements
4. Use SELECT recursively to find the median x of the $\lceil n/5 \rceil$ medians found in step 3.
5. Partition the input array around the median-of-medians x using 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 (n-k) elements on the high side of the partition.
6. If i=k return x. Otherwise use SELECT recursively to find the ~~median~~ i^{th} smallest element on the low side of i<k, or the (i-k)th smallest element on the high side if i>k.

Time Complexity :-

Now, At least half of the medians found in step 3 are greater than the median of medians.

⇒ At least half of the $\lceil n/5 \rceil$ groups contribute 3 elements that are greater than x, except for one group that has fewer than 5 elements if $n \% 5 \neq 0$, and another group containing x itself.

⇒ The no. of elements greater than x is at least $3(\lceil \frac{1}{2}\lceil n/5 \rceil \rceil - 2) \geq \frac{3n}{10} - 6$

⇒ No. of elements < x is at least $(\frac{7n}{10} + 6)$.

⇒ Thus in worst case, SELECT is called recursively on at most $(\frac{7n}{10} + 6)$ elements in step 6.

Thus step 2,3,5 :- O(n) ; step 4 :- $T(\lceil n/5 \rceil)$; step 6 :- almost $T(\frac{7n}{10} + 6)$

$$\Rightarrow T(n) = \begin{cases} \Theta(1) & , n \leq 140 \\ T(\lceil n/5 \rceil) + T(\frac{7n}{10} + 6) + O(n) & , n > 140 \end{cases} \quad \text{— (A)}$$

The recurrence can be solved as below:-

$T(n) \le c\lceil n/5 \rceil + c(7n/10 + 6) + an$

$\le cn/5 + c\frac{7n}{10} + 6c + an$

$= \frac{9cn}{10} + 6c + an = cn + (-cn/10 + 6c + an)$

$= cn$ (if $-\frac{cn}{10} + 7c + an \le 0$)

$\Rightarrow \boxed{T(n) = O(n)}$ Thus the desired time complexity is justified

Why D-n-C algo?

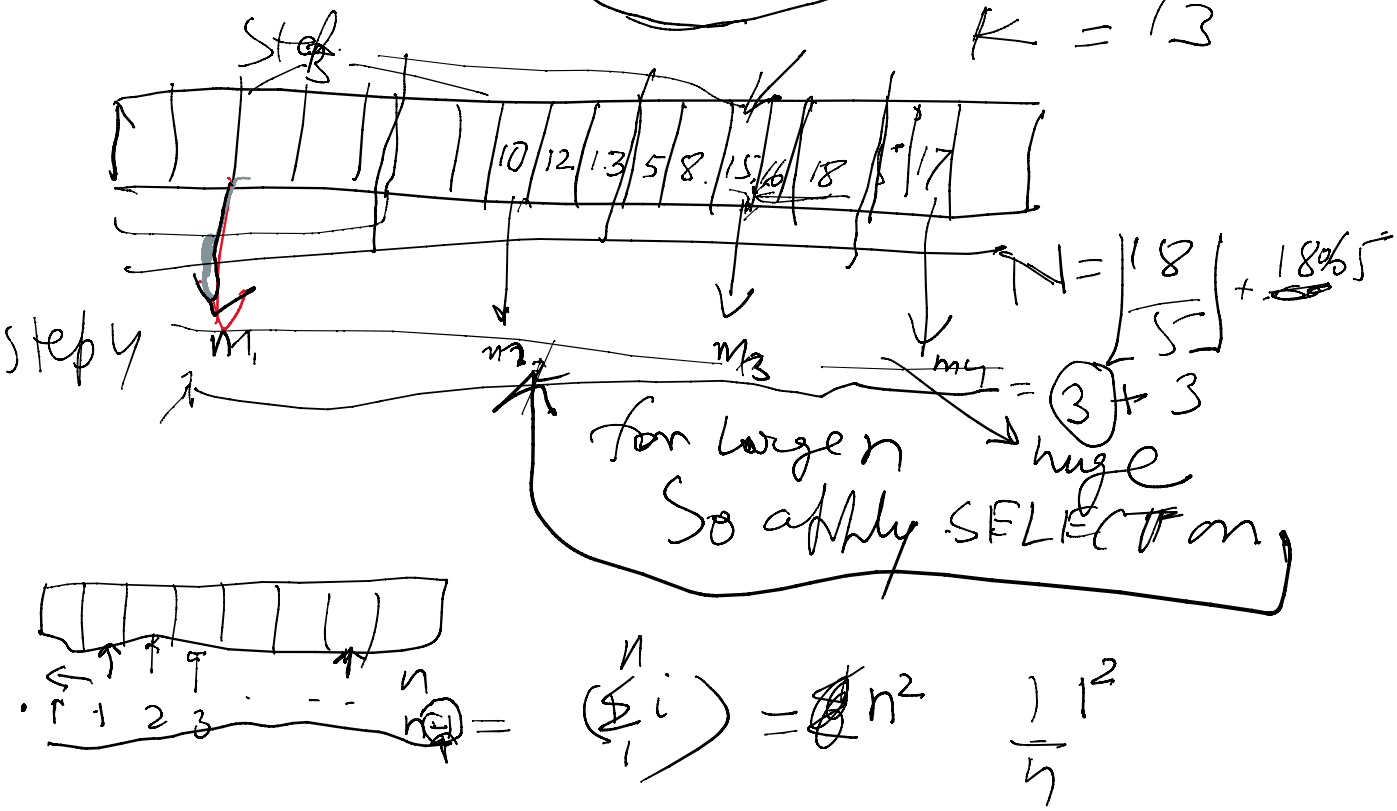
Kth smallest $\equiv$ sorting + searching for kth

non-divide-n-conquer algorithm

$\hookrightarrow$ Time $\rightarrow \Theta(n^2)$ for i = 1 to n

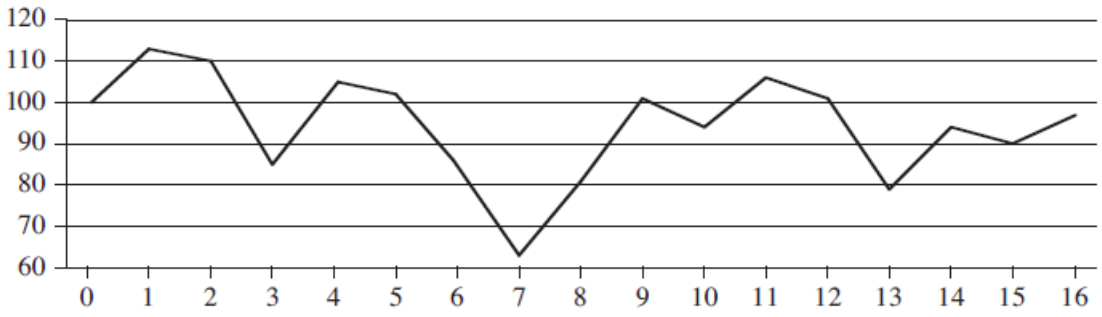
for j = 1 to n--

$\Theta(n \log n)$



$= \left(\sum_{1}^{n} i\right) = \frac{1}{2} n^2$

Transform (Divide) and conquer
The maximum-subarray problem – Cormen page 68



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

Figure 4.1 Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

Transformation- Instead of looking at daily price, consider the day-wise change in price.

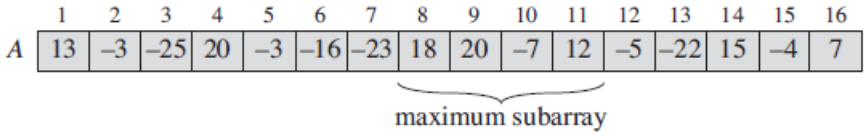


Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8..11]$, with sum 43, has the greatest sum of any contiguous subarray of array A.

A brute-force solution

We can easily devise a brute-force solution to this problem: just try every possible pair of buy and sell dates in which the buy date precedes the sell date. A period of n days has $\binom{n}{2}$ such pairs of dates. Since $\binom{n}{2}$ is $\Theta(n^2)$, and the best we can hope for is to evaluate each pair of dates in constant time, this approach would take $\Omega(n^2)$ time. Can we do better?

Divide and Conquer Solution

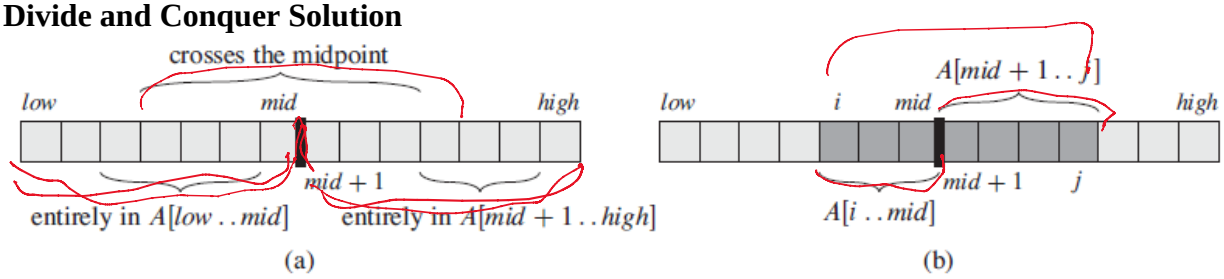
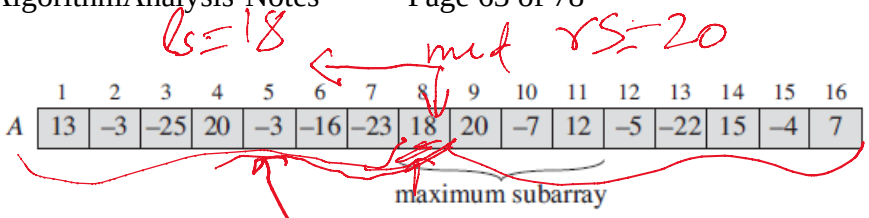


Figure 4.4 (a) Possible locations of subarrays of $A[low\,..\,high]$: entirely in $A[low\,..\,mid]$, entirely in $A[mid + 1\,..\,high]$, or crossing the midpoint mid. (b) Any subarray of $A[low\,..\,high]$ crossing the midpoint comprises two subarrays $A[i\,..\,mid]$ and $A[mid + 1\,..\,j]$, where $low \le i \le mid$ and $mid < j \le high$.

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

	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



FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$

```
1  left-sum = −∞
2  sum = 0
3  for i = mid downto low
4      sum = sum + A[i]
5      if sum > left-sum
6          left-sum = sum
7          max-left = i
8  right-sum = −∞
9  sum = 0
10 for j = mid + 1 to high
11     sum = sum + A[j]
12     if sum > right-sum
13         right-sum = sum
14         max-right = j
15 return (max-left, max-right, left-sum + right-sum)
```

Time Complexity

already seen, the call to FIND-MAX-CROSSING-SUBARRAY in line 6 takes $\Theta(n)$ time. Lines 7–11 take only $\Theta(1)$ time. For the recursive case, therefore, we have

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n) . \end{aligned} \tag{4.6}$$

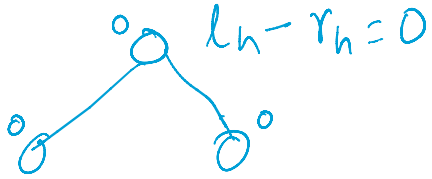
Combining equations (4.5) and (4.6) gives us a recurrence for the running time $T(n)$ of FIND-MAXIMUM-SUBARRAY:

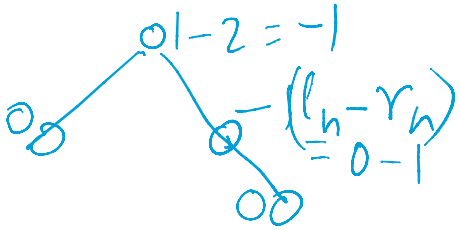
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases} \tag{4.7}$$

This recurrence is the same as recurrence (4.1) for merge sort. As we shall see from the master method in Section 4.5, this recurrence has the solution $T(n) = \Theta(n \lg n)$. You might also revisit the recursion tree in Figure 2.5 to understand why the solution should be $T(n) = \Theta(n \lg n)$.

DnC AVL — Height balanced tree

AVL Insert → Insert as per BST_Insert
→ Balance (± 1) $l(T) - r(T)$ → bf(T) ← postorder

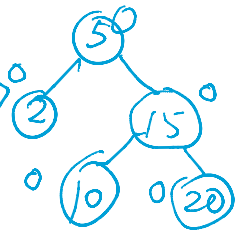




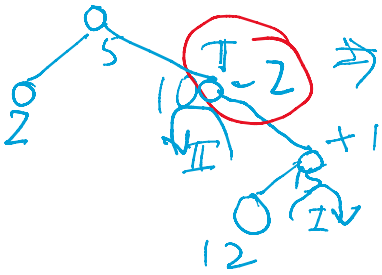
(I)

balancing

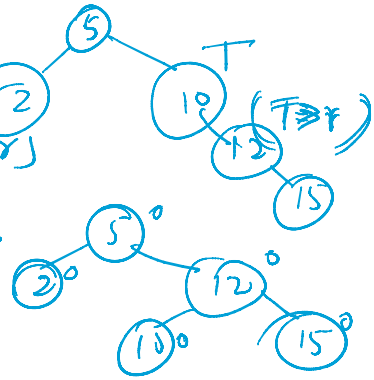
$bf(T) = -2$ && $bf(T \to r) = -1$ ⇒ left rotate at T



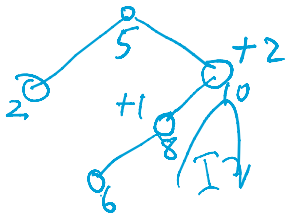
(II)



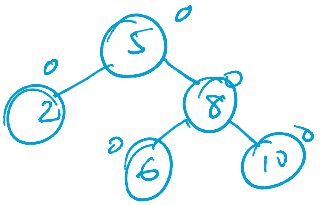
$bf(T) = -2$ && $bf(T \to r) = +1$ → I rr@T, II lr@T



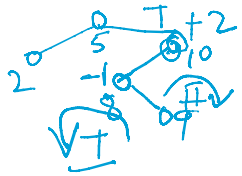
(III)



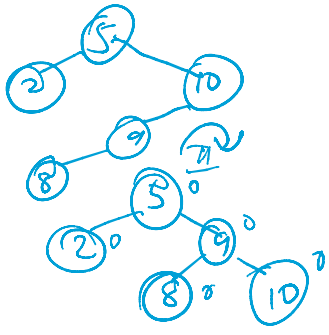
$bf(T) = +2$ && $bf(T \to l) = +1$ → rr@T



(IV)



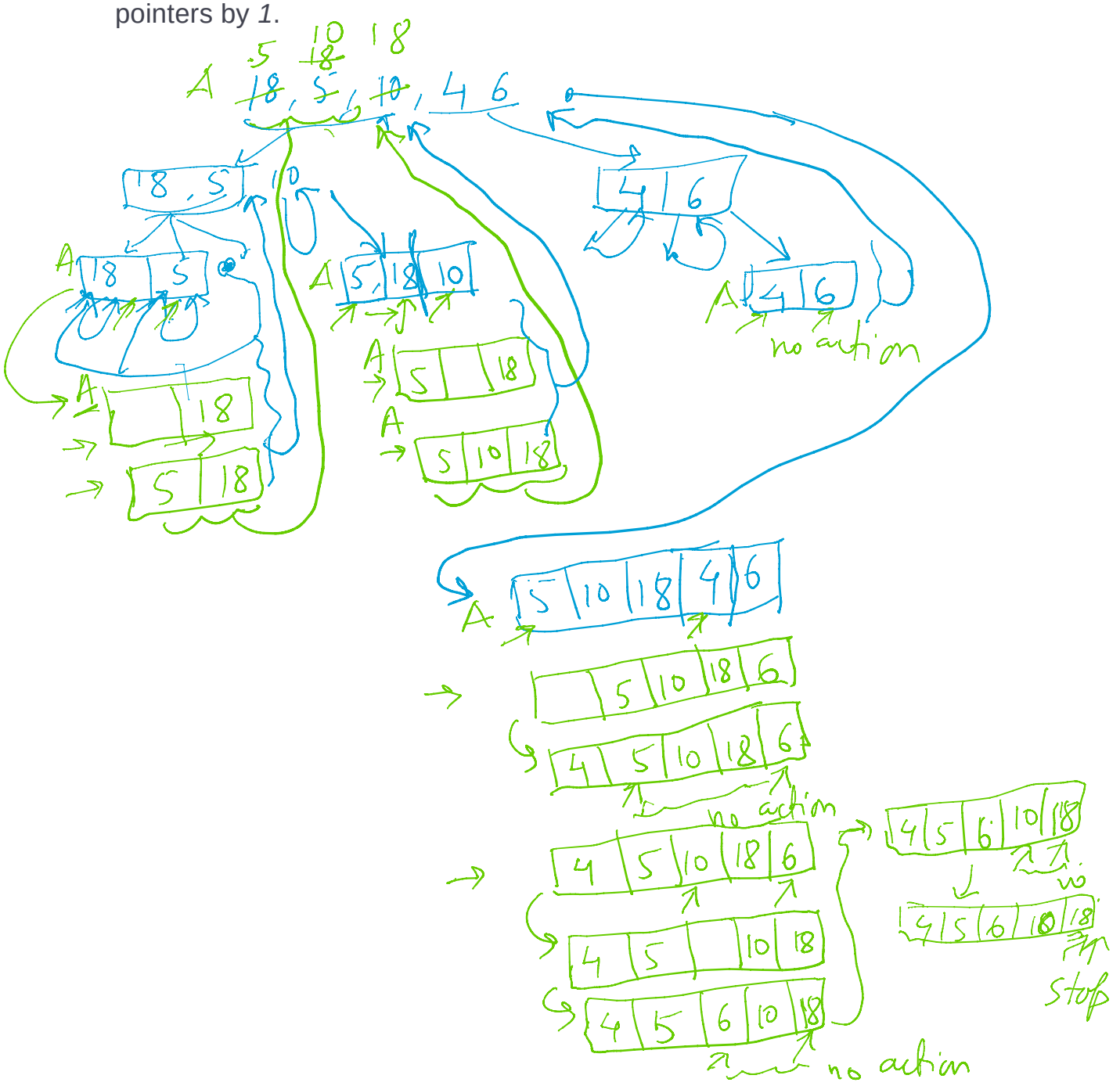
$bf(T) = +2$ && $bf(T \to l) = -1$ → I lr@(T→l), II rr@T



In – Place Merge-Sort

Merge(A,l,m,u)

- Maintain two pointers which point to start of the segments which have to be merged.
- Compare the elements at which the pointers are present.
- If *element1* < *element2* then *element1* is at right position, simply increase *pointer1*.
- Else shift all the elements between element1 and *element2(including element1 but excluding element2)* right by 1 and then place the element2 in the previous place *(i.e. before shifting right)* of element1. Increment all the pointers by *1*.



Time Complexity of above approach is $O(n^2)$ because merge is $O(n^2)$. Time complexity of standard merge sort is less, O(n Log n).

```
// C++ program in-place Merge Sort
#include <bits/stdc++.h>
using namespace std;

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
// Inplace Implementation
void merge(int arr[], int start, int mid, int end)
{
	int start2 = mid + 1;

	// If the direct merge is already sorted
	if (arr[mid] <= arr[start2]) {
		return;
	}

	// Two pointers to maintain start
	// of both arrays to merge
	while (start <= mid && start2 <= end) {

		// If element 1 is in right place
		if (arr[start] <= arr[start2]) {
			start++;
		}
		else {
			int value = arr[start2];
			int index = start2;

			// Shift all the elements between element 1
			// element 2, right by 1.
			while (index != start) {
				arr[index] = arr[index - 1];
				index--;
			}
			arr[start] = value;

			// Update all the pointers
			start++;
			mid++;
			start2++;
		}
	}
}

/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
```

```
{
	if (l < r) {

		// Same as (l + r) / 2, but avoids overflow
		// for large l and r
		int m = l + (r - l) / 2;

		// Sort first and second halves
		mergeSort(arr, l, m);
		mergeSort(arr, m + 1, r);

		merge(arr, l, m, r);
	}
}

/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
	int i;
	for (i = 0; i < size; i++)
		printf("%d ", A[i]);
	printf("\n");
}

/* Driver program to test above functions */
int main()
{
	int arr[] = { 12, 11, 13, 5, 6, 7 };
	int arr_size = sizeof(arr) / sizeof(arr[0]);

	mergeSort(arr, 0, arr_size - 1);

	printArray(arr, arr_size);
	return 0;
}
```

https://www.geeksforgeeks.org/merge-sort-with-o1-extra-space-merge-and-on-lg-n-time/

For integer types, merge sort can be made inplace using some mathematics trick of modulus and division. That means storing two elements value at one index and can be extracted using modulus and division.

First we have to find a value greater than all the elements of the array. Now we can store the original value as modulus and the second value as division. Suppose we want to store **arr[i]** and **arr[j]** both at index i(means in arr[i]). First we have to find a **'maxval'** greater than both arr[i] and arr[j]. Now we can store as **arr[i] = arr[i] + arr[j]*maxval**. Now **arr[i]%maxval** will give the original value of arr[i] and **arr[i]/maxval** will give the value of arr[j]. So below is the implementation on merge sort.

0/1 Knapsack Problem- Greedy Solution - http://www.c4learn.com/c-programs/program-to-implement-knapsack-problem.html

0/1 Knapsack Problem- Greedy Solution - http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf

Coin-Exchange Problem: Greedy Vs Dynamic- file:///E:/MyWork/Lecture/Algorithms/4.12.%20Dynamic%20Programming%20%E2%80%94%20Problem%20Solving%20with%20Algorithms%20and%20Data%20Structures.html
[https://interactivepython.org/runestone/static/pythonds/Introduction/toctree.html]

Python Algorithm Design- https://legacy.python.org/workshops/2002-02/papers/15/index.htm

Randomized

Balancing Principle
[https://www.usenix.org/legacy/events/hotpar11/tech/final_files/Czechowski.pdf]
Modern techniques of algorithm analysis explicitly count parallel operations and I/O operations. Given an algorithm analyzed this way and a cost model for some architecture we can estimate compute-time and I/O-time explicitly. We then connect the algorithm and architecture simply by applying the concept of balance, which says a computation running on some machine is efficient if the compute-time dominates the I/O (memory transfer) time, as suggested originally by Kung (1986) and since applied by numerous others. The result is a constraint equation that binds algorithm and architecture parameters together; we refer to this constraint as a balance principle.

Dynamic Programming

Fibonacci sequence[edit]

Here is a naïve implementation of a function finding the *n*th member of the Fibonacci sequence, based directly on the mathematical definition:

```
function fib(n)
    if n <= 1 return n
    return fib(n − 1) + fib(n − 2)
```

Notice that if we call, say, `fib(5)`, we produce a call tree that calls the function on the same value many different times. In larger examples, many more values of `fib`, or *subproblems*, are recalculated, leading to an exponential time algorithm.

Top-down approach: First break the problem into subproblems and then calculate and store values. requires only O(*n*) time instead of exponential time (but requires O(*n*) space):

```
var m := map(0 → 0, 1 → 1)
function fib(n)
    if key n is not in map m
        m[n] := fib(n − 1) + fib(n − 2)
    return m[n]
```

This technique of saving values that have already been calculated is called *memoization*;

Bottom-up approach: Calculate the smaller values of `fib` first, then build larger values from them. This method also uses O(*n*) time since it contains a loop that repeats n − 1 times, but it only takes constant (O(1)) space, in contrast to the top-down approach which requires O(*n*) space to store the map.

```
function fib(n)
    if n = 0
        return 0
    else
        var previousFib := 0, currentFib := 1
        repeat n − 1 times // loop is skipped if n = 1
            var newFib := previousFib + currentFib
            previousFib := currentFib
```

```
                currentFib  := newFib
                return currentFib
```

Coin Change Problem
Recursive Solution

```
def recMC(coinValueList,change):

   minCoins = change

   if change in coinValueList:

     return 1

   else:

      for i in [c for c in coinValueList if c <= change]:

         numCoins = 1 + recMC(coinValueList,change-i)

         if numCoins < minCoins:

            minCoins = numCoins

   return minCoins


print(recMC([1,5,10,25],63))
```

Dynamic Programming with Memoization (table-lookup)

```
def dpMakeChange(coinValueList,change,minCoins):

   for cents in range(change+1):

      coinCount = cents

      for j in [c for c in coinValueList if c <= cents]:

            if minCoins[cents-j] + 1 < coinCount:

               coinCount = minCoins[cents-j]+1

      minCoins[cents] = coinCount

   return minCoins[change]
```

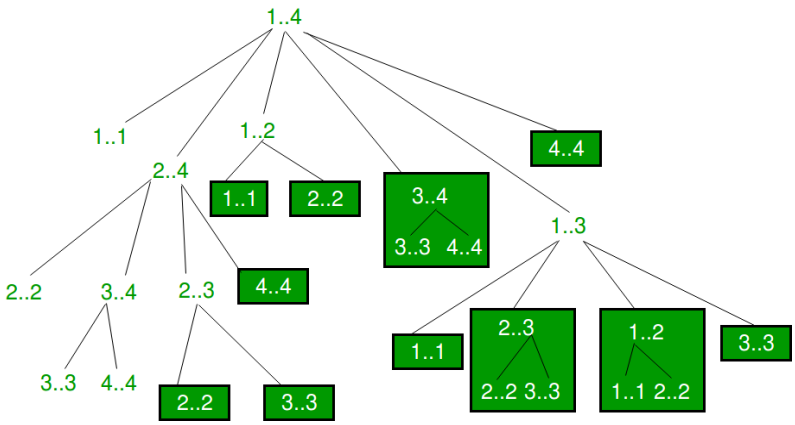
Matrix Chain Multiplication
Simple Recursive Solution

```
int MatrixChainOrder(int p[], int i, int j)
{
    if(i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;

    // place parenthesis at different places between first
    // and last matrix, recursively calculate count of
    // multiplications for each parenthesis placement and
    // return the minimum count
    for (k = i; k <j; k++)
    {
        count = MatrixChainOrder(p, i, k) +
                MatrixChainOrder(p, k+1, j) +
                p[i-1]*p[k]*p[j];

        if (count < min)
            min = count;
    }

    // Return minimum count
    return min;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 2, 3, 4, 3};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
                          MatrixChainOrder(arr, 1, n-1));

    getchar();
    return 0;
}
```

Dynamic Programming with Memoization

```
int MatrixChainOrder(int p[], int n)
{

    /* For simplicity of the program, one extra row and one
```

```
        extra column are allocated in m[][].  0th row and 0th
        column of m[][] are not used */
    int m[n][n];

    int i, j, k, L, q;

    /* m[i,j] = Minimum number of scalar multiplications needed
        to compute the matrix A[i]A[i+1]...A[j] = A[i..j] where
        dimension of A[i] is p[i-1] x p[i] */

    // cost is zero when multiplying one matrix.
    for (i=1; i<n; i++)
        m[i][i] = 0;

    // L is chain length.
    for (L=2; L<n; L++)
    {
        for (i=1; i<n-L+1; i++)
        {
            j = i+L-1;
            m[i][j] = INT_MAX;
            for (k=i; k<=j-1; k++)
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

    return m[1][n-1];
}
```



Greedy Algorithm

Applied to solve optimization problem.

→ optimality criterion (1)

→ constraint (2)

0/1 knapsack problem

O = {O1, O2, O3, O4, O5}

P = {100, 150, 200, 50, 80}

W = {20, 30, 40, 8, 8}

P/W = {5, 5, 5, 6.2, 10}

C = 60

by wt → <1, 0, 0, 1, 1> 230

by p → <1, 0, 1, 0, 0> 300

by P/W → <0, 0, 1, 1, 1> 330

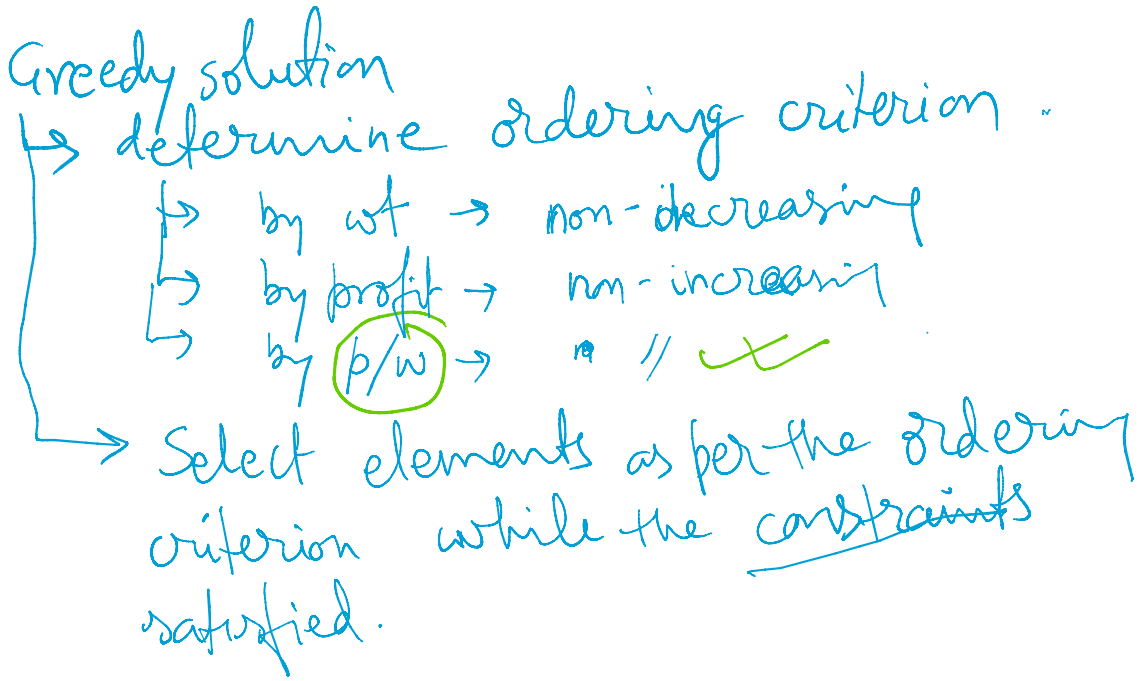
(1) maximize profit → $\max \sum_{i=1}^{N} p_i x_i$

(2) s.t. with capacity → $\sum_{i=1}^{N} w_i x_i \leq C$

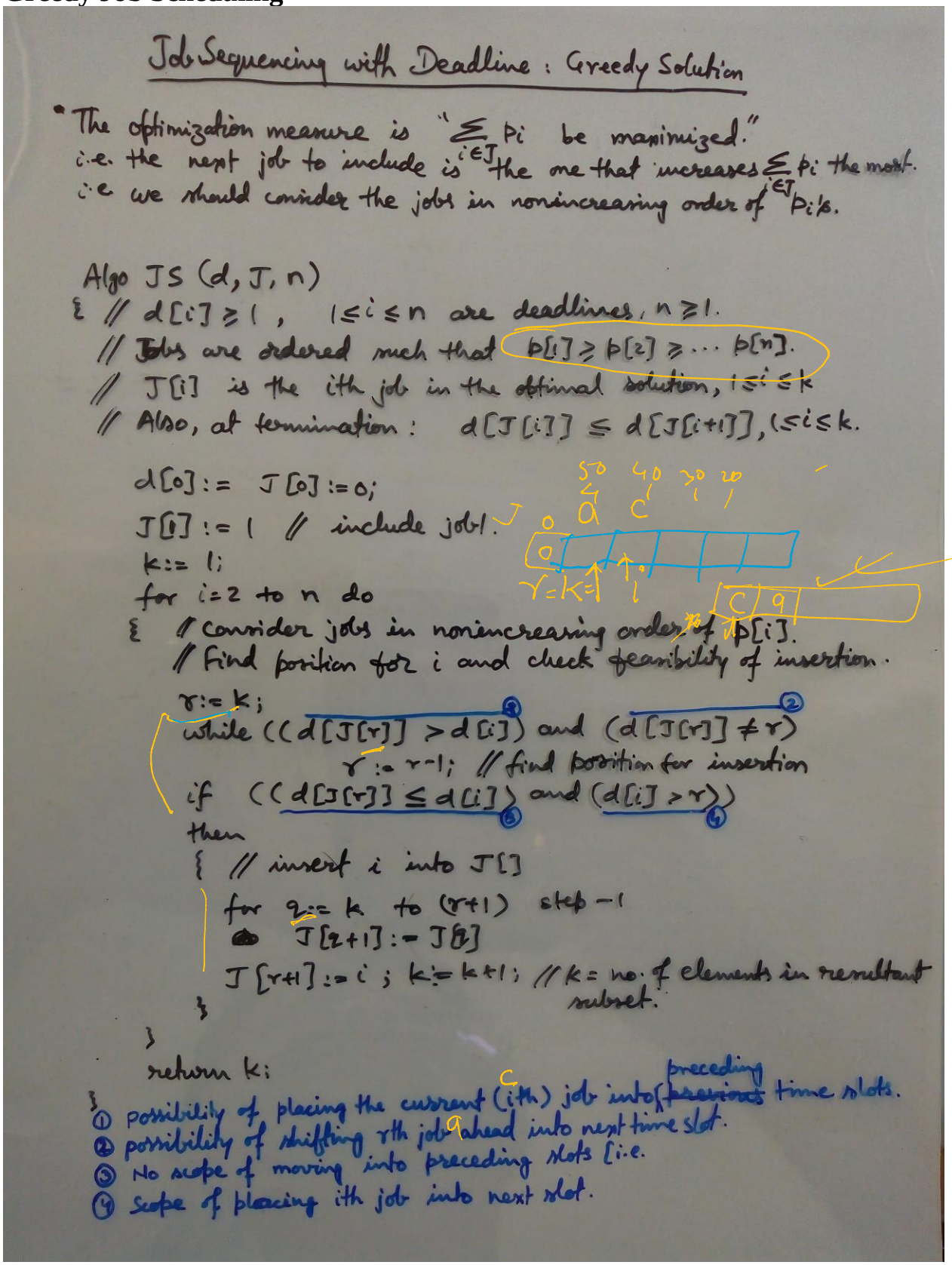
Solution: → feasible solutions - those satisfying the constraint

→ optimal solution (s) - the feasible solution producing most suitable as per the optimality criterion

Greedy Solution → Selection of the ~~best~~ immediate as per optimality function (without considering further effects)


Greedy solution
determine ordering criterion.
by wt → non-decreasing
by profit → non-increasing
by p/w → " //
Select elements as per the ordering criterion while the constraints satisfied.

Greedy Job Scheduling



Job Sequencing with Deadline : Greedy Solution

- The optimization measure is "$\sum_{i \in J} p_i$ be maximized."
  i.e. the next job to include is the one that increases $\sum_{i \in J} p_i$ the most.
  i.e. we should consider the jobs in nonincreasing order of p_i's.

```
Algo JS (d, J, n)
{ // d[i] ≥ 1 , 1≤i≤n are deadlines, n ≥ 1.
  // Jobs are ordered such that p[1] ≥ p[2] ≥ ... p[n].
  // J[i] is the ith job in the optimal solution, 1≤i≤k
  // Also, at termination : d[J[i]] ≤ d[J[i+1]], 1≤i≤k.

  d[0] := J[0] := 0;
  J[1] := 1 // include job1.
  k := 1;
  for i := 2 to n do
  { // consider jobs in nonincreasing order of p[i].
    // find position for i and check feasibility of insertion.
    r := k;
    while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)
          r := r-1; // find position for insertion
    if ((d[J[r]] ≤ d[i]) and (d[i] > r))
    then
    { // insert i into J[]
      for q := k to (r+1) step -1
        J[q+1] := J[q]
      J[r+1] := i; k := k+1; // k = no. of elements in resultant subset.
    }
  }
  return k;
}
```

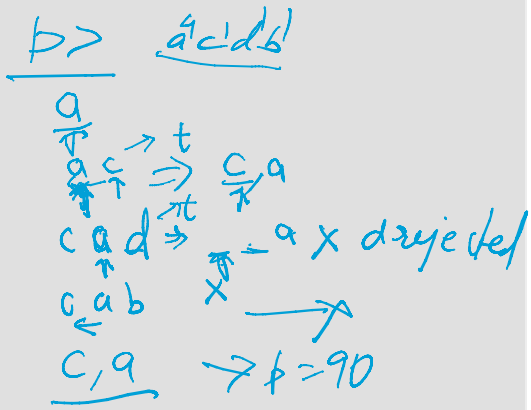
① possibility of placing the current (ith) job into preceding time slots.
② possibility of shifting rth job ahead into next time slot.
③ No scope of moving into preceding slots [i.e.
④ Scope of placing ith job into next slot.

Input: Four Jobs with following deadlines and profits

JobID	Deadline	Profit
a	4	~~20~~ 50
b	1	10
c	1	40
d	1	30

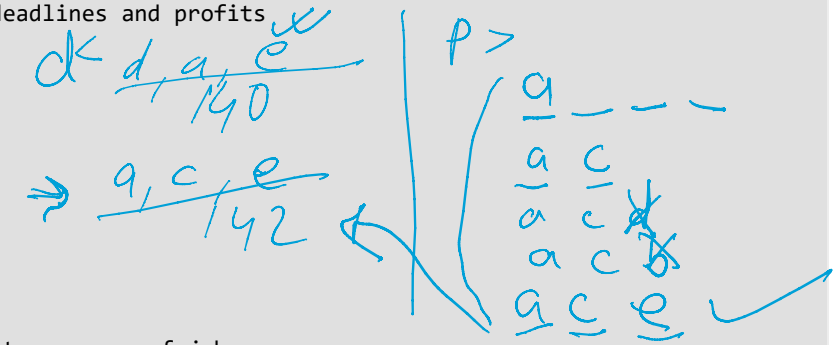
Output: Following is maximum profit sequence of jobs

c, a



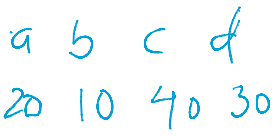
Input: Five Jobs with following deadlines and profits

JobID	Deadline	Profit
a	2	100
b	1	19
c	2	27
d	1	25
e	3	15



Output: Following is maximum profit sequence of jobs

c, a, e



Amortized Analysis

Amortized analysis.

Amortized analysis is concerned with the overall cost of a sequence of operations. It does not say anything about the cost of a specific operation in that sequence. For example, it is invalid to reason, “The amortized cost of insertion into a splay tree with n items is O(log n), so when I insert '45' into this tree, the cost will be O(log n).” In fact, inserting '45' might require O(n) operations! It is only appropriate to say, “When I insert m items into a tree, the average time for each operation will be O(log n).”

Because an amortized bound says nothing about the cost of individual operations, it may be possible that one operation in the sequence requires a huge cost.
worstcase analysis can give overly pessimistic bounds for sequences of operations, because such analysis ignores interactions among different operations on the same data structure (Tarjan 1985). Amortized analysis may lead to a more realistic worstcase bound by taking these interactions into account.

Amortized analysis is similar to averagecase analysis, in that it is concerned with the cost averaged over a sequence of operations. However, averagecase analysis relies on probabilistic assumptions about the data structures and operations in order to compute an *expected* running time of an algorithm. Its applicability is therefore dependent on certain assumptions about probability distributions on algorithm inputs, which means the analysis is invalid if these assumptions do not hold (or that probabilistic analysis cannot be used at all, if input distributions cannot be described!)

Amortized analysis needs no such assumptions. Also, it offers an *upperbound* on the worst case running time of a sequence of operations, and this bound will always hold. An averagecase bound, on the other hand, does not preclude the possibility that one will get “unlucky” and encounter an input that requires much

more than the expected computation time, even if the assumptions on the distribution of inputs are valid.

Amortized analysis is closely related to competitive analysis, which involves comparing the worstcase performance of an **online algorithm** to the performance of an optimal offline algorithm on the same data.

There exist three main approaches to amortized analysis: aggregate analysis, the accounting method, and the potential method (**Cormen, p.451**).

- **Worst-case bound on sequence of operations.**
  - **no probability involved**
- **Ex: union-find.**
  - **sequence of m union and find operations starting with n singleton sets takes O((m+n) α(n)) time.**
  - **single union or find operation might be expensive, but only α(n) on average**

Dynamic tables.

- **Store items in a table (e.g., for open-address hash table, heap).**
- **Items are inserted and deleted.**
  - **too many items inserted $\Rightarrow$ copy all items to larger table**
  - **too many items deleted $\Rightarrow$ copy all items to smaller table**

Amortized analysis.

- **Any sequence of n insert / delete operations take O(n) time.**
- **Space used is proportional to space required.**
- **Note: actual cost of a single insert / delete can be proportional to n if it triggers a table expansion or contraction.**

Bottleneck operation.

- **We count insertions (or re-insertions) and deletions.**
- **Overhead of memory management is dominated by (or proportional to) cost of transferring items.**