Algorithms Design and Analysis [ETCS-301]

Dr. A K Yadav
Amity School of Engineering and Technology
(affiliated to GGSIPU, Delhi)
akyadav1@amity.edu
akyadav@akyadav.in
www.akyadav.in
+91 9911375598

November 9, 2019

Introduction to algorithms I

- What is an algorithm?
  - An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
  - An algorithm is a sequence of computational steps that transform the input into output.
  - An algorithm is a sequence of operations performed on data that have to be organized in data structures.
  - A finite set of instructions that specify a sequence of operations to be carried out in order to solve a specific problem or class of problems.
  - An algorithm is an abstraction of a program to be executed on a physical machine.
  - *An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.*

Introduction to algorithms II

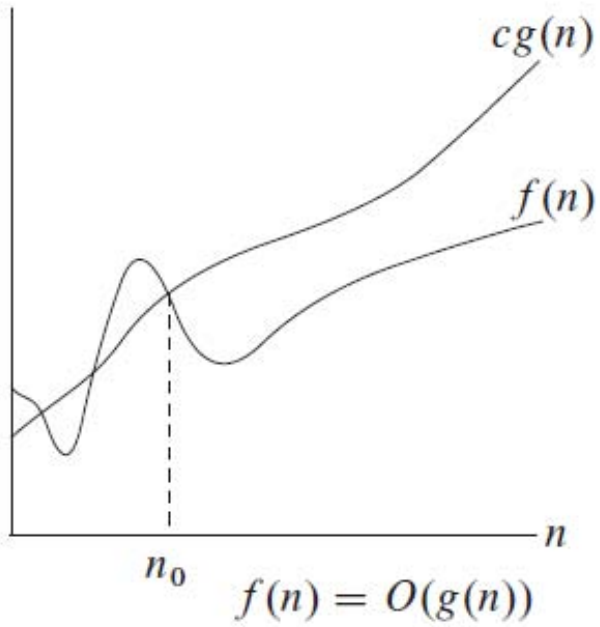
- Why do we study algorithms?
  - To make solution more faster.
  - To compare performance as a function of input size.
- Properties
  - Finiteness
  - Unambiguous
  - Sequence of execution
  - Input/output
  - Feasible
- Algoritm vs Program
  - Algorithm is independent of programming language and written in any natural language.
  - Implementation of any algorithm in a programming language is a program.

Introduction to algorithms III

- Design Techniques
  - Divide and Conquer (D&C)
  - Dynamic Programming (DP)
  - Greedy Approach
  - Branch and Bound
  - Backtracking
  - Randomized
- Random Access Machine (RAM)
  - Simply count primitive operations
  - Use pseudo code
  - Independent of programming language
  - Equal time for all operations

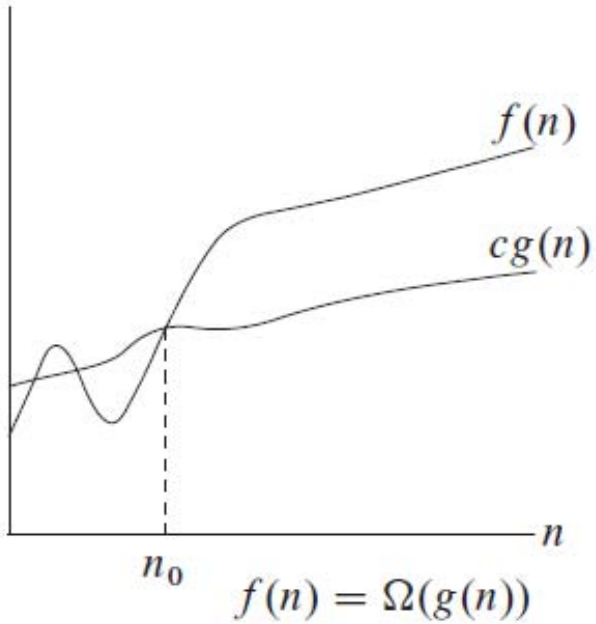
Asymptotic notations I

1. O - notation "Big O" : Asymptotic upper bound,
   $O(g(n)) = \{f(n) : \exists c, n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$
   $f(n) \in O(g(n))$
   $f(n) = O(g(n))$



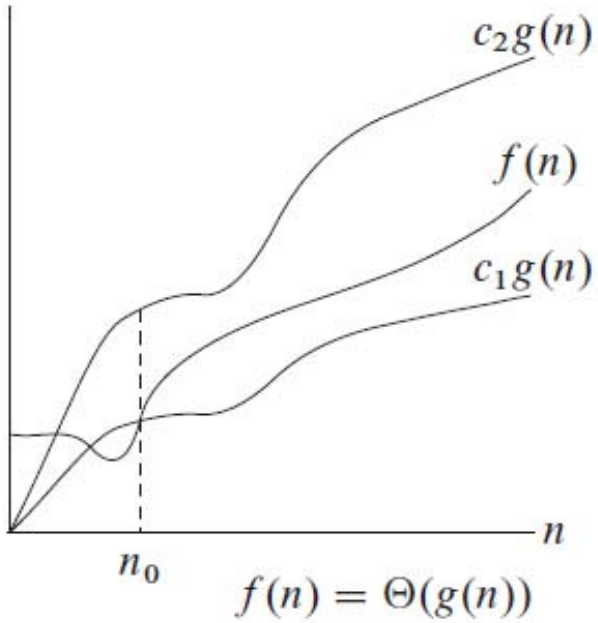
Asymptotic notations II

2. Ω - notation "Big omega" : Asymptotic lower bound,
   $\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$
   $f(n) \in \Omega(g(n))$
   $f(n) = \Omega(g(n))$



Asymptotic notations III

3. Θ - notation : Asymptotic tight bound,
   $\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$
   $f(n) \in \Theta(g(n))$
   $f(n) = \Theta(g(n))$



Asymptotic notations IV

4. o - notation "small o": Asymptotic loose upper bound, $o(g(n)) = \{f(n) : \forall c > 0, \exists n_0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$
5. ω - notation "small omega": Asymptotic loose lower bound, $\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$

Benefits of Asymptotic Notations:
- Simple representation of algorithm efficiency.
- Easy comparison of performance of algorithms.

Relationship between notations I

- $f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$
- $f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$
- $f(n) = O(g(n)) \& f(n) = \Omega(g(n)) \Rightarrow f(n) = \Theta(g(n))$
- $f(n) = O(g(n))$ may or may be tight upper bound but $f(n) = o(g(n))$ is always upper loose bound and $o(g(n)) \in O(g(n))$ and $o(g(n)) \subset O(g(n))$
- $f(n) = \Omega(g(n))$ may or may be tight lower bound but $f(n) = \omega(g(n))$ is always lower loose bound and $\omega(g(n)) \in \Omega(g(n))$ and $\omega(g(n)) \subset \Omega(g(n))$
- $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & f(n) = o(g(n)) \\ \infty & f(n) = \omega(g(n)) \\ c & 0 < c < \infty, f(n) = \Theta(g(n)) \end{cases}$$

Relationship between notations II

- Let $f(n) = \sum_{i=0}^{d} a_i n^i$ where $a_d > 0$, is a degree-d polynomial in n and k is a constant then:
  - if $k \geq d$, then $f(n) = O(n^k)$.
  - if $k \leq d$, then $f(n) = \Omega(n^k)$.
  - if $k = d$, then $f(n) = \Theta(n^k)$.
  - if $k > d$, then $f(n) = o(n^k)$.
  - if $k < d$, then $f(n) = \omega(n^k)$.

Examples

1. Show that $an + b = O(n^2)$
2. Show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
3. Show that $\frac{1}{2}n^2 - 3n = \Omega(n)$

Solutions I

1. Show that $an + b = O(n^2)$
   we have to prove that $an + b \leq cn^2$ for some positive constants c, n_0 and for all $n \geq n_0$.

$$
\begin{aligned}
an + b &\leq an + |b| \text{ for all } n \geq 1 \\
&\leq an + |b|n \text{ for all } n \geq 1 \\
&= (a + |b|)n \\
&= cn \text{ where } c = a + |b| \\
&\leq cn^2
\end{aligned}
$$

$$\Rightarrow an + b \leq cn^2 \text{ where } c = a + |b| > 0 \text{ all } n \geq n_0 = 1$$

$$\therefore an + b = O(n^2)$$

$$\Rightarrow an + b < cn^2 \text{ for all } c > 0 \text{ all } n > n_0 = max(1, -b/a)$$

$$\therefore an + b = o(n^2)$$

Solutions II

2. Show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
   We have to prove that $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$ for some positive constants c_1, c_2, n_0 and for all $n \geq n_0$.

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

$$\Rightarrow c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 \text{ after dividing } n^2$$

$$\Rightarrow 0 < c_1 \leq \frac{1}{2} - \frac{3}{n}$$

$$\Rightarrow 0 < \frac{1}{2} - \frac{3}{n}$$

$$\Rightarrow \frac{3}{n} < \frac{1}{2}$$

Solutions III

$\Rightarrow n > 6$

Let $n = 7$

$\Rightarrow 0 < c_1 \leq \frac{1}{14}$

Let $c_1 = \frac{1}{14}$

Now $0 < \frac{1}{2} - \frac{3}{n} \leq c_2$

$\Rightarrow 0 < \frac{1}{2} \leq c_2$ when $n = \infty$

Let $c_2 = \frac{1}{2}$

$\Rightarrow 0 < \frac{1}{14}n^2 \leq \frac{1}{2}n^2 - 3n \leq \frac{1}{2}n^2$ for all $n \geq n_0 = 7$ and any constant $0 < c_1 \leq \frac{1}{14} \leq \frac{1}{2} \leq c_2$.

Solutions IV

3. Show that $\frac{1}{2}n^2 - 3n = \Omega(n)$
   We have to prove that $cn \leq \frac{1}{2}n^2 - 3n$ for some positive constants c, n_0 and for all $n \geq n_0$.

$$\text{Let } cn \leq \frac{1}{2}n^2 - 3n$$

$$c \leq \frac{1}{2}n - 3 \text{ after dividing } n \text{ both side}$$

$$\Rightarrow c \leq \frac{1}{2}n + 3$$

$$\Rightarrow c \leq \frac{1}{2}n + 3n \text{ for all } n > 0$$

$$\Rightarrow c \leq \frac{7}{2}n$$

Solutions V

$$\Rightarrow \frac{7}{2} \leq \frac{7}{2}n \text{ for all } n \geq 1$$

$$\therefore c \leq \frac{7}{2}$$

Let $c = \frac{7}{2}$

$\Rightarrow \frac{7}{2}n \leq \frac{1}{2}n^2 - 3n$ for all positive constants $c \leq \frac{7}{2}$ and for all $n \geq n_0 = 1$.

Recurrences

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

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

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

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

How to solve recurrence relations?

- Substitution method
- Iteration method
- Recursion tree method
- Master method (D&C)
- Muster Theorem (S&C)

Substitution method

- In Substitution method there are two steps :
1 - First guess the solution
2 - Second use mathematical induction to find the constants and show that the solution works.
- This method is powerful, but depends on the correctness of the guess.
- Method can be used for either upper or lower bounds on a recurrence.

Example of Substitution method

1. Prove that the solution of $T(n) = 2T(n/2) + n$ is $O(n \lg n)$
2. Find the solution of the recurrence relation $T(n) = 2T(\sqrt{n}) + \lg n$
3. Show that the solution of $T(n) = 2T(n/2) + n$ is $\Omega(n \lg n)$
4. Show that the solution of $T(n) = T(n/2) + 1$ is $O(\lg n)$
5. Find the solution of $T(n) = T(n-1) + n$.

Solution Example 1 of Substitution method I

Example 1. Prove that the solution of $T(n) = 2T(n/2) + n$ is $O(n \lg n)$

Solution:

Soln is already given so no need to guess the soln. We have to prove that $T(n) \leq cn \lg n$ for some constant $c > 0$ and for all $n > n_0$.

$$T(n) = 2T(n/2) + n \tag{1}$$

$$T(n) = O(n \lg n) \Rightarrow T(n) \leq cn \lg n \tag{2}$$

$\Rightarrow T(n) = O(n \lg n)$ then it must be the solution for the input size $n/2$.

substitute the value of n as $n/2$ in Eq. 2.

Solution Example 1 of Substitution method II

$$T(n/2) \leq c(n/2)\lg(n/2) \tag{3}$$

Substitute the value of $T(n/2)$ from Eq. 3 into Eq. 1.

$$T(n) \leq 2(c(n/2)\lg(n/2)) + n$$

$$\Rightarrow T(n) \leq cn\lg(n/2) + n$$

$$\Rightarrow T(n) \leq cn\lg n - cn\lg 2 + n$$

$$\Rightarrow T(n) \leq cn\lg n - cn + n$$

$$\Rightarrow T(n) \leq cn\lg n - n(c-1)$$

$$\Rightarrow T(n) \leq cn\lg n \ \forall c \geq 1, n \geq 2$$

$$\therefore T(n) = O(n\lg n)$$

Solution Example 2 of Substitution method I

Example 2. Find the solution of the recurrence relation
$T(n) = 2T(\sqrt{n}) + \lg n$
Solution:
Step 1 is to guess the solution but it is very hard to guess for the given recurrence relation.
So First we have to simplify the relation by changing variables.

$$T(n) = 2T(\sqrt{n}) + \lg n \text{ (given)} \tag{4}$$

Take:

$$\lg n = m \tag{5}$$

$$\Rightarrow n = 2^m \tag{6}$$

Solution Example 2 of Substitution method II

Put value of n from Eq. 6 into Eq. 4

$$T(2^m) = 2T(2^{m/2}) + m \tag{7}$$

Rename the recurrence relation Eq. 7 as $S(m) = T(2^m)$ and $S(m/2) = T(2^{m/2})$

$$\Rightarrow S(m) = 2S(m/2) + m \tag{8}$$

Now we can guess the solution as

$$S(m) = O(m \lg m) \tag{9}$$

Put back the value of $S(m) = T(2^m)$ and m from Eq. 5 into Eq. 9

$$T(n) = O(\lg n \lg \lg n) \tag{10}$$

Solution Example 3 of Substitution method I

Example 3. Show that the solution of $T(n) = 2T(n/2) + n$ is $\Omega(n \lg n)$

Solution:

We have to prove that $T(n) \geq cn \lg n$ for some constant $c > 0$ and for all $n > n_0$.

$$T(n) = 2T(n/2) + n \tag{11}$$

$$T(n) = \Omega(n \lg n) \Rightarrow T(n) \geq cn \lg n \tag{12}$$

$\Rightarrow T(n) = \Omega(n \lg n)$ then it must be the solution for the input size $n/2$.

substitute the value of n as $n/2$ in Eq. 12.

Solution Example 3 of Substitution method II

$$T(n/2) \geq c(n/2)\lg(n/2) \tag{13}$$

Substitute the value of $T(n/2)$ from Eq. 13 into Eq. 11.

$$T(n) \geq 2(c(n/2)\lg(n/2)) + n$$

$$\Rightarrow T(n) \geq cn\lg(n/2) + n$$

$$\Rightarrow T(n) \geq cn\lg n - cn\lg 2 + n$$

$$\Rightarrow T(n) \geq cn\lg n - cn + n$$

$$\Rightarrow T(n) \geq cn\lg n + n(1-c)$$

$$\Rightarrow T(n) \geq cn\lg n \ \forall c \leq 1, n \geq 2$$

$$\therefore T(n) = \Omega(n\lg n)$$

Solution Example 4 of Substitution method I

Example 4. Show that the solution of $T(n) = T(n/2) + 1$ is $O(\lg n)$
Solution:
We have to prove that $T(n) \leq c \lg n$ for some constant $c > 0$ and for all $n > n_0$.

$$T(n) = T(n/2) + 1 \tag{14}$$

$$T(n) = O(\lg n) \Rightarrow T(n) \leq c \lg n \tag{15}$$

$\Rightarrow T(n) = O(\lg n)$ then it must be the solution for the input size $n/2$.
substitute the value of n as $n/2$ in Eq. 15.

Solution Example 4 of Substitution method II

$$T(n/2) \leq c \lg(n/2) \tag{16}$$

Substitute the value of $T(n/2)$ from Eq. 16 into Eq. 14.

$$T(n) \leq c \lg(n/2)) + 1$$

$$\Rightarrow T(n) \leq c \lg(n/2) + 1$$

$$\Rightarrow T(n) \leq c \lg n - c \lg 2 + 1$$

$$\Rightarrow T(n) \leq c \lg n - c + 1$$

$$\Rightarrow T(n) \leq c \lg n - (c - 1)$$

$$\Rightarrow T(n) \leq c \lg n \; \forall c \geq 1, n \geq 2$$

$$\therefore T(n) = O(\lg n)$$

Solution Example 5 of Substitution method I

Example 5. Find the solution of $T(n) = T(n-1) + n$.
Solution:
Guess the solution be $O(n^2)$. Now we have to prove that $T(n) \leq cn^2$ for some constant $c > 0$ and for all $n > n_0$.

$$T(n) = T(n-1) + n \tag{17}$$

$$T(n) = O(n^2) \Rightarrow T(n) \leq cn^2 \tag{18}$$

$\Rightarrow T(n) = O(n^2)$ then it must be the solution for the input size $n-1$.
substitute the value of n as $n-1$ in Eq. 18.

$$T(n-1) \leq c(n-1)^2 \tag{19}$$

Solution Example 5 of Substitution method II

Substitute the value of $T(n-1)$ from Eq. 19 into Eq. 17.

$$T(n) \leq c(n-1)^2 + n$$

$$\Rightarrow T(n) \leq c(n^2 - 2n + 1) + n$$

$$\Rightarrow T(n) \leq cn^2 - 2cn + c + n$$

$$\Rightarrow T(n) \leq cn^2 - (2c-1)n + c$$

Take value of c such that $(2c-1) \geq c$

$$\Rightarrow T(n) \leq cn^2 \ \forall c \geq 1, n \geq 1$$

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

Iteration method

- In iteration method, expend the right hand side of the recurrence relation.
- Try to find out the series and number of terms in the series.
- Find out the sum of the series

Example of Iteration method

1. Find the solution of $T(n) = 2T(n/2) + n$
2. Find the solution of $T(n) = 3T(n/4) + \Theta(n^2)$
3. Find the solution of $T(n) = T(n-1) + n^4$
4. Find the solution of $T(n) = T(\sqrt{n}) + n$

Example 1 for Iteration method I

Example 1 - Find the solution of $T(n) = 2T(n/2) + n$
Take

$$T(n) = 2T(n/2) + n \tag{20}$$

Find the value of $T(n/2) = 2T(n/4) + n/2$ from Eq. 20 and put into Eq. 20

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

$$T(n) = 4T(n/4) + 2n \tag{21}$$

Again find the value of $T(n/4) = 2T(n/8) + n/4$ from Eq. 20 and put into Eq. 21

$$\Rightarrow T(n) = 4(2T(n/8) + n/4) + 2n$$

Example 1 for Iteration method II

$$T(n) = 8T(n/8) + 3n \tag{22}$$

Repeating this for i times $T(n)$ will be

$$\Rightarrow T(n) = 2^i T(n/2^i) + in \tag{23}$$

$T(n/2^i)$ will terminate when $n/2^i = 1$ i.e $T(1)$

$$\Rightarrow n/2^i = 1$$

$$\Rightarrow n = 2^i$$

$$\Rightarrow \lg n = i$$

So after $\lg n$ iteration, the recurrence term will be terminated.
Eq. 23 will be

$$\Rightarrow T(n) = 2^{\lg n} T(1) + n \lg n$$

Example 1 for Iteration method III

$$\Rightarrow T(n) = nT(1) + n \lg n$$

$$\Rightarrow T(n) \leq cn + n \lg n, \forall c \geq T(1)$$

$$\Rightarrow T(n) \leq cn \lg n + n \lg n, \forall n \geq 2$$

$$\Rightarrow T(n) = (c+1)n \lg n$$

$$\Rightarrow T(n) \leq c_1 n \lg n, \forall c_1 \geq (c+1) > 0, \forall n \geq 2$$

$$\therefore T(n) = O(n \lg n)$$

Example 2 for Iteration method I

Example 2 - Find the solution of $T(n) = 3T(n/4) + \Theta(n^2)$
Take for $c > 0$

$$T(n) = 3T(n/4) + cn^2 \tag{24}$$

Find the value of $T(n/4) = 3T(n/16) + c(n/4)^2$ from Eq. 24 and put into Eq. 24

$$\Rightarrow T(n) = 3\left(3T(n/16) + c(n/4)^2\right) + cn^2$$

$$T(n) = 9T(n/16) + \left(1 + \frac{3}{16}\right)cn^2 \tag{25}$$

Example 2 for Iteration method II

Again find the value of $T(n/16) = 3T(n/64) + c(n/16)^2$ from Eq. 24 and put into Eq. 25

$$\Rightarrow T(n) = 9\left(3T(n/64) + c(n/16)^2\right) + \left(1 + \frac{3}{16}\right) cn^2$$

$$T(n) = 27T(n/64) + \left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2\right) cn^2 \tag{26}$$

Repeating this for i times $T(n)$ will be

$$\Rightarrow T(n) = 3^i T(n/4^i) + \left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \cdots + \left(\frac{3}{16}\right)^{(i-1)}\right) cn^2 \tag{27}$$

$T(n/4^i)$ will terminate when $n/4^i = 1$ i.e $T(1)$

Example 2 for Iteration method III

$\Rightarrow n/4^i = 1$

$\Rightarrow n = 4^i$

$\Rightarrow \log_4 n = i$

So after $\log_4 n$ iteration, the recurrence term will be terminated.
Eq. 27 will be

$$\Rightarrow T(n) = 3^{\log_4 n} T(1) + \left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \cdots + \left(\frac{3}{16}\right)^{(\log_4 n - 1)}\right) cn^2$$

$$\Rightarrow T(n) = n^{\log_4 3} T(1) + \left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \cdots + \left(\frac{3}{16}\right)^{(\log_4 n - 1)}\right)$$

Example 2 for Iteration method IV

$$\Rightarrow T(n) \leq c_1 n + \left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \ldots\right) cn^2, \forall c_1 \geq T(1)$$

$$\Rightarrow T(n) \leq c_1 n + \left(\frac{1}{1 - 3/16}\right) cn^2$$

$$\Rightarrow T(n) \leq c_1 n + \left(\frac{16}{13}\right) cn^2$$

$$\Rightarrow T(n) \leq c_1 n^2 + \left(\frac{16}{13}\right) cn^2, \ \forall n \geq 1$$

$$\Rightarrow T(n) \leq \left(\frac{13c_1 + 16c}{13}\right) n^2$$

Example 2 for Iteration method V

$$\Rightarrow T(n) \leq c_2 n^2 \text{ where } \forall c_2 \geq \frac{13c_1 + 16c}{13} > 0 \text{ and } \forall n \geq 1$$

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

Example 3 for Iteration method I

Example 3 - Find the solution of $T(n) = T(n-1) + n^4$

$$T(n) = T(n-1) + n^4 \tag{28}$$

Find the value of $T(n-1) = T(n-2) + (n-1)^4$ from Eq. 28 and put into Eq. 28

$$\Rightarrow T(n) = \left(T(n-2) + (n-1)^4\right) + n^4$$

$$T(n) = T(n-2) + (n-1)^4 + n^4 \tag{29}$$

Again find the value of $T(n-2) = T(n-3) + (n-2)^4$ from Eq. 28 and put into Eq. 29

$$\Rightarrow T(n) = \left(T(n-3) + (n-2)^4\right) + (n-1)^4 + n^4$$

Example 3 for Iteration method II

$$T(n) = T(n-3) + (n-2)^4 + (n-1)^4 + n^4 \quad (30)$$

Repeating this for i times $T(n)$ will be

$$\Rightarrow T(n) = T(n-i) + (n-i+1)^4 + \cdots + (n-1)^4 + n^4 \quad (31)$$

$T(n-i)$ will terminate when $n-i=0$ i.e $T(0)$

$$\Rightarrow n - i = 0$$

$$\Rightarrow i = n$$

So after n iteration, the recurrence term will be terminated. Eq. 31 will be

$$\Rightarrow T(n) = T(0) + 1^4 + 2^4 + 3^4 + \cdots + (n-1)^4 + n^4$$

Example 3 for Iteration method III

$$\Rightarrow T(n) \leq T(0) + n^4 + n^4 + \cdots + n^4, \forall n \geq 1$$

$$\Rightarrow T(n) \leq T(0) + nn^4$$

$$\Rightarrow T(n) \leq cn^5 + n^5, \forall c \geq T(0)$$

$$\Rightarrow T(n) \leq (c+1)n^5$$

$$\Rightarrow T(n) \leq c_1 n^5 \text{ where } \forall c_1 \geq (c+1) > 0 \text{ and } \forall n \geq 1$$

$$\therefore T(n) = O(n^5)$$

Example 4 for Iteration method I

Example 4 - Find the solution of $T(n) = T(\sqrt{n}) + n$

$$T(n) = T(\sqrt{n}) + n \tag{32}$$

Find the value of $T(\sqrt[2]{n}) = T(\sqrt[4]{n}) + \sqrt[2]{n}$ from Eq. 32 and put into Eq. 32

$$\Rightarrow T(n) = (T(\sqrt[4]{n}) + \sqrt[2]{n}) + n$$

$$T(n) = T(\sqrt[4]{n}) + \sqrt[2]{n} + n \tag{33}$$

Again find the value of $T(\sqrt[4]{n}) = T(\sqrt[8]{n}) + \sqrt[4]{n}$ from Eq. 32 and put into Eq. 33

$$\Rightarrow T(n) = (T(\sqrt[8]{n}) + \sqrt[4]{n}) + \sqrt[2]{n} + n$$

Example 4 for Iteration method II

$$T(n) = T(\sqrt[8]{n}) + \sqrt[4]{n} + \sqrt[2]{n} + n \tag{34}$$

Repeating this for i times $T(n)$ will be

$$\Rightarrow T(n) = T(\sqrt[2^i]{n}) + \sqrt[2^{i-1}]{n} + \cdots + \sqrt[4]{n} + \sqrt[2]{n} + n \tag{35}$$

$T(\sqrt[2^i]{n})$ will terminate when $\sqrt[2^i]{n} = 2$ i.e $T(2)$

$$\Rightarrow \sqrt[2^i]{n} = 2$$

$$\Rightarrow \lg(\sqrt[2^i]{n}) = \lg 2$$

$$\Rightarrow \frac{1}{2^i} \lg n = 1$$

$$\Rightarrow \lg n = 2^i$$

$$\Rightarrow \lg \lg n = i$$

Example 4 for Iteration method III

So after $\lg \lg n$ iteration, the recurrence term will be terminated. Eq. 35 will be

$$\Rightarrow T(n) = n + \sqrt[2]{n} + \sqrt[4]{n} + \cdots + \sqrt[2^{i-1}]{n} + T(2)$$

$$\Rightarrow T(n) \leq n + n + n + \ldots \quad \lg \lg n \text{ times}$$

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

$$\therefore T(n) = O(n \lg \lg n)$$

Recursion tree method

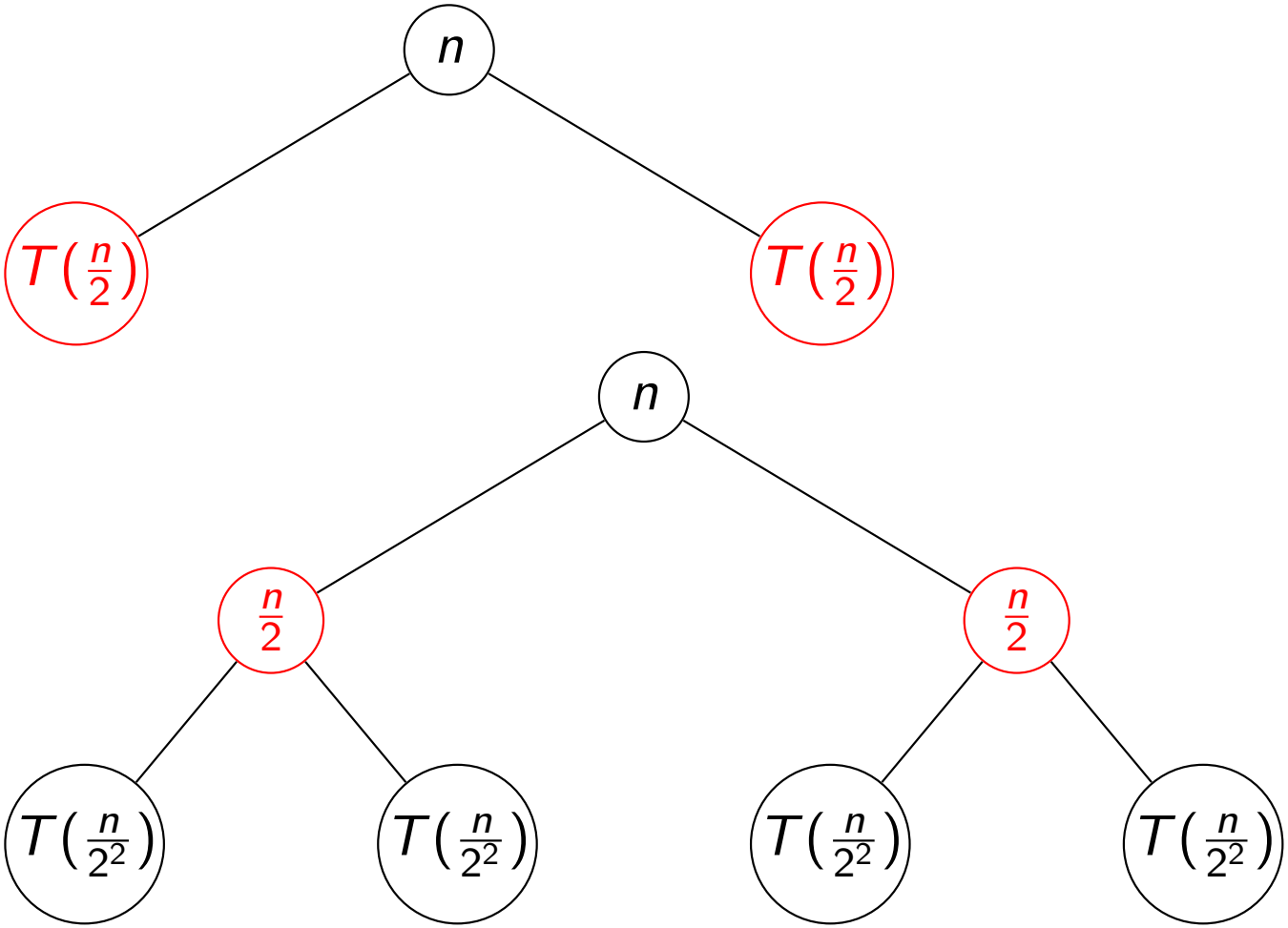
- Represent the problem as a tree taking non recurrence term on the root
- Expend the leaf node by using the recurrence relation
- Stop the process when input size reaches at a particular level
- Each node represent the cost of a sub problem
- Sum of all nodes of a level is level-cost
- Sum of all level is the cost of the problem

Examples of Recursion tree method

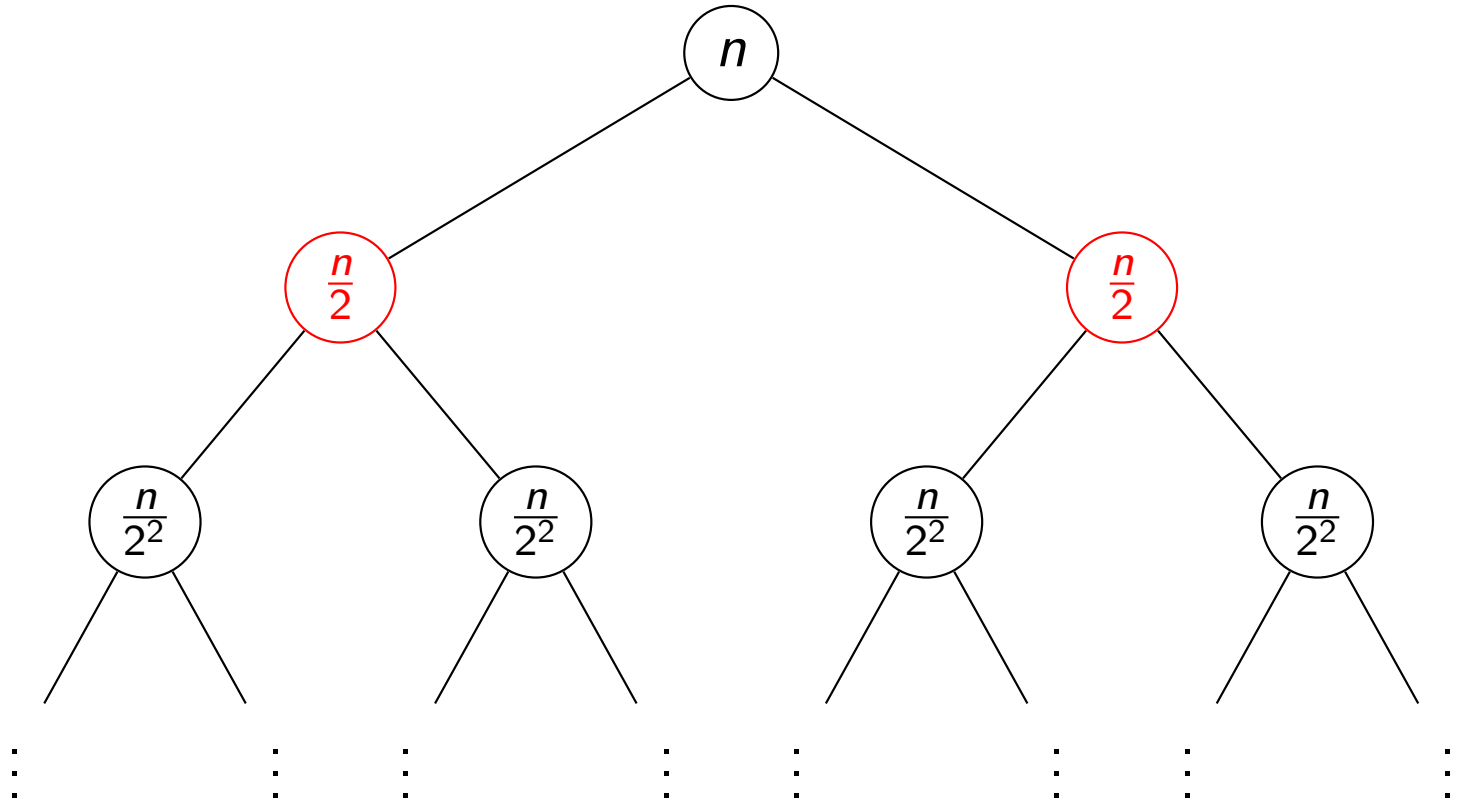
1. Find the solution of $T(n) = 2T(n/2) + n$
2. Find the solution of $T(n) = 3T(n/4) + cn^2$
3. Find the solution of $T(n) = T(n/2) + 1$
4. Find the upper and lower bound solution of $T(n) = T(n/3) + T(2n/3) + cn$
5. Determine a good asymptotic upper bound on the recurrence $T(n) = 3T(n/2) + n$.
6. Determine a good asymptotic upper bound on the recurrence $T(n) = T(n/2) + n^2$.
7. Determine a good asymptotic upper bound on the recurrence $T(n) = T(n-1) + T(n/2) + n$.
8. Give an asymptotically tight solution to the recurrence $T(n) = T(n-a) + T(a) + cn$, where $a \geq 1$ and $c > 0$ are constants.

Example 1 of Recursion tree method I

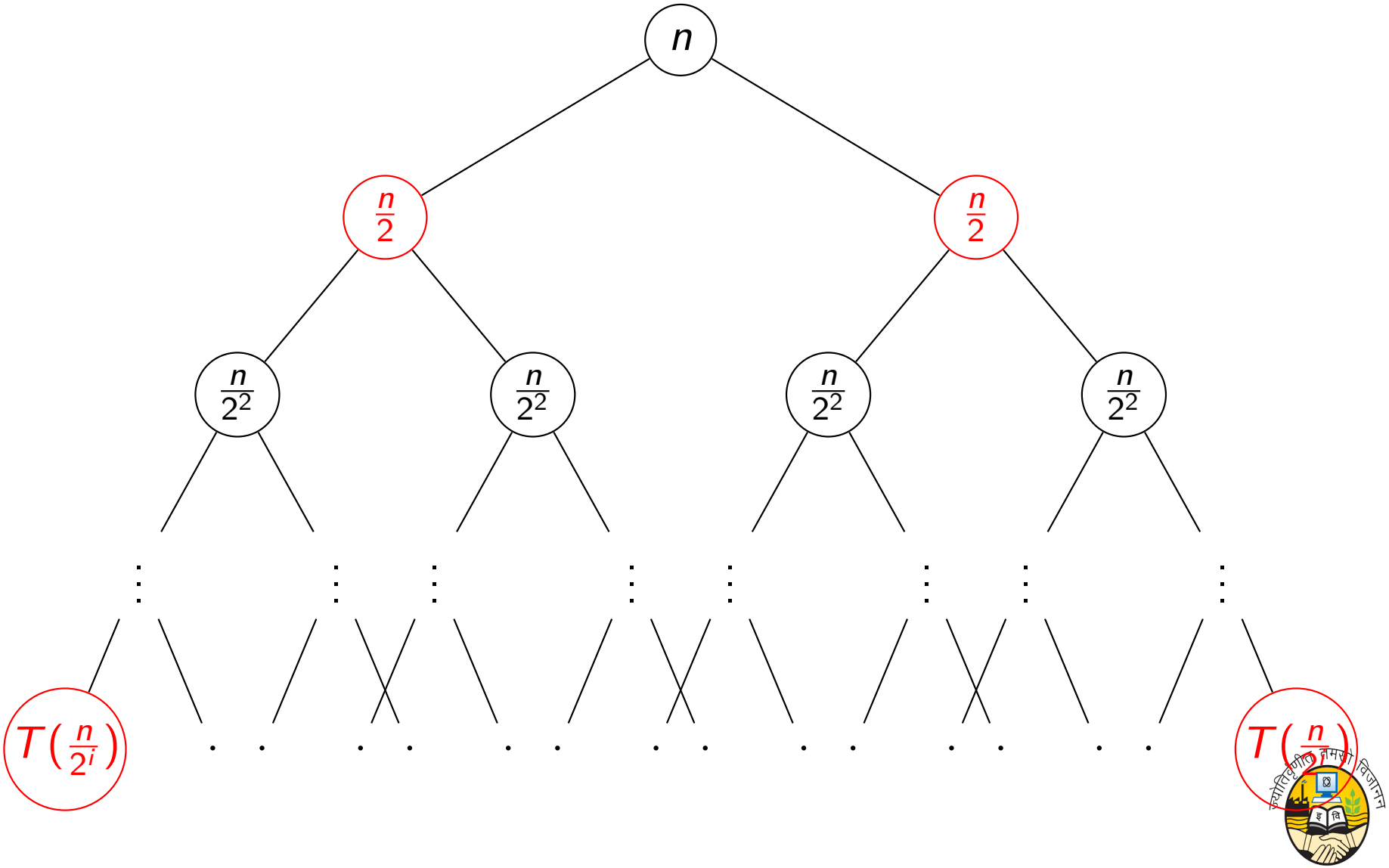
1. Find the solution of $T(n) = 2T(n/2) + n$



Example 1 of Recursion tree method II

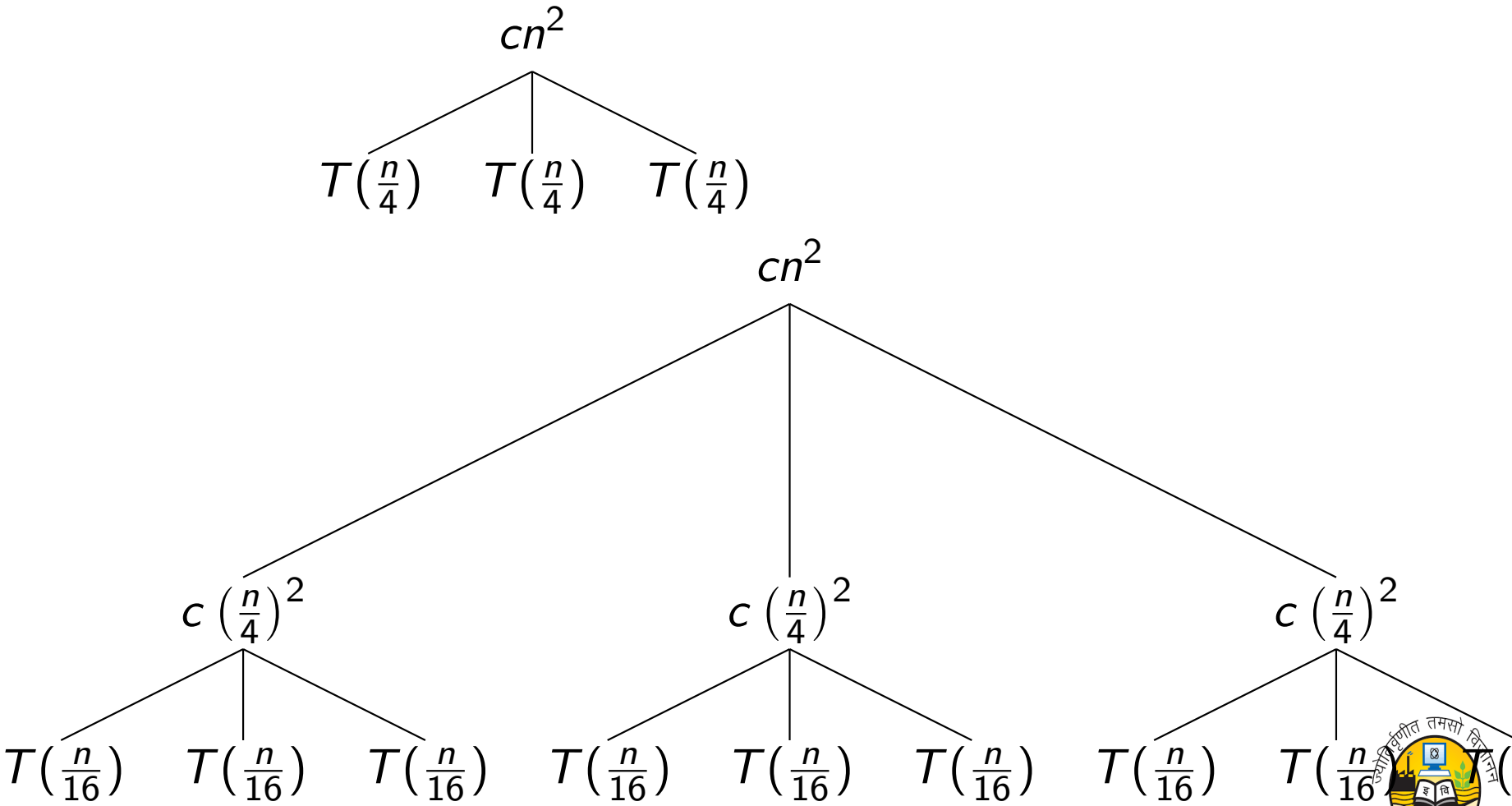


Example 1 of Recursion tree method III



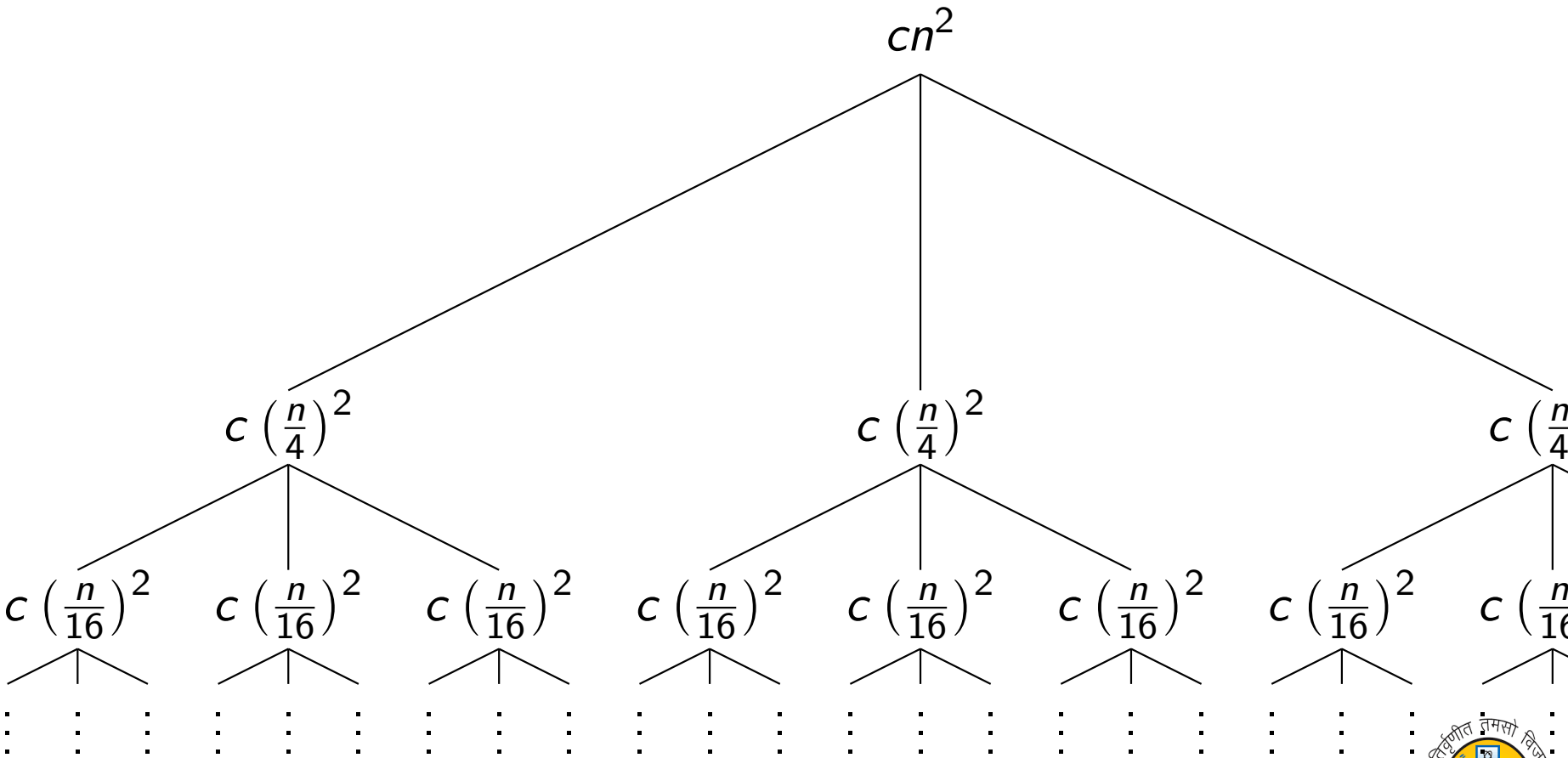
Example 2 of Recursion tree method I

2. Find the solution of $T(n) = 3T(n/4) + cn^2$



Example 2 of Recursion tree method I

2. Find the solution of $T(n) = 3T(n/4) + cn^2$



Master method (D&C)

Master method is based on the Master Theorem, which states as: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be an asymptotically positive function, and let $T(n)$ be defined on the non-negative integers by the recurrence

$$T(n) = aT(\frac{n}{b}) + f(n)$$

Then $T(n)$ has the following asymptotic bounds:

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

Examples of Master method (D&C)

1. Find the solution of $T(n) = 9T\left(\frac{n}{3}\right) + n$
2. Find the solution of $T(n) = T\left(\frac{2n}{3}\right) + 1$
3. Find the solution of $T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$
4. Find the solution of $T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$
5. Find the solution of $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$
6. Find the solution of $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
7. Find the solution of $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$
8. Find the solution of $T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^3)$

Solution of Examples 1 of Master method (D&C)

1. Find the solution of $T(n) = 9T\left(\frac{n}{3}\right) + n$

Comparing $T(n) = 9T\left(\frac{n}{3}\right) + n$ with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
$a = 9 \geq 1, b = 3 > 1$ and $f(n) = n$ so we can apply Master Method to solve this recursion.
As $n^{\log_b a} = n^{\log_3 9} = n^2$ is polynomially greater then $f(n) = n$.
so $n = O\left(n^{\log_3 9 - \epsilon}\right)$ where for any $0 < \epsilon \leq 1$.
Case 1 of Master Method will be applied and solution will be

$$T(n) = \Theta\left(n^{\log_3 9}\right) = \Theta\left(n^2\right)$$

Solution of Examples 2 of Master method (D&C)

2. Find the solution of $T(n) = T\left(\frac{2n}{3}\right) + 1$

Comparing $T(n) = T\left(\frac{2n}{3}\right) + n$ with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
$a = 1 \geq 1, b = \frac{3}{2} > 1$ and $f(n) = n^0 = 1$ so we can apply Master Method to solve this recursion.
As

$$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$$

is polynomially equal to $f(n) = 1$.
so $n^0 = \Theta\left(n^{\log_{\frac{3}{2}} 1}\right) = \Theta(n^0) = \Theta(1)$.
Case 2 of Master Method will be applied and solution will be

$$T(n) = \Theta\left(n^{\log_{\frac{3}{2}} 1} \lg n\right) = \Theta\left(n^0 \lg n\right) = \Theta(\lg n)$$

Solution of Examples 3 of Master method (D&C) I

3. Find the solution of $T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$

Comparing $T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$ with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ $a = 3 \geq 1, b = 4 > 1$ and $f(n) = n \lg n$ so we can apply Master Method to solve this recursion.
As $n^{\log_b a} = n^{\log_4 3} = n^{0.79}$ is polynomially less than $f(n) = n \lg n$.
so $n \lg n = \Omega(n^{(\log_4 3 + \epsilon)}) = n^{(0.79+\epsilon)}$ for $\epsilon = 0.2$
Also

$$3f(n/4) = 3\frac{n}{4} \lg \left(\frac{n}{4}\right)$$

$$\Rightarrow \frac{3}{4} n \lg \left(\frac{n}{4}\right)$$

$$\Rightarrow \frac{3}{4} n \lg n - \frac{3}{4} n \lg 4$$

Solution of Examples 3 of Master method (D&C) II

$$\Rightarrow \frac{3}{4} n \lg n - \frac{3}{2} n$$

$$\Rightarrow \frac{3}{4} n \lg n - \frac{3}{2} n \leq cn \lg n$$

where $\frac{3}{4} \leq c < 1$.
Case 3 of Master Method will be applied and solution will be

$$T(n) = \Theta\left(f(n)\right) = \Theta\left(n \lg n\right)$$

Solution of Examples 4 of Master method (D&C)

4. Find the solution of $T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$

Comparing $T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$ with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ $a = 2 \geq 1, b = 2 > 1$ and $f(n) = n \lg n$ so we can apply Master Method to solve this recursion.
As $n^{\log_b a} = n^{\log_2 2} = n^1$ is not polynomially less than, greater than or equal to $f(n) = n \lg n$. It is asymptotically less than $n \lg n$. It lies between the gaps of Case 2 and Case 3.
So it cann't be solved using Master Method.

Solution of Examples 5 of Master method (D&C)

5. Find the solution of $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$

Comparing $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$ with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
$a = 2 \geq 1, b = 2 > 1$ and $f(n) = \frac{n}{\lg n}$ so we can apply Master Method to solve this recursion.
As $n^{\log_b a} = n^{\log_2 2} = n^1$ is not polynomially less than, greater than or equal to $f(n) = \frac{n}{\lg n}$. It is asymptotically greater than $\frac{n}{\lg n}$. It lies between the gaps of Case 1 and Case 2.
So it cann't be solved using Master Method.

Solution of Examples 6 of Master method (D&C)

6. Find the solution of $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

Comparing $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ $a = 2 \geq 1, b = 2 > 1$ and $f(n) = \Theta(n) = cn$ so we can apply Master Method to solve this recursion.
As

$$n^{\log_b a} = n^{\log_2 2} = n^1$$

is polynomially equal to $f(n) = cn$.
so $cn = \Theta(n)$.
Case 2 of Master Method will be applied and solution will be

$$T(n) = \Theta\left(n^1 \lg n\right) = \Theta(n \lg n)$$

Solution of Examples 7 of Master method (D&C)

7. Find the solution of $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$

Comparing $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$ with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
$a = 8 \geq 1, b = 2 > 1$ and $f(n) = \Theta(n^2) = cn^2$ so we can apply Master Method to solve this recursion.
As

$$n^{\log_b a} = n^{\log_2 8} = n^3$$

is polynomially greater than $f(n) = cn^2$.
so $cn^2 = O(n^{3-\epsilon})$ for $0 < \epsilon \leq 1$
Case 1 of Master Method will be applied and solution will be

$$T(n) = \Theta\left(n^3\right)$$

Solution of Examples 8 of Master method (D&C) I

8. Find the solution of $T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^3)$

Comparing $T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^3)$ with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ $a = 4 \geq 1, b = 2 > 1$ and $f(n) = \Theta(n^3) = dn^3, d > 0$ so we can apply Master Method to solve this recursion.
As

$$n^{\log_2 4} = n^2$$

is polynomially less than $f(n) = dn^3$.
Also $af\left(\frac{n}{b}\right) \leq cf(n)$ for $0 < c < 1$

$$\Rightarrow 4d\left(\frac{n}{2}\right)^3$$

$$\Rightarrow \frac{4}{8}dn^3$$

Solution of Examples 8 of Master method (D&C) II

$$\Rightarrow \frac{1}{2} dn^3 \leq cdn^3$$

Where $\frac{1}{2} \leq c < 1, \Rightarrow c = 0.75$
so $dn^3 = \Omega(n^{2+\epsilon})$ for $0 < \epsilon \leq 1$
Case 3 of Master Method will be applied and solution will be

$$T(n) = \Theta\left(n^3\right)$$

Muster Theorem (S&C) I

Let $a > 0$ and $b > 0$ be constants, let $f(n)$ be an asymptotically positive function such that $f(n) = O(n^d)$ for $d \geq 0$ and let $T(n)$ be defined on the positive integers by the recurrence

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

Then $T(n)$ has the following asymptotic bounds:

1. if $a < 1$ then $T(n) = O(n^d)$
2. if $a = 1$ then $T(n) = O(n^{d+1})$
3. if if $a > 1$ then $T(n) = O(a^{\frac{n}{b}} n^d)$

Muster Theorem (S&C) II

Proof:

Expend the left hand side term of given recurrence

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

$$\Rightarrow T(n) = a^2 T(n-2b) + af(n-b) + f(n)$$

$$\Rightarrow T(n) = a^3 T(n-3b) + a^2 f(n-2b) + af(n-b) + f(n)$$

$$\Rightarrow T(n) = a^i T(n-ib) + a^{i-1} f(n-(i-1)b) + \cdots + af(n-b) + f(n)$$

This will be terminated when $n - ib = 0$ that is $i = \frac{n}{b}$

$$\Rightarrow T(n) = a^{\frac{n}{b}} T(0) + a^{\frac{n}{b}-1} f\left(n - \left(\frac{n}{b} - 1\right) b\right) + \cdots + af(n-b) + f(n)$$

$$\Rightarrow T(n) = \sum_{i=0}^{\frac{n}{b}-1} a^i f(n-ib) + a^{\frac{n}{b}} T(0)$$

Muster Theorem (S&C) III

$$\Rightarrow T(n) \leq \sum_{i=0}^{\frac{n}{b}} a^i f(n - ib)$$

$$\Rightarrow T(n) = O\left(n^d \sum_{i=0}^{\frac{n}{b}} a^i\right)$$

Now there are 3 cases:

1. if $a < 1$ then

$$\sum_{i=0}^{\frac{n}{b}} a^i = \frac{1 - a^{\frac{n}{b}+1}}{1 - a} \leq \frac{1}{1 - a} = O(1)$$

$$\Rightarrow T(n) = O\left(n^d\right) O(1)$$

$$\Rightarrow T(n) = O\left(n^d\right)$$

Muster Theorem (S&C) IV

2. if $a = 1$ then

$$\sum_{i=0}^{\frac{n}{b}} a^i = \sum_{i=0}^{\frac{n}{b}} 1 = \frac{n}{b} + 1 = O(n)$$

$$\Rightarrow T(n) = O\left(n^d\right) O(n)$$

$$\Rightarrow T(n) = O\left(n^{d+1}\right)$$

Muster Theorem (S&C) V

3. if $a > 1$ then

$$\sum_{i=0}^{\frac{n}{b}} a^i = \frac{a^{\frac{n}{b}+1} - 1}{a - 1} = O(a^{\frac{n}{b}})$$

$$\Rightarrow T(n) = O\left(n^d\right) O(a^{\frac{n}{b}})$$

$$\Rightarrow T(n) = O\left(n^d a^{\frac{n}{b}}\right)$$

so

$$T(n) = \begin{cases} O(n^d) & a < 1 \\ O(n^{d+1}) & a = 1 \\ O(a^{\frac{n}{b}} n^d) & a > 1 \end{cases}$$

Examples of Muster Theorem (S&C)

1. Find the solution of $T(n) = \frac{1}{2}T(n-1) + n$
2. Find the solution of $T(n) = \frac{7}{8}T(n-1) + n^3$
3. Find the solution of $T(n) = T(n-2) + n$
4. Find the solution of $T(n) = T(n-2) + n \lg n$
5. Find the solution of $T(n) = T(n-2) + \frac{n}{\lg n}$
6. Find the solution of $T(n) = 2T(n-2) + n$
7. Find the solution of $T(n) = 3T(n-1) + n^2$

Solutions of Examples of Muster Theorem (S&C) I

1. Find the solution of $T(n) = \frac{1}{2}T(n-1) + n$
   Here $a = \frac{1}{2}, b = 1, f(n) = O(n), d = 1,$
   Case 1 applies. So the solution is :

   $$T(n) = O\left(n^d\right)$$

   $$T(n) = O\left(n\right)$$

2. Find the solution of $T(n) = \frac{7}{8}T(n-1) + n^3$
   Here $a = \frac{7}{8}, b = 1, f(n) = O(n^3), d = 3,$
   Case 1 applies. So the solution is :

   $$T(n) = O\left(n^d\right)$$

   $$T(n) = O\left(n^3\right)$$

Solutions of Examples of Muster Theorem (S&C) II

3. Find the solution of $T(n) = T(n-2) + n$
   Here $a = 1, b = 2, f(n) = O(n), d = 1$,
   Case 2 applies. So the solution is :

$$T(n) = O\left(n^{d+1}\right)$$

$$T(n) = O\left(n^{2}\right)$$

4. Find the solution of $T(n) = T(n-2) + n \lg n$
   Here $f(n) = n \lg n$, function is not polynomial of n so method cann't be applied.
   $a = 1, b = 2, f(n) = O(n^2), d = 2$,
   Case 2 applies. So the solution is :

$$T(n) = O\left(n^{d+1}\right)$$

$$T(n) = O\left(n^{3}\right)$$

Solutions of Examples of Muster Theorem (S&C) III

5. Find the solution of $T(n) = T(n-2) + \frac{n}{\lg n}$
   Here $f(n) = \frac{n}{\lg n}$, function is not polynomial of n so method cann't be applied.
   $a = 1, b = 2, f(n) = O(n), d = 1,$
   Case 2 applies. So the solution is :

   $$T(n) = O\left(n^{d+1}\right)$$

   $$T(n) = O\left(n^2\right)$$

6. Find the solution of $T(n) = 2T(n-2) + n$
   Here $a = 2, b = 2, f(n) = O(n), d = 1,$
   Case 3 applies. So the solution is :

   $$T(n) = O\left(n^d a^{\frac{n}{b}}\right)$$

   $$T(n) = O\left(n^1 2^{\frac{n}{2}}\right)$$

Solutions of Examples of Muster Theorem (S&C) IV

7. Find the solution of $T(n) = 3T(n-1) + n^2$
   Here $a = 3, b = 1, f(n) = O(n^2), d = 2,$
   Case 3 applies. So the solution is :

$$T(n) = O\left(n^d a^{\frac{n}{b}}\right)$$

$$T(n) = O\left(n^2 3^n\right)$$

Complexity analysis I

Analyzing an algorithm means predicting the resources that the algorithm requires. Resources may be memory, communication bandwidth, computer hardware or CPU time. Our primary concern is to measures the computational time required for the algorithm.
Running time:-The running time of an algorithm is the number of primitive operations or steps executed on a particular input.
Why do we normally concentrate on finding only the worst-case running time?

1. The worst-case running time of an algorithm gives us an upper bound on the running time for any input. So it guarantees that the algorithm will never slower than this.
2. In real applications, worst case normally occurs for example searching a non existing data.

Complexity analysis II

3. Best case is like an ideal case which guarantees that the algorithm will never faster than stated. Based upon this we can't allocate the resources.
4. Average case normally perform as worst case because normally we take average case as average of best and worst or best for half size input and worst for other half size.

Complexity analysis: Insertion sort I

Insertion-Sort(A,N)	Cost	Times
1. for $j = 2$ to N	c_1	n
2. $key = A[j]$	c_2	n-1
Insert $A[j]$ in sorted $A[1]$ to $A[j-1]$		
3. $i = j - 1$	c_3	n-1
4. while $i > 0$ and $A[i] > key$	c_4	$\sum_{j=2}^{n} t_j$
5. $A[i+1] = A[i]$	c_5	$\sum_{j=2}^{n}(t_j - 1)$
6. $i = i - 1$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
while-end		
7. $A[i+1] = key$	c_7	n-1
for-end		

Complexity analysis: Insertion sort II

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j - 1) + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7(n-1)$$

$$\Rightarrow T(n) = an + b + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j - 1) + c_6 \sum_{j=2}^{n}(t_j - 1)$$

Now consider different cases:

Complexity analysis: Insertion sort III

1. **Best Case:** The algorithm performs best if $key \leq A[i]$ for every value of j in step 4.
   Then it executes only once for each value of j and total of n-1 times.
   Step 5 and 6 will not be execute at all.
   This is the case when array is already sorted

   $$T(n) = an + b = O(n)$$

2. **Worst Case:** The algorithm performs worst if $key > A[i]$ for each value of j and stops only when $i < 1$ in step 4.
   Then it will execute always j times for each value of $j = 2, 3, \ldots, n$

Complexity analysis: Insertion sort IV

so

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

and step 5 and 6 will execute

$$\sum_{j=2}^{n} (j-1) = \frac{(n-1)n}{2}$$

This is the case when array is already sorted in reverse order

$$T(n) = an^2 + bn + c = O(n^2)$$

Complexity analysis: Merge Sort I

MERGE-SORT(A, p, r)

1 if $p < r$
    2 $q = (p + r)/2$
    3 MERGE-SORT(A, p, q)
    4 MERGE-SORT(A, q+1, r)
    5 MERGE(A,p,q,r)

MERGE(A, p, q, r)

6 $n_1 = q - p + 1$

7 $n_2 = r - q$

8 Let $L[1..n_1]$ and $R[1..n_2]$ be new arrays

9 for $i = 1$ to n_1
    10 $L[i] = A[p + i - 1]$

11 for $j = 1$ to n_2
    12 $R[j] = A[q + j]$

13 $i = 1$

Complexity analysis: Merge Sort II

14 $j = 1$

15 for $k = p$ to r

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

    17 $A[k] = L[i]$

    18 $i = i + 1$

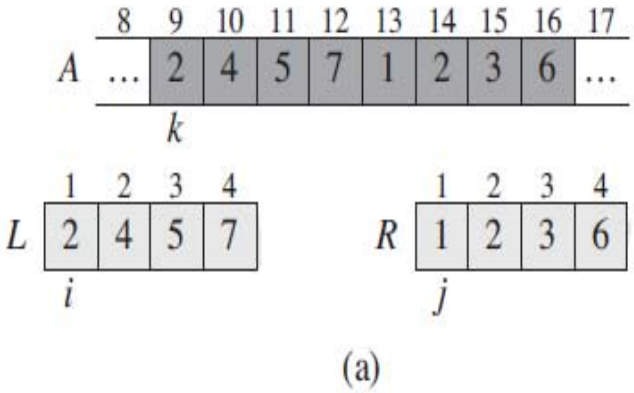
  19 else

    20 $A[k] = R[j]$

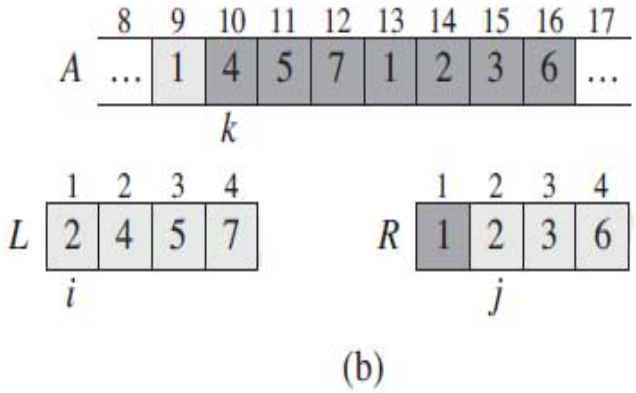
    21 $j = j + 1$

Complexity analysis: Merge Sort III

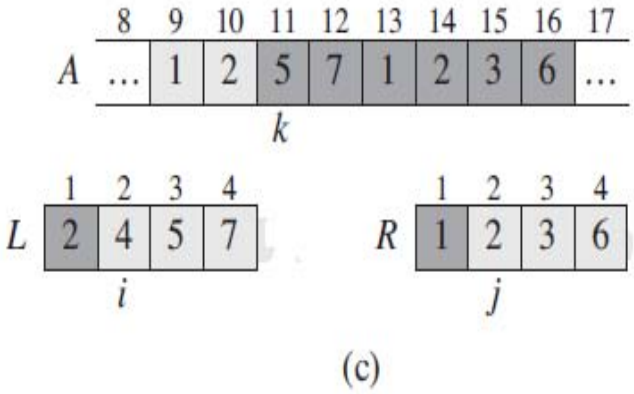
Working of Merge



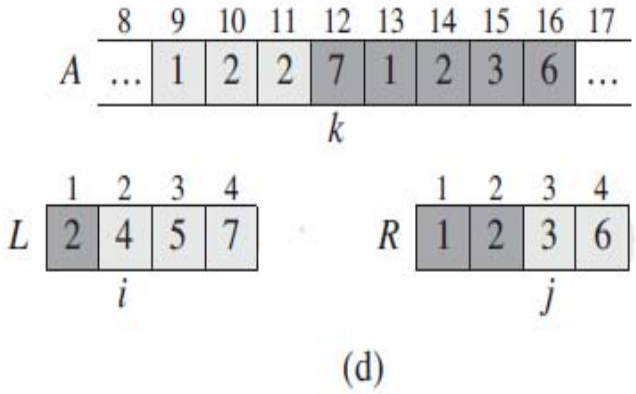
(a)



(b)



(c)



(d)

Analysis:

Complexity analysis: Merge Sort IV

- First call of merge-sort will be MERGE-SORT(A, 1, n) for input size n.
- q is half of n,$q = \frac{n}{2}$ in step 2.
- Every call of merge-sort divides the size in half and double the sub-problems.
- So there will be only lg n calls of merge-sort and sum of size of all sub-problems is n
- There is only one call for each merge-sort call
- So total calls of the merge will be lg n
- Step 6 to 8 will be executed lg n times
- Step 9 will be executed n_1 lg n times
- Step 10 will be executed n_1 lg n times
- Step 11 will be executed n_2 lg n times
- Step 12 will be executed n_2 lg n times

Complexity analysis: Merge Sort V

- Total of Step 9 to 12 will be executed $2(n_1 + n_2) \lg n$ times
- Step 13 & 14 will be executed $\lg n$ times
- For each call of $\lg n$ step 15 will be executed $n \lg n$ times
- Step 16 will be executed $n \lg n$ times
- Every time either step 17 & 18 or step 20 & 21 will be executed and sum of these will be $n \lg n$.
- After adding cost of all these step
  $T(n) = an \lg n + bn + c = O(n \lg n)$

Complexity analysis: Quick Sort I

QUICK-SORT(A, p, r)

1 if $p < r$
    2 q=PARTITION(A,p,r)
    3 QUICK-SORT(A, p, q-1)
    4 QUICK-SORT(A, q+1, r)

PARTITION(A, p, r)

5 $x = A[r]$

6 $i = p$

7 for $j = p$ to $r - 1$
    8 if $A[j] \leq x$
        9 if$(i \neq j)$ swap$(A[i], A[j])$
        10 $i = i + 1$

11 swap$(A[i], A[r])$

12 return i

Analysis:

Complexity analysis: Quick Sort II

- The number of calls of PARTITION depends on QUICK-SORT
- The number of calls of QUICK-SORT depends on q
- If PARTITION returns q as middle value for every call
- Then array will be divided in two half every time
- So after $\lg n$ times, $p = r$ and process will terminate.
- This will be **Best Case** of the algorithm.
- $\therefore T(n) = O(n \lg n)$ for best case
- But if PARTITION find $A[r]$ fit at its current location after checking p to r-1 elements then it return q as an end position and making partition of size 0 and $n - 1$.
- So every time the array size is reduced by only 1.
- The process will terminate after n operations.

Complexity analysis: Quick Sort III

- This will be **Worst Case** of the algorithm and happens when input is already sorted.
- Step 7 will be executed n times for every value of q in step 2.
- $\therefore T(n) = O(n^2)$ for worst case

Randomized Quick Sort I

- To avoid the worst performance of the Quick Sort for sorted input, we use Randomized Quick Sort.
- In this we choose pivot element randomly.

Algorithm:

RANDOMIZED-QUICK-SORT(A, p, r)

1 if $p < r$
  2 q=RANDOMIZED-PARTITION(A,p,r)
  3 RANDOMIZED-QUICK-SORT(A, p, q-1)
  4 RANDOMIZED-QUICK-SORT(A, q+1, r)

RANDOMIZED-PARTITION(A, p, r)

5 $i = Random(p, r)$
6 $swap(A[i], A[r])$
7 $x = A[r]$
8 $i = p$

Randomized Quick Sort II

```
9  for j = p to r − 1
     10  if A[j] ≤ x
           11  if(i ≠ j) swap(A[i], A[j])
           12  i = i + 1
13 swap(A[i], A[r])
14 return i
```

This algorithm performs worst case only when $Random(p, r)$ always gives the location of the largest/smallest number which is very rare.

Strassen's algorithm for Matrix Multiplications I

- if we multiply two matrices $C_{n\times m} = A_{n\times l}B_{l\times m}$
- Total number of scaler multiplications will be $n \times l \times m$
- If we take matrices of size $n \times n$ where n is power of 2 i.e $n = 2^i$
- Total number of scaler multiplications will be n^3
- If we divide size by 2 then the matrices will be

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Strassen's algorithm for Matrix Multiplications II

- $$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$
- $$C_{11} = A_{11}.B_{11} + A_{12}.B_{21}$$
  $$C_{12} = A_{11}.B_{12} + A_{12}.B_{22}$$
  $$C_{21} = A_{21}.B_{11} + A_{22}.B_{21}$$
  $$C_{22} = A_{21}.B_{12} + A_{22}.B_{22}$$
- So there are 8 multiplication and 4 submissions of size $n/2$

Strassen's algorithm for Matrix Multiplications III

- $$T(n) = 8T\left(\frac{n}{2}\right) + 4(n/2)^2$$
- Solution of the recurrence will be $\Theta(n^3)$ using master theorem. So no benefit of dividing the main problem in subproblems.
- Strassen's gives 18 sum and 7 products of size $n/2$ as follows:

$$S_1 = B_{12} - B_{22}$$

$$S_2 = A_{11} + A_{12}$$

$$S_3 = A_{21} + A_{22}$$

$$S_4 = B_{21} - B_{11}$$

$$S_5 = A_{11} + A_{22}$$

Strassen's algorithm for Matrix Multiplications IV

$$S_6 = B_{11} + B_{22}$$

$$S_7 = A_{12} - A_{22}$$

$$S_8 = B_{21} + B_{22}$$

$$S_9 = A_{11} - A_{21}$$

$$S_{10} = B_{11} + B_{12}$$

- 7 products will be

$$P_1 = A_{11}.S_1$$

$$P_2 = S_2.B_{22}$$

$$P_3 = S_3.B_{11}$$

$$P_4 = A_{22}.S_4$$

Strassen's algorithm for Matrix Multiplications V

$$P_5 = S_5.S_6$$

$$P_6 = S_7.S_8$$

$$P_7 = S_9.S_{10}$$

- Now the the matrix will be

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

$$C_{12} = P_1 + P_2$$

$$C_{21} = P_3 + P_4$$

$$C_{22} = P_5 + P_1 - P_3 - P_7$$

- There are 7 multiplication and 18 submissions of size $n/2$

Strassen's algorithm for Matrix Multiplications VI

- $$T(n) = 7T\left(\frac{n}{2}\right) + 18(n/2)^2$$
- The solution of the recurrence will be $\Theta(n^{\lg 7})$ using master theorem which is less then $\Theta(n^3)$
- So Strassen's algorithm is faster then divide and conquer.
- Ques.1 Find the product of the following matrices using Strassen's algorithm.

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$$

Medians and Order Statistics

- The i^{th} **Order Statistic** of a set of n elements is the i^{th} smallest element.
- The 1^{st} **Order Statistic** is the **minimum** of the set.
- The n^{th} **Order Statistic** is the **maximum** of the set.
- **Median** is the middle element of the set.
- If n is odd then unique median will be at $(n+1)/2$
- If n is even then two median **lower** and **upper** will be at $n/2$ and $n/2+1$ respectively.
- So **lower median** will be at $\lfloor (n+1)/2 \rfloor$ and **upper median** will be at $\lceil (n+1)/2 \rceil$ for both n even or odd.

Minimum and Maximum I

MINIMUM(A,n)

1 $min = A[1]$
2 $Loc = 1$
3 for $i = 2$ to n
4 if $min > A[i]$
5 $min = A[i]$
6 $Loc = i$
7 return Loc

Number of comparison is $n - 1$

Minimum and Maximum II

Min-Max(A,n)

1. $min = A[1]$
2. $MinLoc = 1$
3. $max = A[1]$
4. $MaxLoc = 1$
5. for $i = 2$ to n
6. if $min > A[i]$
7. $min = A[i]$
8. $MinLoc = i$
9. if $max < A[i]$
10. $max = A[i]$
11. $MaxLoc = i$
12. return $MinLoc, MaxLoc$

Number of comparison is $2(n-1)$

We can reduce this to $3\left\lfloor \frac{n}{2} \right\rfloor$

Minimum and Maximum III

Min-Max(A,n)

1. if $n \leq 0$ return
2. if $n\%2 = 0$ and $n \geq 2$
3. if $A[1] < A[2]$
4. $Min = A[1]$
5. $MinLoc = 1$
6. $Max = A[2]$
7. $MaxLoc = 2$
8. else
9. $Min = A[2]$
10. $MinLoc = 2$
11. $Max = A[1]$
12. $MaxLoc = 1$

Minimum and Maximum IV

13. for $i = 3$ to n with step size 2
14. if $A[i] < A[i+1]$
15. if $A[i] < min$
16. $min = A[i]$
17. $MinLoc = i$
18. if $A[i+1] > max$
19. $max = A[i+1]$
20. $MaxLoc = i+1$
21. else
22. if $A[i+1] < min$
23. $min = A[i+1]$
24. $MinLoc = i+1$
25. if $A[i] > max$

Minimum and Maximum V

26. $max = A[i]$
27. $MaxLoc = i$
28. else
29. $Min = A[1]$
30. $MinLoc = 1$
31. $Max = A[1]$
32. $MaxLoc = 1$
33. for $i = 2$ to n with step size 2
34. if $A[i] < A[i + 1]$
35. if $A[i] < min$
36. $min = A[i]$
37. $MinLoc = i$
38. if $A[i + 1] > max$

Minimum and Maximum VI

39. $max = A[i+1]$
40. $MaxLoc = i+1$
41. else
42. if $A[i+1] < min$
43. $min = A[i+1]$
44. $MinLoc = i+1$
45. if $A[i] > max$
46. $max = A[i]$
47. $MaxLoc = i$
48. return $MaxLoc, MinLoc$

Number of comparison is:

$$\begin{cases} 3\frac{n-1}{2} & \text{if n is odd} \\ 3\frac{n-2}{2}+1 & \text{if n is even} \end{cases}$$

or we can say max number of comparison is $3\left\lfloor \frac{n}{2} \right\rfloor$

Minimum and Maximum VII

RANDOMIZED-SELECT(A,p,r,i)

1. if $p == r$
2. return $A[p]$
3. $q = RANDOMIZED - PARTITION(A, p, r)$
4. $k = q - p + 1$
5. if $i == k$
6. return $A[q]$
7. elseif $i < k$
8. return $RANDOMIZED - SELECT(A, p, q - 1, i)$
9. else
10. return $RANDOMIZED - SELECT(A, q + 1, r, i - k)$

Dynamic programming

- Solves problems by combining the solutions to sub-problems
- Sub-problems are overlapping
- Doesn't solve overlapping sub-problems again and again
- Behave like Divide and Conquer if sub-problems are not overlapping
- Used in optimization problems
- Can be used either top-down with memoization or bottom-up method

Four Steps of Dynamic programming

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution
4. Construct an optimal solution from computed information

Elements of dynamic programming

Two key ingredients that an optimization problem must have in order to apply dynamic programming:

1. Optimal substructure
2. Overlapping sub-problems

Memoization

- Memoization is mainly storing the value in the form of Memo or in some tabular method
- Whenever we need some calculation of the sub-problem then we first check the memo
- If solution of the sub-problems is already available in memo then we use that solution and not solve the sub-problem again
- If the solution of the sub-problem is not in memo then we solve the sub-problem and note the result in the form of the memo for next call

Difference between Dynamic Programming and Memoization I

- Memoization is a term describing an optimization technique where we cache previously computed results, and return the cached result when the same computation is needed again.
- Dynamic programming is a technique for solving problems of recursive nature, iteratively and is applicable when the computations of the subproblems overlap.
- Dynamic programming is typically implemented using tabulation, but can also be implemented using memoization. So as we can see, neither one is a "subset" of the other.
- When we solve a dynamic programming problem using tabulation we solve the problem bottom-up i.e. by solving all related sub-problems first, typically by filling up an n-dimensional table. Based on the results in the table, the solution to the top / original problem is then computed.

Difference between Dynamic Programming and Memoization II

- If we use memoization to solve the problem we do it by maintaining a map of already solved sub problems. We do it top-down in the sense that we solve the top problem first (which typically recurses down to solve the sub-problems).

Matrix Chain Multiplication(MCM)

- Matrix chain multiplication is not the multiplication of the matrices but it is the way to find the order of matrix multiplication with minimum number of scalar multiplication.
- It is mainly fully parenthesization of the matrices
- ABC can be multiplied by two way : $A(BC)$ or $(AB)C$.
- Which one cost the minimum number of scalar multiplication can be found using MCM
- Find the order of multiplication of the matrices $A_1 A_2 \ldots A_n$
- We need an array p of size $n+1$ to store the dimensions of n compatible matrices

Step 1: The structure of an optimal parenthesization I

- Let $A_{i\ldots j} = A_i \ldots A_j$ is the product of matrices from $A_i, A_{i+1}, \ldots A_j$ for $i \leq j$
- if $i = j$ then there is only one matrix and number of scalar multiplication is zero.
- if $i < j$ then we can put parenthesis anywhere after A_i but before A_j
- Suppose we use parenthesis after matrix A_k i.e. $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$ where $i \leq k < j$ for optimal solutions
- Now the total cost of the $A_{i\ldots j}$ will be cost of $A_{i\ldots k}$ plus cost of $A_{k+1\ldots j}$ plus the cost of multiplying them together.
- We supposed k is at optimal position so no other value of k is less costly.
- So $A_{i\ldots k}$ and $A_{k+1\ldots j}$ is also optimal.

Step 1: The structure of an optimal parenthesization II

- We can find out the optimal solution of the the problem from optimal solution of the the sub-problem
- We have to take the correct value of $k, i \leq k < j$ such that the sub-problems having the optimal solutions.
- We have to examine all possible value of k for the optimal solution.

Step 2: A recursive solution I

We have to define cost of an optimal solution recursively in terms of the optimal solutions to sub-problems.

- Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i \ldots j}$ where $1 \leq i \leq j \leq n$
- The lowest cost to multiply $A_{1 \ldots n}$ will be $m[1, n]$.
- If $i = j$ that is there is only one matrix then no need to multiply. The $m[i, i] = 0$ for all $1 \leq i \leq n$.
- If $i < j$ then we split the matrix at position k for optimal solution.
- The total cost will be $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} \times p_k \times p_j$
- For all possible values of $k = i, i + 1, \ldots, j - 1$ we have to find out $m[i, j]$
- We pick that value of k for which $m[i, j]$ is minimum.

Step 2: A recursive solution II

- $\therefore$ Recursive definition for the minimum cost of parenthesizing the product $A_{i\ldots j} = A_i A_{i+1} \ldots A_j$ will be:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} \times p_k \times p_j\} & \text{if } i < j \end{cases}$$

- Store the value of k in $s[i,j]$ for which $m[i,j]$ is minimum.

Step 3: Computing the optimal costs I

- We can find the solution of the above recurrence by using two methods: Bottom-up or top-down.
- We compute overlapping sub-problems only once to avoid the exponential time cost.
- A_i is on dimensions $p_{i-1} \times p_i$.
- Table $m[1 \ldots n, 1 \ldots n]$ is used for storing $m[i, j]$ and $s[1 \ldots n-1, 2 \ldots n]$ is used to store the value of k for which $m[i, j]$ is minimum for all $i \leq k < j$.

Step 3: Computing the optimal costs II

Bottom-up Approach:

MATRIX-CHAIN-ORDER(p,n)

1. Let $m[1 \ldots n, 1 \ldots n]$ and $s[1 \ldots n-1, 2 \ldots n]$
2. for $i = 1$ to n
3. $\quad m[i, i] = 0$
4. for $l = 2$ to n // l is the chain length
5. $\quad$ for $i = 1$ to $n - l + 1$
6. $\quad\quad j = i + l - 1$
7. $\quad\quad m[i, j] = \infty$
8. $\quad\quad$ for $k = i$ to $j - 1$
9. $\quad\quad\quad q = m[i, k] + m[k + 1, j] + p_{i-1} \times p_k \times p_j$
10. $\quad\quad\quad$ if $q < m[i, j]$
11. $\quad\quad\quad\quad m[i, j] = q$
12. $\quad\quad\quad\quad s[i, j] = k$
13. return m, s

Step 3: Computing the optimal costs III

Top-down Approach:

MEMOIZED-MATRIX-CHAIN(p,n)

1. Let $m[1 \ldots n, 1 \ldots n]$
2. for $i = 1$ to n
3. for $j = i$ to n
4. $m[i, i] = \infty$
5. return LOOKUP-CHAIN(m,s,p,1,n)

LOOKUP-CHAIN(m,s,p,i,j)

1. if $m[i, j] < \infty$
2. return $m[i, j]$
3. if $i = j$
4. $m[i, j] = 0$
5. else for $k = i$ to $j - 1$

Step 3: Computing the optimal costs IV

6. q =LOOKUP-CHAIN(m,s,p,i,k)
   +LOOKUP-CHAIN(m,s,p,k+1,j)+$p_{i-1}p_k p_j$
7. if $q < m[i,j]$
8. $m[i,j] = q$
9. $s[i,j] = k$
10. return $m[i,j]$

Step 4: Constructing an optimal solution I

- Table $m[i, j]$ gives the number of scalar multiplication to multiply matrices from A_i to A_j but does not show the order of multiplication
- Table $s[i, j]$ store the position of the parenthesis to partition the matrices from A_i to $A_k = A_{s[i,j]}$ and $A_{s[i,j]+1}$ to A_j.
- So the multiplication will be $(A_i \ldots A_{s[i,j]})$ and $(A_{s[i,j]+1} \ldots A_j)$

PRINT-OPTIMAL-PARENS(s,i,j)

1. if $i = j$
2. print A_i
3. else print "("
4. PRINT-OPTIMAL-PARENS$(s, i, s[i, j])$
5. PRINT-OPTIMAL-PARENS$(s, s[i, j] + 1, j)$
6. print ")"

Complexity of the MCM is $O(n^3)$

Longest Common Subsequence

- Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ of length m and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ of length n
- $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a common subsequence of X and Y if Z is a subsequence of both X and Y
- Z is a Longest common subsequence of X and Y if X and Y have no common subsequence of length greater then Z
- ith prefix of X is $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ for $i = 0, 1, \ldots, m$

Step 1: Characterizing a longest common subsequence

- Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.
- **Optimal substructure of an LCS**
- If $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
- If $x_m \neq y_n$ then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y
- If $x_m \neq y_n$ then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}

Step 2: A recursive solution

- In step 1 there are two cases:
- Case 1: if $x_m = y_n$ then find the LCS of X_{m-1} and Y_{n-1}
- Appending x_m to LCS of X_{m-1} and Y_{n-1} gives LCS of X and Y
- Case 2: if $x_m \neq y_n$ then solve two sub-problems: find LCS of X_{m-1} and Y and find LCS of X and Y_{n-1}
- Longer of these two LCS will be the LCS of X and Y
- Let $c[i,j]$ is the length of an LCS of the sequences X_i and Y_j
- If either of two sequences is empty i.e. $i = 0$ or $j = 0$ then LCS will be of length zero that is $c[i,j] = 0$
- Recursive solution will be

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Step 3: Computing the length of an LCS I

- Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.
- Let Table $c[i, j]$ to store the length of an LCS of the sequences X_i and Y_j
- $c[0 \ldots m, 0 \ldots n]$
- Let Table $b[i, j]$ to store the direction of the flow of length of LCS to construct an optimal solution
- $b[1 \ldots m, 1 \ldots n]$

Step 3: Computing the length of an LCS II

Bottom-up Approach

LCS-LENGTH(X, Y)

1. $m = length(X)$
2. $n = length(Y)$
3. Let $c[0 \ldots m, 0 \ldots n]$ and $b[1 \ldots m, 1 \ldots n]$ two new tables
4. for $i = 1$ to m
5. $\quad c[i, 0] = 0 // Y$ is empty
6. for $j = 1$ to n
7. $\quad c[0, j] = 0 // X$ is empty
8. for $i = 1$ to m
9. $\quad$ for $j = 1$ to n
10. $\quad\quad$ if $x_i = y_j$
11. $\quad\quad\quad c[i, j] = c[i - 1, j - 1] + 1$

Step 3: Computing the length of an LCS III

12. $b[i,j] = \nwarrow$
13. else if $c[i-1,j] \geq c[i,j-1]$
14. $c[i,j] = c[i-1,j]$
15. $b[i,j] = \uparrow$
16. else
17. $c[i,j] = c[i,j-1]$
18. $b[i,j] = \rightarrow$
19. return c, b

Length of the LCS is **c[m,n]**
Complexity of the algorithm is $\Theta(mn)$

Step 3: Computing the length of an LCS IV

Top-down Approach:

MEMOIZED-LCS(X, Y, m, n)

1. Let $c[0 \ldots m, 0 \ldots n]$
2. for $i = 0$ to m
3. for $j = 0$ to n
4. $c[i, j] = 0$
5. return LCS(X, Y, c, b, m, n)

LCS(X, Y, c, b, i, j)

1. if $c[i, j] > 0$
2. return $c[i, j]$
3. if $x_i = y_j$
4. $c[i, j] = LCS(X, Y, c, b, i - 1, j - 1] + 1$
5. $b[i, j] = \nwarrow$

Step 3: Computing the length of an LCS V

6. else if $LCS(X, Y, c, b, i-1, j) \geq LCS(X, Y, c, b, i, j-1)$
7. $\quad c[i, j] = LCS(X, Y, c, b, i-1, j)$
8. $\quad b[i, j] = \uparrow$
9. else
10. $\quad c[i, j] = LCS(X, Y, c, b, i, j-1)$
11. $\quad b[i, j] = \rightarrow$
12. return $c[i, j]$

Step 4: Constructing an LCS I

- Step 3 calculates $c[i,j]$ and $b[i,j]$
- $c[i,j]$ gives the length of LCS of X_i and Y_j
- $b[i,j]$ tells from where we calculated $c[i,j]$
- $\nwarrow$ shows that $x_i = y_j$ and part of LCS
- $\uparrow$ shows that $c[i-1,j] \geq c[i,j-1]$
- $\rightarrow$ shows that $c[i,j-1] > c[i-1,j]$
- call PRINT-LCS(b, X, m, n)

PRINT-LCS(b, X, i, j)

1. if $i = 0$ or $j = 0$
2. return
3. if $b[i,j] = \nwarrow$
4. PRINT-LCS$(b, X, i-1, j-1)$
5. print x_i

Step 4: Constructing an LCS II

6. else if $b[i, j] = \uparrow$
7. PRINT-LCS$(b, X, i - 1, j)$
8. else
9. PRINT-LCS$(b, X, i, j - 1)$

Complexity of the algorithm is $\Theta(m + n)$

Optimal binary search trees I

- Suppose we want to store English dictionary
- We can use any balanced binary search try to store the dictionary word
- If every word of dictionary is to be searched same number of times
- we can search any world in $O(\lg n)$
- Total time taken is $O(mn \lg n)$
  where $m =$ number of times each word to be searched
  $n =$ number of words
- This system works perfectly if every word is searched equal number of times.
- But if the some words are searched very frequently and some are very rare then this performance of this system goes slow
- An alternate of this is Optimal binary search trees

Optimal binary search trees II

- Let a $K = \langle k_1, k_2, \ldots k_n \rangle$ sequence of n distinct key in the sorted order such that $k_1 < k_2 < \cdots < k_n$
- If we search keys not in database that we represent as dummy keys d_i
- Total key and dummy keys will be $d_0 < k_1 < d_1 < k_2 \ldots d_{n-1} < k_n < d_n$
- d_0 represents all keys less than k_1 and not in database, so are dummy keys
- d_n represents all keys greater then k_n and not in database, so are dummy keys
- d_i for $0 < i < n$ represents all keys greater then k_i and less than k_{i+1} and not in database, so are dummy keys.
- So Total no of n Keys are $k_1, k_2, \ldots k_n$
- So Total no of $n+1$ Dummy Keys are $d_0, d_1, d_2, \ldots d_n$

Optimal binary search trees III

- Let probability of key k_i to be searched is p_i for $1 \leq i \leq n$.
- Let probability of dummy key d_i to be searched is q_i for $0 \leq i \leq n$.
- If we build some binary search tree for these keys and dummy keys then keys will be on internal nodes and represents successful search and dummy keys will be on external nodes and represents unsuccessful search.
- Every search is either successful finding some key k_i or unsuccessful finding some dummy key d_i so:

$$\sum_{k=1}^{n} p_k + \sum_{k=0}^{n} q_k = 1$$

Optimal binary search trees IV

- if k_r is the root of the tree then two subtree will be: one subtree having keys from k_1 to k_{r-1} and dummy keys d_0 to d_{r-1} and other subtree having keys from k_{r+1} to k_n and dummy keys d_r to d_n
- if we search any keys from k_i to k_j then dummy keys will be d_{i-1} to d_j for $1 \leq i \leq j \leq n$
- If $j = i - 1$ then there is no key but only one dummy key d_{i-1}
- Let

$$w[i,j] = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k$$

Optimal binary search trees V

- k_r is the root of the tree having keys from k_i to k_j and dummy keys from d_{i-1} to d_j then two subtree will be: one subtree having keys from k_i to k_{r-1} and dummy keys d_{i-1} to d_{r-1} and other subtree having keys from k_{r+1} to k_j and dummy keys d_r to d_j
- The actual cost of a search equals the number of nodes examined i.e. the depth of the node found by the search in tree plus 1
- Then the expected cost of a search in tree for al keys and dummy keys is

$$E[1,n] = \sum_{k=1}^{n} (depth(k_k) + 1) \cdot p_k + \sum_{i=0}^{n} (depth(d_k) + 1) \cdot q_k$$

$$E[1,n] = \sum_{k=1}^{n} depth(k_k).p_k + \sum_{i=0}^{n} depth(d_k).q_k + \sum_{k=1}^{n} p_k + \sum_{k=0}^{n} q_k$$

Optimal binary search trees VI

$$E[1,n] = \sum_{k=1}^{n} depth(k_k).p_k + \sum_{i=0}^{n} depth(d_k).q_k + 1$$

- The expected cost of a search in tree for keys from k_i to k_j and dummy keys d_{i-1} to d_j is

$$E[i,j] = \sum_{k=i}^{j} (depth(k_k) + 1) .p_k + \sum_{k=i-1}^{j} (depth(d_k) + 1) .q_k$$

$$E[i,j] = \sum_{k=i}^{j} depth(k_k).p_k + \sum_{k=i-1}^{j} depth(d_k).q_k + \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k$$

$$E[i,j] = \sum_{k=i}^{j} depth(k_k).p_k + \sum_{k=i-1}^{j} depth(d_k).q_k + w[i,j]$$

Step 1: The structure of an optimal binary search tree I

- Let T is a optimal tree having subtree $T^{'}$ with keys $k_i \ldots k_j$ and dummy keys $d_{i-1} \ldots d_j$ for some $1 \leq i \leq j \leq n$
- Then $T^{'}$ will also be a optimal subtree
- We can find the optimal cost of the tree by optimal cost of the subtree
- Take k_r as the root from keys k_i to k_j and dummy keys d_{i-1} to d_j
- Now there are two subtrees one having keys from k_i to k_{r-1} with dummy keys from d_{i-1} to d_{r-1}
- The second tree having keys from k_{r+1} to k_j with dummy keys from d_r to d_j

Step 1: The structure of an optimal binary search tree II

- Now the total cost of the tree will be sum of cost of left subtree having keys from k_i to k_{r-1} with dummy keys from d_{i-1} to d_{r-1} plus cost of right subtree having keys from k_{r+1} to k_j with dummy keys from d_r to d_j plus cost of key k_r
- d_i is the dummy key between the keys k_i and k_{i+1}
- If we choose k_i as the root then there is no key (keys from k_i to k_{i-1}) on left subtree but only one dummy key d_{i-1}
- If we choose k_j as the root then there is no key (keys from k_{j+1} to k_j) on right subtree but only one dummy key d_j
- By taking all key from k_i to k_j as root one by one, we can find the optimal cost of the tree

Step 2: A recursive solution I

- Let a subproblem with keys k_i to k_j and dummy keys d_{i-1} to d_j where $i \geq 1$, $j \leq n$ and $j \geq i-1$
- Let $e[i,j]$ is the expected cost of searching an optimal binary search tree containing the keys k_i to k_j and dummy keys d_{i-1} to d_j
- Finally we have to find out $e[1,n]$
- Let $w[i,j]$ is the sum of the probability of all the keys $k_i \ldots k_j$ and dummy keys $d_{i-1} \ldots d_j$ and $w[1,n] = 1$

$$w[i,j] = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k$$

- When $j = i-1$ then no key but only dummy key so $e[i,i-1] = q_{i-1}$ and $w[i,i-1] = q_{i-1}$

Step 2: A recursive solution II

- When $j > i-1$, then take a key k_r as root such that left subtree having keys from k_i to k_{r-1} and right subtree having keys from k_{r+1} to k_j is optimal
- When a tree becomes subtree of a root node then hight of the each node of the tree increases by one.
- The expected search cost of this subtree increases by the sum of all the probabilities in the subtree i.e.

$$e[i, r-1] = e[i, r-1] + w[i, r-1]$$

$$e[r+1, j] = e[r+1, j] + w[r+1, j]$$

Step 2: A recursive solution III

- Taking k_r as the root $e[i,j]$ becomes

$$e[i,j] = e[i,r-1] + w[i,r-1] + p_r + e[r+1,j] + w[r+1,j]$$

$$e[i,j] = e[i,r-1] + e[r+1,j] + w[i,j]$$

$$\because w[i,j] = w[i,r-1] + p_r + w[r+1,j]$$

- The recursive solution will be:

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w[i,j]\} & \text{if } j > i-1 \end{cases}$$

Step 3: Computing the expected search cost of OBST I

- Optimal cost of OBST is stored in $e[1 \ldots n+1, 0 \ldots n]$ that is size of upper triangular matrix is $(n+1) \times (n+1)$.
- $e[1,0]$ is used to store the value of d_0 and $e[n+1, n]$ is used to store the value of d_n
- $e[i,j]$ is used to store the value of optimal search cost of the keys from k_i to k_j for $1 \leq i \leq n+1, 0 \leq j \leq n,$ and $j \geq i-1$
- $root[i,j]$ is used to store the root of the tree having keys from k_i to k_j for $1 \leq i \leq j \leq n$
- $e[i, i-1] = w[i, i-1] = q_{i-1}$ for $1 \leq i \leq n+1$
- $w[i,j] = w[i,j-1] + p_j + q_j$ for $1 \leq i \leq n+1, j > i-1$
- $e[i,j] = \min_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w[i,j]\}$ for $j > i-1$

Step 3: Computing the expected search cost of OBST II

Bottom-up Approach:

OPTIMAL-BST(p,q,n)

1. Let $e[1 \ldots n+1, 0 \ldots n], w[1 \ldots n+1, 0 \ldots n]$ and $root[1 \ldots n, 1 \ldots n]$
2. for $i = 1$ to $n+1$ //when no keys, will be only one dummy key
3. $\quad w[i, i-1] = q_{i-1}$
4. $\quad e[i, i-1] = q_{i-1}$
5. for $l = 1$ to n // l number of keys in the subtree
6. $\quad$ for $i = 1$ to $n - l + 1$
7. $\quad\quad j = i + l - 1$
8. $\quad\quad e[i, j] = \infty$
9. $\quad\quad w[i, j] = w[i, j-1] + p_j + q_j$
10. $\quad\quad$ for $r = i$ to j

Step 3: Computing the expected search cost of OBST III

11. $q = e[i, r-1] + e[r+1, j] + w[i, j]$
12. if $q < e[i, j]$
13. $e[i, j] = q$
14. $root[i, j] = r$
15. return e, $root$

Step 3: Computing the expected search cost of OBST IV

Top-down Approach:

MEMOIZED-OPTIMAL-BST(p,q,n)

1. Let $e[1 \ldots n+1, 0 \ldots n]$, $w[1 \ldots n+1, 0 \ldots n]$ and $root[1 \ldots n, 1 \ldots n]$
2. for $i = 1$ to $n+1$
3. for $j = i-1$ to n
4. $e[i,j] = \infty$
5. if $j = i-1$
6. $w[i,j] = q_{i-1}$
7. else
8. $w[i,j] = w[i,j-1] + p_j + q_j$
9. return LOOKUP-OBST(e,w,root,p,q,1,n)

Step 3: Computing the expected search cost of OBST V

LOOKUP-OBST(e,w,root,p,q,i,j)

1. if $e[i,j] < \infty$
2. return $e[i,j]$
3. if $j = i - 1$
4. $e[i,j] = q_{i-1}$
5. else for $r = i$ to j
6. q =LOOKUP-OBST(e,w,root,p,q,i,r-1) +LOOKUP-OBST(e,w,root,p,q,r+1,j)+$w[i,j]$
7. if $q < e[i,j]$
8. $e[i,j] = q$
9. $root[i,j] = r$
10. return $e[i,j]$

Step 4: Constructing an OBST I

- Table $e[i, j]$ gives the cost of OBST for the keys k_i to k_j with dummy keys d_{i-1} to d_j
- Table $root[i, j]$ store the root node $k_r = k_{root[i,j]}$
- At $k_r = k_{root[i,j]}$ there will be two subtree: one left subtree with keys k_i to $k_{root[i,j]-1}$ and second right subtree with $k_{root[i,j]+1}$ to k_j
- First call will be:
- if $n \geq 1$ //atleast one key
- $r = root[1, n]$
- print k_r is the root of the OBST
- PRINT-OBST$(root, i, r-1, r, "left")$
- PRINT-OBST$(root, r+1, j, r, "right")$
- else
- print d_0 is the root of the OBST

Step 4: Constructing an OBST II

PRINT-OBST(root,i,j,r,child)

1. if $i \leq j$
2. $\quad c = root[i, j]$
3. $\quad$ print k_c is child of k_r
4. $\quad$ PRINT-OBST$(root, i, r-1, c, "left")$
5. $\quad$ PRINT-OBST$(root, r+1, j, c, "right")$

Complexity of the OBST is $O(n^3)$

Class Test I ADA, IT 5^{th} *Sem*, Dated: 06^{th} *Sep*, 2019

Even numbered students will attempt even numbered questions and odd numbered students will attempt odd numbered questions.

Q1. Prove that $\lg n! = O(n \lg n)$

Q2. Find the solution of $T(n) = 3T(\sqrt[3]{n}) + \lg_3 n$

Q3. Find the cost to multiply the matrices whose dimensions are given as 15,5,10,2,5.

Q4. Find the maximum length subsequence of the two sequences $< 00110101 >$ and $< 101100110 >$

Class Test I ADA, CSE 5^{th} *Sem*, Dated: 06^{th} *Sep*, 2019

Even numbered students will attempt even numbered questions and odd numbered students will attempt odd numbered questions.

Q1. Prove that $n! = O(n^n)$

Q2. Find the solution of $T(n) = 2T(\frac{n}{2}) + n \lg n$

Q3. Drive the recursive formula to find the cost of multiplication of n matrices.

Q4. Drive the recursive formula to find the maximum length of common sequence of two sequences.

0-1 Knapsack Problem I

- Set of n items from 1 to n are given
- w_i is the weight of the i^{th} item
- v_i is the value of the i^{th} item
- We can pick any item as a whole, fraction of weight is not allowed
- Total weight we pick can not be more than W but value of the sum of picked items must be maximum
- So 0-1 Knapsack Problem is an optimization problem stated as:

$$\text{Maximize } \sum_{i=1}^{n} v_i x_i$$

$$\text{under the constraints } \sum_{i=1}^{n} x_i w_i \leq W, \text{ and } x_i \in \{0, 1\}$$

0-1 Knapsack Problem II

- In bounded Knapsack Problem we can pick upto c_i copies of x_i item:

  Maximize $\sum_{i=1}^{n} v_i x_i$

  under the constraints $\sum_{i=1}^{n} x_i w_i \leq W$, and $x_i \in \{0 \ldots c_i\}$

- In unbounded Knapsack Problem we can pick any number of x_i item including none:

  Maximize $\sum_{i=1}^{n} v_i x_i$

  under the constraints $\sum_{i=1}^{n} x_i w_i \leq W$, and $x_i \geq 0$

Compute the maximum Bag Value I

- Total number of items is n
- $v[i]$ is the value of the i^{th} item whose weight is $w[i]$.
- Let Table $m[0 \ldots n, 0 \ldots W]$ is used to store the value of the bag for n items and W maximum capacity.
- $m[i, j]$ is the value of the bag from i item and j capacity.

Compute the maximum Bag Value II

Bottom-up Approach:

OPTIMAL-0-1KP(w,v,n,W)

1. Let $m[0 \ldots n, 0 \ldots W]$
2. for $i = 0$ to n
3. for $j = 0$ to W
4. if $i = 0$ or $j = 0$ //no item or nil bag capacity
5. $m[i, j] = 0$
6. else if $w[i] \leq j$// i^{th} item can be accommodated in bag
7. if $(m[i-1, j-w[i]] + v[i]) > m[i-1, j]$
8. $m[i, j] = m[i-1, j-w[i]] + v[i]$//pick the item
9. else
10. $m[i, j] = m[i-1, j]$//don't pick the item
11. return m

Compute the maximum Bag Value III

Top-down Approach:

Mamoized-0-1KP(w,v,n,W)

1. Let $m[0 \ldots n, 0 \ldots W]$
2. for $i = 0$ to n
3. for $j = 0$ to W
4. $m[i, j] = 0$
5. return $knapsack(m, w, v, n, W)$

$knapsack(m, w, v, i, j)$

1. if $m[i, j] > 0$
2. return $m[i, j]$
3. if $i = 0$ or $j = 0$ //no item or nil bag capacity
4. return 0
5. if $w[i] < j$

Compute the maximum Bag Value IV

6. return $max(knapsack(m, w, v, i-1, j),$ $knapsack(m, w, v, i-1, j-w[i]) + v[i])$
7. else
8. return $knapsack(m, w, v, i-1, j)$
9. return m

Complexity of the Knapsack is $O(nW)$

Binomial Coefficient

- $(x+a)^n$ can be expended as ${}^nC_0x^na^0 + {}^nC_1x^{n-1}a^1 + \cdots + {}^nC_kx^{n-k}a^k + \ldots {}^nC_nx^0a^n$
- Coefficient of the term $x^{n-k}a^k$ is know as Binomial Coefficient $C(n,k) = {}^nC_k$
- $C(n,k) = \frac{n!}{n-k!k!}$
- $C(n,k) = C(n-1,k-1) + C(n-1,k)$ if $n > k > 0$
- $\Rightarrow \frac{n-1!}{n-k!k-1!} + \frac{n-1!}{n-k-1!k!}$
- $\Rightarrow \frac{n-1!}{n-k!k-1!}\frac{k}{k} + \frac{n-1!}{n-k-1!k!}\frac{n-k}{n-k}$
- $\Rightarrow \frac{(n-1)!k}{n-k!k!} + \frac{(n-1)!(n-k)}{n-k!k!}$
- $\Rightarrow \frac{(n-1)!(k+n-k)}{n-k!k!}$
- $\Rightarrow \frac{n!}{n-k!k!}$
- $\Rightarrow C(n,k)$
- $C(n,0) = C(n,n) = 1$

Computing a Binomial Coefficient I

Bottom-up Approach:

No optimal solution only fixed value

BC(c,n,k)

1. Let $c[0 \ldots n, 0 \ldots k]$
2. for $i = 0$ to n
3. for $j = 0$ to i
4. if $i = j$ or $j = 0$
5. $c[i, j] = 1$
6. else
7. $c[i, j] = c[i - 1, j - 1] + c[i - 1, j]$
8. return c

Computing a Binomial Coefficient II

Top-down Approach:

Mamoized-BC(c,n,k)

1. Let $c[0 \ldots n, 0 \ldots k]$
2. for $i = 0$ to n
3. for $j = 0$ to i
4. $c[i, j] = 0$
5. return $BC(c, n, k)$

$BC(c, i, j)$

1. if $c[i, j] > 0$
2. return $c[i, j]$
3. if $i = j$ or $j = 0$
4. return 1
5. return $c[i, j] = BC(c, i - 1, j - 1) + BC(c, i - 1, j]$

Complexity of the Knapsack is $O(nk)$

Floyd Warshall algorithm

- The algorithm find the shortest route between all nodes/vertices of the non-negative directed weighted graph G.
- Suppose V is set of n nodes/vertices in graph G i.e. $\{1, 2, \ldots, n\} \in V$
- E is the set of edges of the graph G .i.e $(i, j) \in E$ if there is a edge from i to j in the graph
- So we can say $G = (V, E)$
- w_{ij} is the weight of the edge from vertex i to j.
- if p is a path from vertex v_i to v_j through vertices $v_1, v_2, \ldots, v_l$ then the nodes other than source and destination is known as intermediate nodes.
- Sum of weight of the edges of the path p is the distance d_{ij} between vertex v_i to v_j
- if this distance is the shortest among all possible paths from v_i to v_j then it is called the shortest path $\delta(i, j)$

Step 1: The structure of a shortest path I

- Let graph G has set of vertices $V = \{1, 2, \ldots, n\}$
- Let any pair of vertices $i, j \in V$, all paths can be drawn from the set of intermediate vertices $\{1, 2, \ldots, k\}$ for some k
- Let p is the shortest/minimum-weight simple path from i to j.
- There are two cases for any vertex k : either k is the part of the minimum-weight path or not the part of the path.
- If k is not the part of the path then it can be removed from the the set of vertices without effecting the path. So the path will be from $\{1, 2, \ldots, k-1\}$ or we can say a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$
- But if it is part of the path then we can divide the whole path p into two paths: p_1 - from i to k and p_2 - from k to j

Step 1: The structure of a shortest path II

- Intermediate nodes of p_1 and p_2 will be from $\{1, \ldots, k-1\}$ because now k is a source in one path and destination in other but not the intermediate node.

$$\delta(i,j) = \begin{cases} \delta(i,j) & \text{if } k \text{ is not part of } p \\ \delta(i,k) + \delta(k,j) & \text{if } k \text{ is part of } p \end{cases}$$

with intermediate nodes $\{1, 2, \ldots, k-1\}$

Step 2: A recursive solution I

- d_{ij}^k is the shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$
- if $k = 0$ then no intermediate node so $d_{ij}^0 = w_{ij}$ if there is a direct edge from i to j else $d_{ij}^0 = \infty$ if there is no direct edge from i to j and $d_{ij}^0 = 0$ if $i = j$.
- $d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$
- π_{ij}^k is the predecessor of vertex j for the shortest path from vertex i with all intermediate vertices in the set $\{1, 2, \ldots, k\}$
- if $k = 0$ then no intermediate node so $\pi_{ij}^0 = i$ if there is a direct edge from i to j else $\pi_{ij}^0 = NIL$ if there is no direct edge from i to j and $\pi_{ij}^0 = NIL$ if $i = j$.

Step 2: A recursive solution II

- $\pi_{ij}^k = \pi_{ij}^{k-1}$ if $d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1}$ or $\pi_{ij}^k = \pi_{kj}^{k-1}$ if $d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1}$

$$d_{ij}^k = \begin{cases} 0 & \text{if } i = j, k = 0 \\ \infty & \text{if } (i,j) \notin E, k = 0 \\ w_{ij} & \text{if } (i,j) \in E, k = 0 \\ \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\} & \end{cases}$$

$$\pi_{ij}^k = \begin{cases} NIL & \text{if } i = j, k = 0 \\ NIL & \text{if } w_{ij} = \infty, k = 0 \\ i & \text{if } w_{ij} < \infty, k = 0 \\ \pi_{ij}^{k-1} & \text{if } d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1} \\ \pi_{kj}^{k-1} & \text{if } d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1} \end{cases}$$

Step 3: Computation of Shortest path I

Let W is the weight matrix and A is the adjacency matrix of the graph G of n vertices.

Bottom-up Approach-I

FLOYD-WARSHALL(W,A,n)

1. $D^0 = W$
2. $\Pi^0 = A$
3. for $k = 1$ to n
4. Let $D^k = (d_{ij}^k)$ a $n \times n$ matrix
5. Let $\Pi^k = (\pi_{ij}^k)$ a $n \times n$ matrix
6. for $i = 1$ to n
7. for $j = 1$ to n
8. if $d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1}$
9. $d_{ij}^k = d_{ij}^{k-1}$

Step 3: Computation of Shortest path II

10. $\pi_{ij}^{k} = \pi_{ij}^{k-1}$
11. else
12. $d_{ij}^{k} = d_{ik}^{k-1} + d_{kj}^{k-1}$
13. $\pi_{ij}^{k} = \pi_{kj}^{k-1}$
14. return D^n, Π^n

Step 3: Computation of Shortest path III

Bottom-up Approach-II

FLOYD-WARSHALL(W,A,n)

1. $D = W$
2. $\Pi = A$
3. for $k = 1$ to n
4. for $i = 1$ to n
5. for $j = 1$ to n
6. if $d_{ij} \leq d_{ik} + d_{kj}$
7. $d_{ij} = d_{ij}$
8. $\pi_{ij} = \pi_{ij}$
9. else
10. $d_{ij} = d_{ik} + d_{kj}$
11. $\pi_{ij} = \pi_{kj}$
12. return D, Π

Greedy Algorithm

- Used for optimization
- Always makes the choice that looks best at the moment
- makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution
- Greedy algorithms do not always yield optimal solutions that means it can not be used for every problem
- If a problem can be solved using greedy that can also be solved using dynamic programming but not vice versa.
- Greedy algorithms are more faster than dynamic programming
- By nature dynamic programming follows bottom up and greedy follows top down approach

Elements of the greedy strategy I

Steps involved in greedy strategy:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

Elements of the greedy strategy II

More generally, we design greedy algorithms according to the following sequence of steps:

1. Design the optimization problem by selecting one as a choice and rest with the subproblem to solve
2. Prove that there is always an optimal solution to the original problem with greedy choice, and greedy choice is always safe.
3. Demonstrate optimal substructure by showing that if we combine an optimal solution of the the subproblem with main problem , we get an optimal solution to the original problem.

Ingredients of greedy approach

Two key ingredients of greedy approach:

1. Optimal substructure: an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms
2. Greedy-choice property: we can assemble a globally optimal solution by making locally optimal (greedy) choices.

An activity-selection problem

- Suppose $S = \{a_1, a_2, \ldots, a_n\}$ is a set of n activities
- Each activity a_i have start time s_i and finish time f_i such that $0 \leq s_i < f_i < \infty$
- Two activity a_i and a_j are compatible if one start after other finishes that is either $s_i \geq f_j$ or $s_j \geq f_i$ or we can say time interval $[s_i, f_i)$ of a_i and $[s_j, f_j)$ of a_j is non overlapping.
- a_0 finishes at 0 and a_{n+1} starts after f_n that is $[-\infty, 0)$ for a_0 and $[f_n, \infty)$ for a_{n+1}

Dynamic programming Solution

- S_{ij} is the set of activities that start after activity a_i finishes and that finish before activity a_j starts
- A_{ij} is the set of maximum mutually compatible activities that start after activity a_i finishes and that finish before activity a_j starts and $Aij \subseteq Sij$
- We want to find A_{0n+1}
- Suppose $a_k \in A_{ij}$
- There is two sub-problems A_{ik} and A_{kj} where $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$
- $A_{ij} = Aik \cup \{a_k\} \cup A_{kj}$
- $|A_{ij}| = |Aik| + |A_{kj}| + 1$
- If $c[i,j]$ is the size of an optimal solution for the set S_{ij} then $c[i,j] = c[i,k] + c[k,j] + 1$

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Greedy choice Solution

- We should choose an activity that leaves the resource available for as many other activities as possible
- Choose the activity in S with the earliest finish time, so that maximum resource available for the following activities
- Since activities are in sorted increasing order of finishing time so select a_1
- After selecting a_1 only one sub-problem remains: selecting the activity from rest of the activities witch starts after a_1.
- Let $S_k = \{a_i \in S : s_i \geq f_k\}$ is the set of activities that start after activity a_k finishes
- So a_1 is finishing first, we have to find out S_1 only.

A recursive greedy algorithm

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

1. $m = k + 1$
2. while $m \leq n$ and $s_m < f_k$ // find the first activity in S_k to finish
3. $\quad m = m + 1$
4. if $m \leq n$
5. $\quad$ return $\{a_m\} \cup$RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6. else
7. $\quad$ return $\emptyset$

Let a_0 is a activity which finishes at $f_0 = 0$. We want to find out optimal solution $S_0 = S$, So first call we be RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n)

An iterative greedy algorithm

GREEDY-ACTIVITY-SELECTOR(s, f, n)

1. $A = \{a_1\}$
2. $k = 1$
3. for $m = 2$ to n
4. if $s_m \geq f_k$
5. $A = A \cup \{a_m\}$
6. $k = m$
7. return A

a_k is the most recent addition to A and activities are arranged in monotonically increasing finish time, so f_k is always the maximum finish time of any activity in A i.e. $f_k = \max\{f_i : a_i \in A\}$ and we are searching the next activity which starts after a_k finishes.

Correctness of the algorithm I

Is the greedy choice in which we choose the first activity to finish is always part of some optimal solution?

- Statement: Consider any nonempty subproblem S_k. let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k
- Let A_k is a maximum-size subset of mutually compatible activities in S_k so $A_k \subseteq S_k$.
- let a_j is the activity in A_k with the earliest finish time
- Now there are two cases : either $a_j = a_m$ or we can replace a_j with a_m in A_k without affecting the solution.
- if $a_j = a_m$ then nothing to do and Statement is correct.
- if $a_j \neq a_m$, then let another set $A'_k = (A_k - \{a_j\}) \cup \{a_m\}$ that is substitute a_j with a_m.

Correctness of the algorithm II

- Now we have to check that is still $A_k^{'}$ is a maximum-size subset of mutually compatible activities.
- For that we have to check the new activity a_m is compatible with the rest of activities in $A_k^{'}$.
- As a_m is the first activity to finish in S_k and a_j is the first activity to finish in A_k and $A_k \subseteq S_k$ so either $a_j = a_m$ or $f_m \leq f_j$
- It means we can replace a_j with a_m and still set A_k will remain maximum sized non overlapping set of activities.
- So a_m will be the part of optimal solution.

Fractional Knapsack problem

- There are n item.
- Value of i^{th} item is v_i and weight is w_i.
- Knapsack capacity of the bag is W.
- We wish to maximize the total bag value subject to the constraint that total weight is less than or equal to W that is:

$$\text{Maximize } \sum_{i=1}^{n} v_i x_i$$

$$\text{under the constraints } \sum_{i=1}^{n} x_i w_i \leq W, \text{ and } x_i = [0, 1]$$

Optimal-substructure property

- Consider that W is the optimal load of the bag.
- If we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ from the $n - 1$.
- If this $W^1 = W - w$ is not optimal but W^2 is optimal then $W^2 + w$ will be optimal not W
- But W is optimal under assumption so W^1 will be optimal.
- So Fractional Knapsack problem satisfied optimal-substructure property

Greedy property

- Suppose items are arranged in descending order of $\frac{v_i}{w_i}$
- Take W is the sum of weights from first i weights.
- If we remove a weight w of one item i from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ from the $i - 1$.
- Because weights are arranged in descending order of value per unit weight
- So if we remove one item then remaining items also more valuable then the rest.
- Hence greedy-choice property is applicable.

Correctness of the algorithm

- Suppose items are arranged in descending order of $\frac{v_i}{w_i}$
- Take W is the sum of weights from first i weights with value V.
- If we remove a weight w of one item $j \leq i$ from the optimal load and replace w from the rest of item $k > i$
- The value of resultant load $V^{'} = V - v_j + v_k$
- But $V^{'} \leq V$ as $v_j \geq v_k$.
- So the the selection of the most valuable item is correct.

An iterative greedy algorithm

FKP(v, w, n, W)

1. Let items are arranged in descending order of $\frac{v_i}{w_i}$
2. J=0,V=0,i=1
3. while($i \leq n$ and $J < W$)
4. if $w_i \leq W - J$
5. $V = V + v_i$
6. $J = J + w_i$
7. else
8. $V = V + \frac{W-J}{w_i} \times v_i$
9. $J = W$
10. $i = i + 1$
11. return V

Complexity of the algorithm is $O(n \lg n)$ for sorting and $O(n)$ for selecting the weights. So it will be $O(n \lg n)$

Huffman Codes I

- Huffman Codes is used for data compression.
- Data is a sequence of characters.
- Each character is given with their frequency
- Each character is encoded into a codeword using some scheme.
- Suppose there are n characters in the set C.
- Each character $c \in C$ have frequency $freq_c$
- $bit(c)$ is the number of bits required to code c character whose frequency is $freq_c$.
- Our aim is to design a encoding scheme so that we can minimize the total length of codeword.

$$\text{minimize} \sum_{c \in C} freq_c \times bit(c)$$

Huffman Codes II

- We can use some standard fixed length formatting such as ASCII
- In this case the the total space requirement will be $8 \times \sum_{c \in C} freq_c$
- But we can save more space using non standard fixed length formatting scheme for n characters.
- The length of the code will be $\lceil \lg n \rceil$
- In this case the $\lceil \lg n \rceil$ number of bits are required to represent each character and total bits will be $\lceil \lg n \rceil \times \sum_{c \in C} freq_c$
- Huffman codes can be used even to save more space than both of the above and this uses variable length encoding scheme for each character.
- Huffman codes are prefix codes.

Huffman Codes III

- **Prefix Codes:** Codes in which no codeword is a prefix of some other codeword are called **Prefix Codes**.
- The benefits of Prefix codes are simplified decoding, unambiguous encoding.
- But the disadvantages is that we can not start decoding in between the encoded codeword into the original character.
- Huffman code uses full binary tree for encoding, in which every non leaf node has two children
- All characters are used as leaf, so total leaf will be n
- There are $n - 1$ internal nodes as in full binary tree.
- Each left child is labelled as 0 and each right child is labelled as 1.
- Label value from root to leaf will be the encoding for that leaf character.

Huffman Codes IV

- All characters are kept in min priority queue according to their frequency that is least frequency character is at front of the queue.
- Every time we sum the least two frequency of the first two element of the queue and make one
- So after $n-1$ operation we will be having only one element in the min priority queue and that will be the root of the tree.
- Code length of the character c will be equal to the depth of the c $d_T(c)$ in tree T. So

$$\text{minimize } B(T) = \sum_{c \in C} freq_c \times d_T(c)$$

Huffman Coding algorithm

HUFFMAN(C)//C is the set of n characters

1. $n = |C|$
2. $Q = C$ //Q is Min-priority Queue
3. for $i = 1$ to $n - 1$
4. allocate a new node z
5. $z \rightarrow left = EXTRACT - MIN(Q)$
6. $z \rightarrow right = EXTRACT - MIN(Q)$
7. $freq_z = freq_{z \rightarrow left} + freq_{z \rightarrow right}$
8. $INSERT(Q, z)$
9. return $EXTRACT - MIN(Q)$

Line 2 will be require $O(n \lg n)$ time for building the min heap of n items. For loop executes $n - 1$ times and each time it rebuild the min heap in $O(\lg n)$ times. Total time complexity will be $O(n \lg n)$

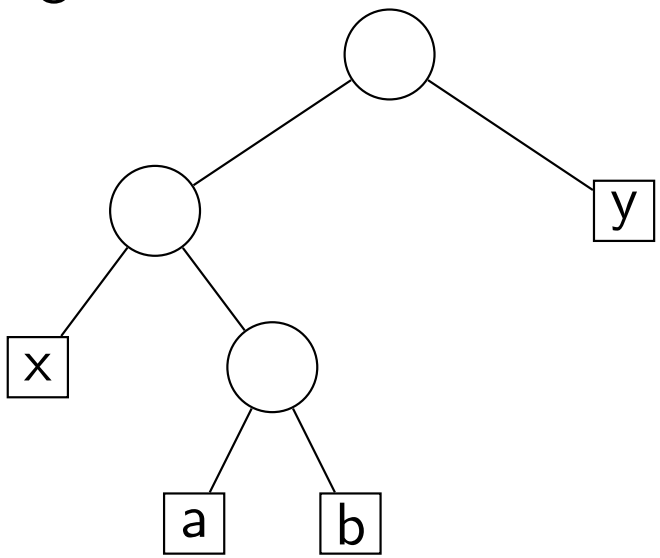
Correctness of the Huffman algorithms I

If character x and y having the least frequency then they will be at highest depth in the some optimal tree T for Huffman Code.

- Let character x and y having the least frequency $freq_x$ and $freq_y$ respectively. Also suppose $freq_x \leq freq_y$
- Suppose x and y are not at highest depth but a and b are at the highest depth in the tree T.
- Let $freq_a \leq freq_b$
- Since $freq_x, freq_y$ are lowest frequency and $freq_a, freq_b$ are any arbitrary frequency. Also $freq_x \leq freq_y$ and $freq_a \leq freq_b$. So $freq_x \leq freq_y \leq freq_a \leq freq_b$.
- if $freq_x = freq_b$ then $freq_x = freq_a = freq_y = freq_b$ and we can change the position of the x and y with a and b and hence proved.

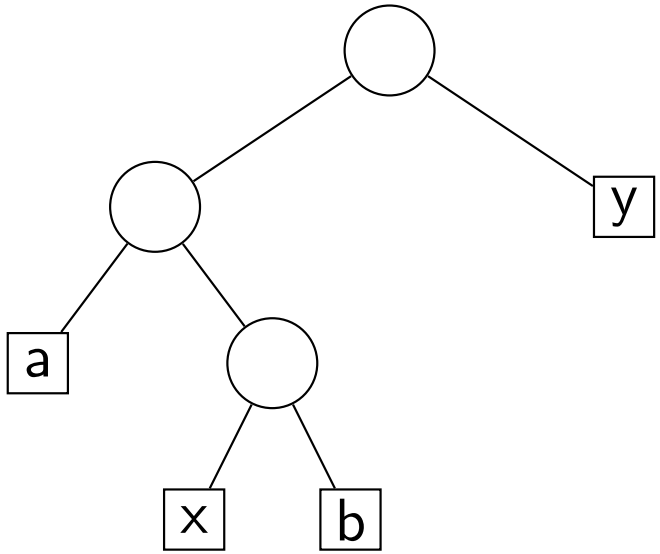
Correctness of the Huffman algorithms II

- So assume $freq_x \neq freq_b$ and take tree T as shown in below figure:



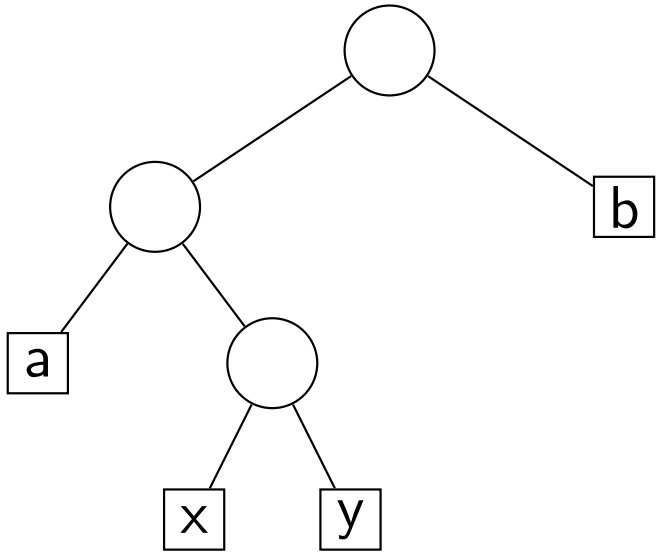
Correctness of the Huffman algorithms III

- Interchange the position of x with a in tree T and build tree T' as shown in below figure:



Correctness of the Huffman algorithms IV

- Interchange the position of y with b in tree $T^{'}$ and build tree $T^{''}$ as shown in below figure:



Correctness of the Huffman algorithms V

- Find $B(T) - B(T')$

$B(T) - B(T')$

$$= \sum_{c \in C} freq_c \times d_T(c) - \sum_{c \in C} freq_c \times d_{T'}(c)$$

$$= freq_x \times d_T(x) + freq_a \times d_T(a) - freq_x \times d_{T'}(x) - freq_a \times d_{T'}(a)$$

$$= freq_x \times d_T(x) + freq_a \times d_T(a) - freq_x \times d_T(a) - freq_a \times d_T(x)$$

$$= (freq_a - freq_x)(d_T(a) - d_T(x))$$

Now $(freq_a - freq_x) \geq 0$ because $freq_x$ is the least frequency and $(d_T(a) - d_T(x)) \geq 0$ because a is at the highest depth. So

$$(freq_a - freq_x)(d_T(a) - d_T(x)) \geq 0$$

Correctness of the Huffman algorithms VI

$\Rightarrow B(T) - B(T^{'}) \geq 0$

$\Rightarrow B(T) \geq B(T^{'})$

But $B(T)$ can not be greater then $B(T^{'})$ because $B(T)$ is the optimal value. So the only possibility is $B(T) = B(T^{'})$
Same way we can prove $B(T^{'}) = B(T^{''})$
So $B(T) = B(T^{'}) = B(T^{''})$
So $B(T^{''})$ is an optimal Huffman code where least frequency characters x and y is at the highest depth.

Task-scheduling problem I

The problem of scheduling unit-time tasks with deadlines and penalties/profits for a single processor has the following inputs:

- Set $S = \{a_1, a_2, \ldots, a_n\}$ is a set of n tasks.
- Each task requires unit time to complete
- Every task a_i having some deadline d_i such that $1 \leq d_i \leq n$
- There is n task and every task requires unit time so in n unit of time starting from 0 to n all tasks will be finished.
- If task a_i finishes before deadline d_i then there will be no penalty but if the task is late and finishes after deadline then there will be a penalty p_i
- Or if task a_i finishes before deadline d_i then there will be profit p_i but if the task is late and finishes after deadline then there will be no profit.

Task-scheduling problem II

- Our aim is to minimize the penalty or maximize the profit.

  minimize $Penalty(S) = \sum_{a_i \in S} p_i$ if a_i finishes after deadline d_i

  maximize $Profit(S) = \sum_{a_i \in S} p_i$ if a_i finishes before deadline d_i

Solution

- Suppose task a_i having deadline $d_i = 5$ is late and task a_j having deadline $d_j = 6$ is early
- If we execute task a_i on or after 6, still it will be late
- If we execute task a_j on or before 5, still it will be early
- So a early task will be early if we execute that task on or before the deadline and a late task will be late if we execute that task anywhere after deadline.
- We try to execute the high penalty task on time and if any task is late then we can make that task maximum late.
- We try to execute the early task at the scheduled deadline or before.
- We can always transform an arbitrary schedule into early-first form, in which the early tasks precede the late tasks
- Complexity of the algorithm will be $O(n^2)$ using Greedy because n independence check takes $O(n)$ time.

Minimum Spanning Trees (MST) I

- Tree is a connected undirected acyclic graph.
- Tree is a set of nodes and set of edges connected that nodes that is $T = (V, E)$ and each edge $(u, v) \in E$ having weight $w(u, v)$
- Minimum means the sum of all edges to connect all nodes is minimum
- Spanning means tree span over all the nodes.
- So Minimum Spanning Tree is a undirected weighted acyclic graphs that covers all the nodes with minimum sum of the weights.
- There will be $n - 1$ edges for n vertices in the MST.

Minimum Spanning Trees (MST) II

- Let G is a undirected graph with set of vertices V and set of edges E and each edge $(u, v) \in E$ having weight $w(u, v)$. We have to find out an acyclic subset $T \subseteq G$ that connects all of the vertices and whose total weight is minimum that is

$$\text{minimize } W(T) = \sum_{(u,v)\in T} w(u, v)$$

- T in this case will be MST.

GENERIC-MST(G,w)

1. $A = \emptyset$
2. while A does not form a spanning tree
3. find an edge (u, v) that is safe for A
4. $A \cup \{(u, v)\}$
5. return A

Kruskal's algorithm I

- Kruskal's algorithm uses the least weight edge to connect the two tree to make one.
- Kruskal's algorithm qualifies as a greedy algorithm because at each step it adds to the forest an edge of least possible weight
- It uses a disjoint-set data structure to maintain several disjoint sets of elements
- The operation $FIND - SET(u)$ returns address of the representative element from the set that contains u
- We check $FIND - SET(u)$ and $FIND - SET(v)$ for any edge (u, v). If they return the same address then it means the are already connected.
- If $FIND - SET(u)$ and $FIND - SET(v)$ returns different value that it means they are the members of two different tree and using $UNION(u, v)$ we make one tree connected by edge (u, v)

Kruskal's algorithm II

MST-KRUSKAL(G,w)

1. $A = \emptyset$
2. for each vertex $v \in V$
3. $\quad MAKE - SET(v)$
4. sort the edges of E into nondecreasing order by weight w
5. for each edge $(u, v) \in E$ taken in nondecreasing order by weight
6. $\quad$ if $FIND - SET(u) \neq FIND - SET(v)$
7. $\quad\quad A \cup \{(u, v)\}$
8. $\quad\quad UNION(u, v)$
9. return A

Time complexity of the algorithm is $O(E \lg E)$ or $O(E \lg V)$ as $|E| < |V|^2 \Rightarrow \lg E = O(\lg V)$

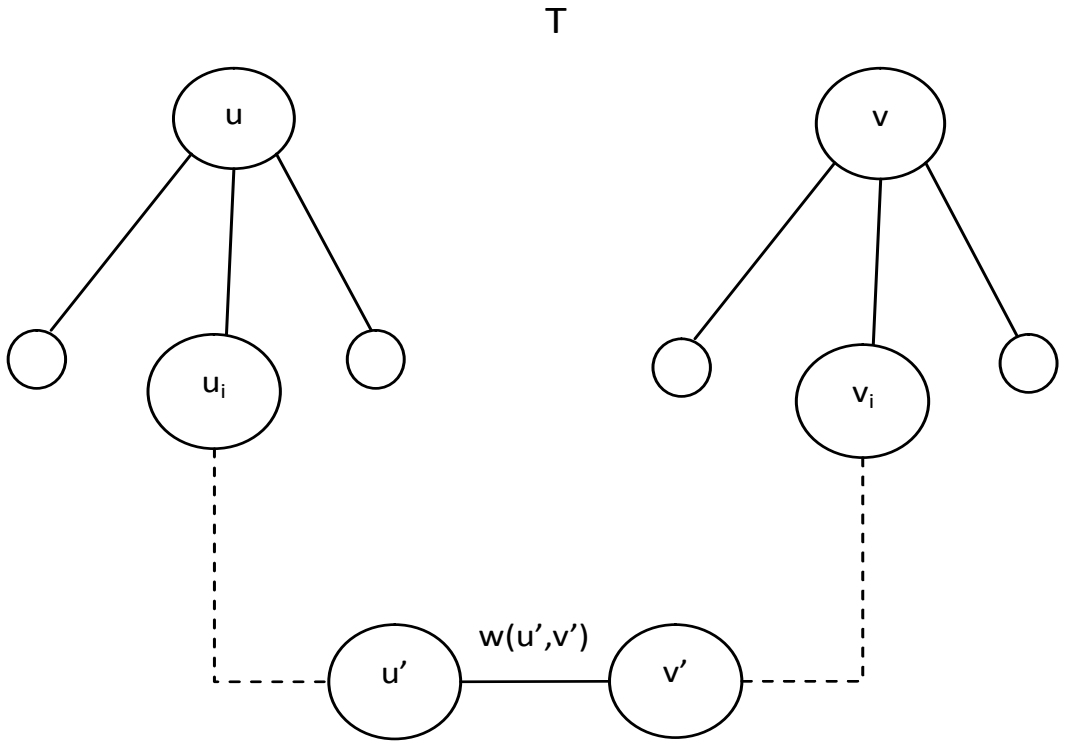
Correctness of Kruskal's Algorithm I

Prove that if $w(u, v)$ is minimum weighted edge of the graph G then edge (u, v) will be part of the some MST T.

- Suppose T is a MST with weight $W(T)$ and edge (u, v) is not the part of the tree T.
- It means u and v are connected by some other edges being tree T is spanning tree.

Correctness of Kruskal's Algorithm II

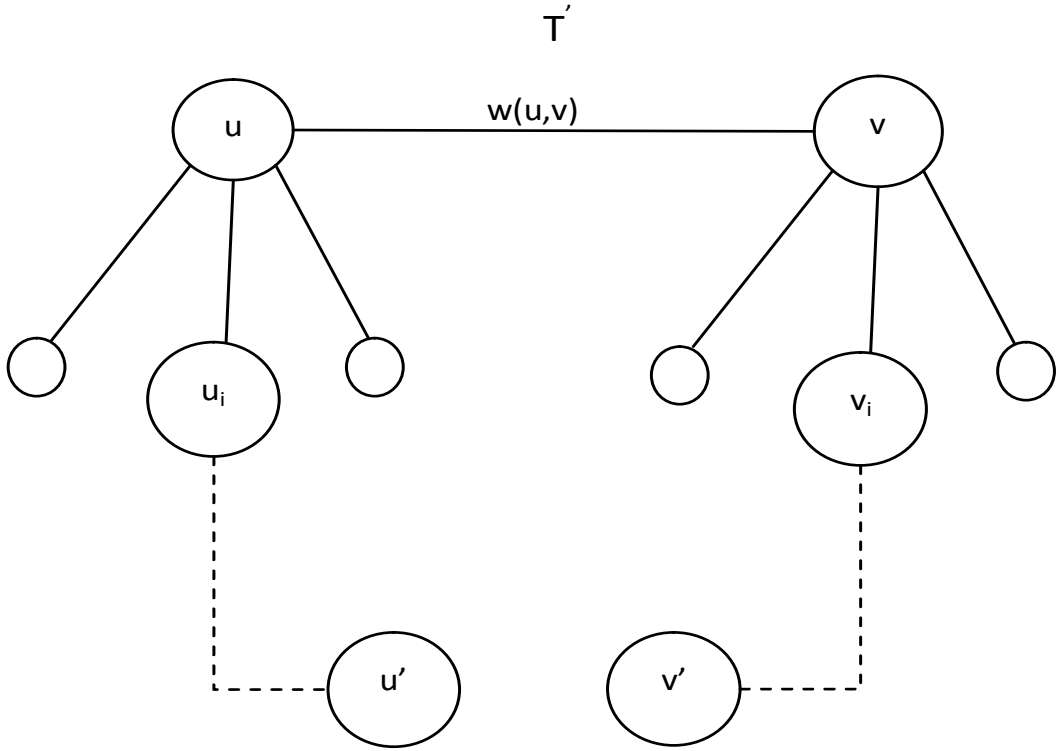
- Suppose vertex u is connected by u_i vertices and v is connected by v_j vertices as shown in below figure:



- Now suppose $(u^{'}, v^{'})$ is a least weight edge from the path $(u, u_i, \ldots, v_j, v)$

Correctness of Kruskal's Algorithm III

- Remove (u', v') and add edge (u, v) to make the tree connected as shown in below figure:



- The weight of the tree T' will be: $W(T') = W(T) -$ weight of the edge removed $+$ weight of the edge added

Correctness of Kruskal's Algorithm IV

- $\Rightarrow W(T^{'}) = W(T) - w(u^{'}, v^{'}) + w(u, v)$
- $\Rightarrow W(T^{'}) = W(T) - (w(u^{'}, v^{'}) - w(u, v))$
- $\Rightarrow W(T) - W(T^{'}) = w(u^{'}, v^{'}) - w(u, v)$
- $\Rightarrow W(T) - W(T^{'}) \geq 0$
- $\Rightarrow W(T) \geq W(T^{'})$
- But $W(T)$ is the MST as per assumption so $W(T)$ can't be greater than $W(T^{'})$
- So $W(T) = W(T^{'})$ and hence $T^{'}$ is also a MST and (u, v) will be the part of MST $T^{'}$

Prim's algorithm I

- Prim's algorithm has the property that the edges in the set A always form a single tree unlike Kruskal.
- We starts taking any node r as root and grows till the tree spans all the vertices in V
- In each step a new minimum edge vertex is added into A
- The algorithm uses greediness, as each step it adds to the tree an edge that contributes the minimum weight to the tree.
- We use min-priority queue Q based on key to store all vertices that are not the part of the MST till now.
- For each vertex v, the value $key[v]$ will be the minimum weight of the edge connected v with all other nodes of the graph.
- if any node is not connected with node v then $key[v] = \infty$
- $key[r] = 0$ as r is the root of the tree

Prim's algorithm II

- As we are expending tree by taking the node with minimum weight edge at every step so the tree will be MST at every step.
- $\pi[v]$ is the predecessor of v. For example if (u, v) is the edge of the tree then $\pi[v] = u$
- $adj[u] = v$ if there is an edge from u to v that is (u, v)

Algorithm for Prim I

MST-PRIM(G,w,r)

1. $A = \emptyset$
2. $Q = \emptyset$
3. for each $u \in V$
4. $\quad key[u] = \infty$
5. $\quad \pi[u] = NIL$
6. $\quad Insert(Q, u)$
7. $key[r] = 0$
8. while $Q \neq \emptyset$
9. $\quad u = Extract(Q)$
10. $\quad$ for each $v \in adj[u]$
11. $\quad\quad$ if $v \in Q$ and $w(u, v) < key[v]$
12. $\quad\quad\quad \pi[v] = u$

Algorithm for Prim II

13. $key[v] = w(u, v)$
14. $A \cup \{(u, v)\}$
15. return

Time complexity of the algorithm is $O(V \lg V + E \lg V)$ or $O(E \lg V)$ using Min-heap and $O(E + V \lg V)$ using Fibonacci heap.

Shortest-paths problem

- G is a given weighted and directed graph with set of vertices V and set of edges E such that $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}$
- The weight $w(p)$ of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of edges of the path. That is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- The shortest-path weight $\delta(u, v)$ from u to v is defined as

$$\delta(u, v) = \begin{cases} min\{w(p) : u \rightsquigarrow v & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

- A shortest path from vertex u to vertex v is any path p with weight $w(p) = \delta(u.v)$

Variants of Shortest path

- Single-source shortest-paths problem : Find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$
- Single-destination shortest-paths problem: Find a shortest path from each vertex $u \in V$ to a given destination vertex $d \in V$
- Single-pair shortest-path problem: Find a shortest path from a given source $s \in V$ to a given destination $d \in V$.
- All-pairs shortest-paths problem: Find a shortest path from each vertex $u \in V$ to each vertex $v \in V$.

Optimal substructure of a shortest path I

Statement: If p is a optimal path from v_0 to v_k then p_{ij} will be the optimal path from p_i to p_j for any value of i and j such that $0 \leq i \leq j \leq k$.

Proof:

- Given p is the optimal path from v_0 to v_k that is $w(p)$ is minimum over $v_0 \rightsquigarrow v_k$
- Let's break path p into 3 parts :
- p_{0i} is the path from v_0 to v_i with weight $w(p_{0i})$
- p_{ij} is the path from v_i to v_j with weight $w(p_{ij})$
- p_{jk} is the path from v_j to v_k with weight $w(p_{jk})$
- That is $v_0 \rightsquigarrow v_i \rightsquigarrow v_j \rightsquigarrow v_k$
- Now $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$
- Let $w(p'_{ij})$ is the minimum weight path from v_i to v_j through other vertices that is $w(p'_{ij}) \leq w(p_{ij})$

Optimal substructure of a shortest path II

- So $w(p^{'}) = w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$
- As $w(p'_{ij}) \leq w(p_{ij})$ so $w(p^{'}) \leq w(p)$
- But $w(p)$ is the optimal path so $w(p^{'}) \not< w(p)$
- The only possible case is $w(p^{'}) = w(p)$
- It means $w(p'_{ij}) = w(p_{ij})$

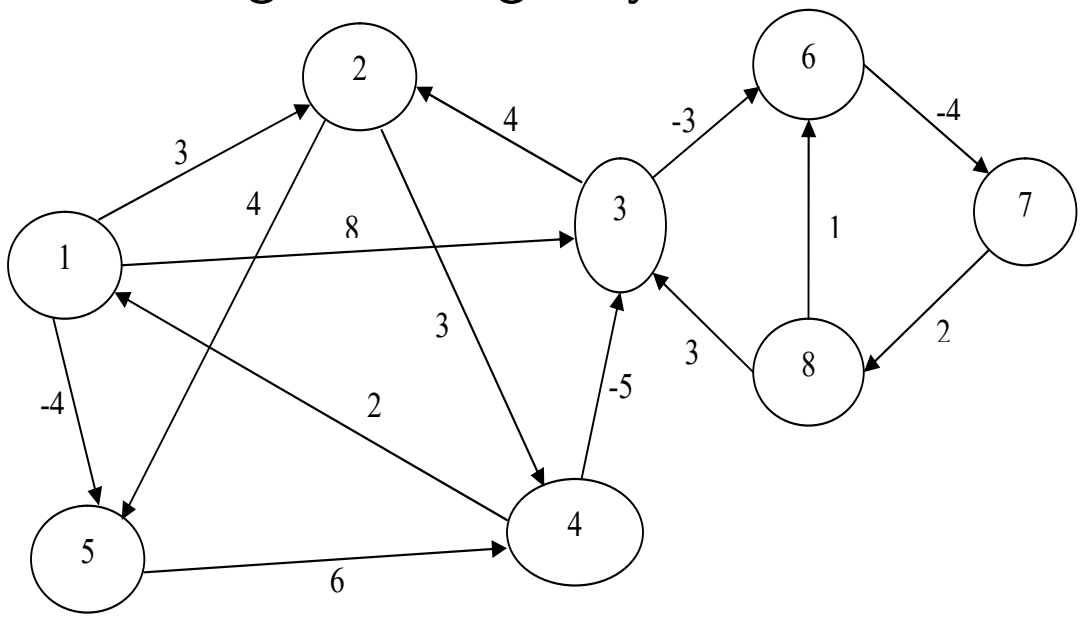
So Subpaths of shortest paths are shortest paths.

Negative-weight cycle I

- An edge with with negative weight is called negative-weight edge.
- If $w(p)$ is less than 0 ($w(p) < 0$) then it is known as negative-weight cycle
- If no negative-weight cycles reachable from the source s then shortest path said to well defined
- If negative-weight cycles reachable from the source s then shortest path said to not well defined
- If shortest path is well defined then $\delta(u, v) =$"some finite value"
- If shortest path is not well defined then $\delta(u, v) = -\infty$
- It means no path from s to a vertex on the cycle can be a shortest path

Negative-weight cycle II

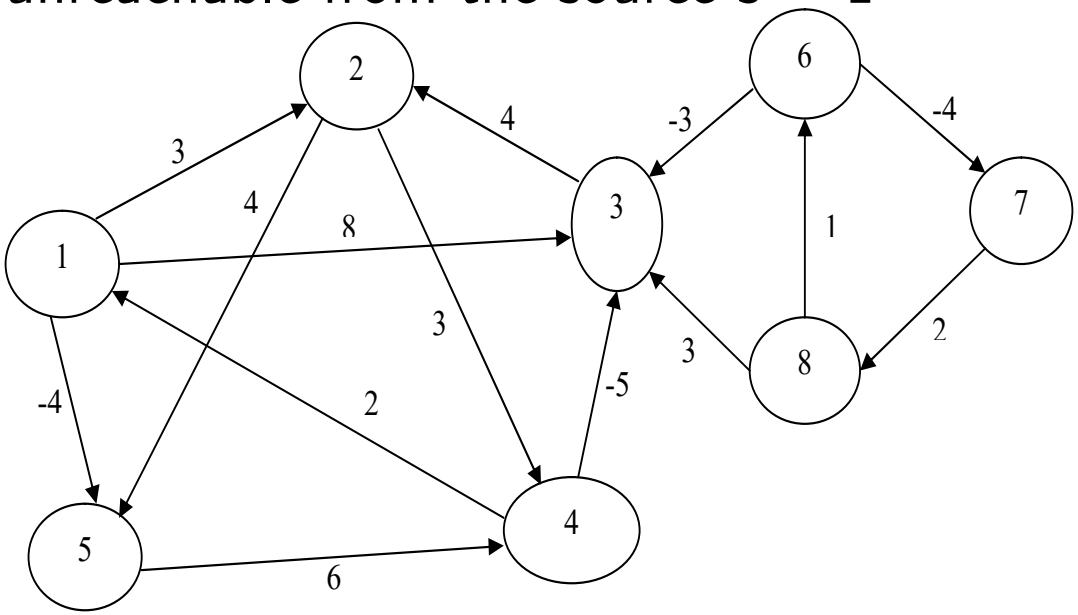
- That is we can always find a path with lower weight by just traversing the negative-weight cycle once more.
- See below figure for negative-weight cycle reachable from the source $s = 1$



- But if negative-weight cycles unreachable from the source $s = 1$ then also we can find the shortest path

Negative-weight cycle III

- Please see below figure for negative-weight cycle cycles unreachable from the source $s = 1$



- There is no path from source $s = 1$ to the unreachable nodes. so $w(p) = \infty$ in this case
- Any acyclic path can have maximum $|V| - 1$ edges and $|V|$ vertices.

Representing shortest paths I

- G is a graph with $G = (V, E)$. $\pi[v]$ is a predecessor of node v. It means there is edge between $(\pi[v], v)$
- **Predecessor subgraph** $G_\pi = (V_\pi, E_\pi)$ is subgraph of G taking s as source where:.
- The vertex set V_π to be the set of vertices of G with non-NIL predecessors, plus the source s:
  $V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$
- The directed edge set E_π is the set of edges induced by the π values for vertices in V_π:
  $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$

Representing shortest paths II

- **Shortest-paths tree** is a directed subgraph $G' = (V', E')$ with s as root where $V' \subseteq V$ and $E' \subseteq E$ such that :
  1. G' forms a rooted tree with root s
  2. V' is the set of vertices reachable from s in G
  3. $\forall v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G
- Shortest paths are not necessarily unique and neither are shortest-paths trees.
- $d[v]$ is the shortest-path estimate between the source s and each vertex $v \in V$
- We initialize the shortest-path estimates and predecessors as : **INITIALIZE-SINGLE-SOURCE(G,s)**
  1. for each vertex $v \in V$
  2. $\quad d[v] = \infty$
  3. $\quad \pi[v] = NIL$

Representing shortest paths III

4. $d[s] = 0$

- **Relaxation**: Relaxing an edge (u, v) means testing whether we can improve the shortest path to v found so far by going through u, if yes then update $d[v]$ and $\pi[v]$
- The following code performs a relaxation step on edge (u, v) **RELAX(u,v,w)**
  1. if $d[v] > d[u] + w(u, v)$
  2. $\quad d[v] = d[u] + w(u, v)$
  3. $\quad \pi[v] = u$

Properties of shortest paths I

- **Triangle inequality:** For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$
- **Upper-bound property:** $\forall v \in V, d[v] \geq \delta(s, v)$ and once $d[v] = \delta(s, v)$ then it never changes
- **No-path property:** If there is no path from s to v, then we always have $d[v] = \delta(s, v) = \infty$
- **Convergence property:** If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$ and if $d[u] = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $d[v] = \delta(s, v)$ at all times afterward.
- **Path-relaxation property:** If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $d[v_k] = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Properties of shortest paths II

- **Predecessor-subgraph property:** Once $f[v] = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Dijkstra's algorithm I

- It is used to find single-source shortest-paths problem on a non-negative weighted,directed graph
- S is the set of vertices whose final shortest-path weights from the source s have already been determined
- The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate
- Adds u to S, and relaxes all edges leaving u.
- We use a min-priority queue Q of vertices based on their d values

Algorithm: DIJKSTRA(G,w,s)

1. INITIALIZE-SINGLE-SOURCE(G,s)
2. $S = \emptyset$
3. $Q = V$
4. while $Q \neq \emptyset$

Dijkstra's algorithm II

5. $u =$EXTRACT-MIN(Q)
6. $S = S \cup \{u\}$
7. for each vertex $v \in adj[u]$
8. $\quad RELAX(u, v, w)$

Time complexity of the algorithm is $O(V \lg V + E \lg V)$ or $O(E \lg V)$ using Min-heap and $O(E + V \lg V)$ using Fibonacci heap.

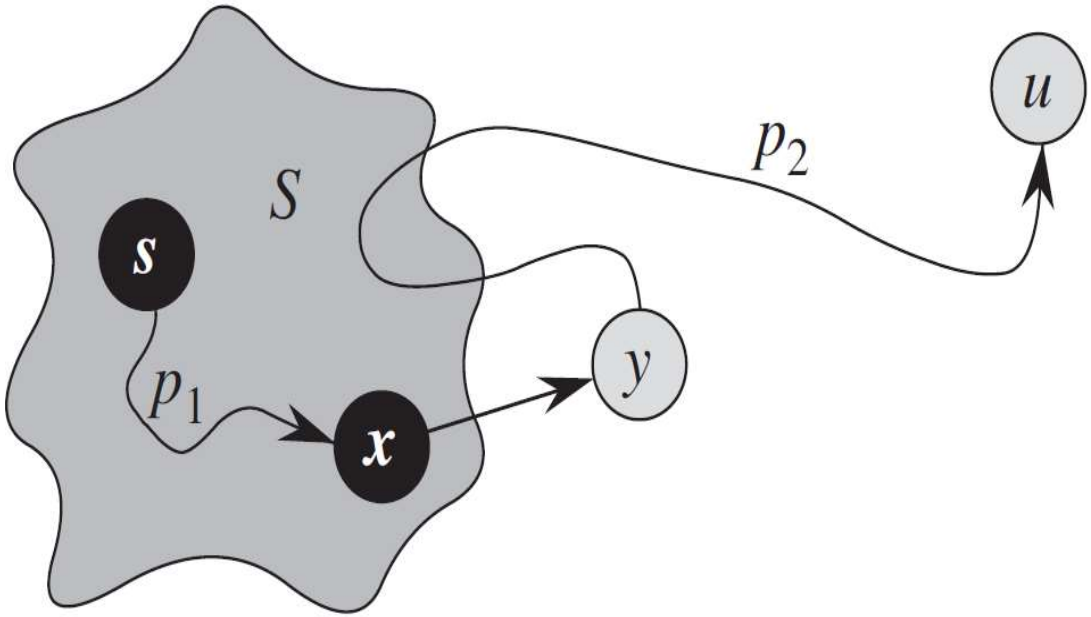
Correctness of Dijkstra's algorithm I

We have to prove that $d[u] = \delta(s, u)$ when u is added to set S for each vertex $u \in E$ if u is reachable from s.

- We will prove it by contradict.
- Let u is the first vertex such that $\delta(s, u) < d[u] < \infty$ when add to S.
- $u \neq s$ because s is the first vertex added to the S and $d[s] = \delta(s, s) = 0$
- $S \neq \emptyset$ because at least s was there in S before u is added to S

Correctness of Dijkstra's algorithm II

- There must be at least one path from s to u because our assumption is that $\delta(s, u) < d[u] < \infty$, There must be at least one shortest path p from s to u.



- Let p is a path from s to u before adding u into S such that $s \rightsquigarrow x \rightsquigarrow y \rightsquigarrow u$ and $s, x \in S$ and $y, u \notin S$

Correctness of Dijkstra's algorithm III

- $d[y] = \delta(s, y)$ because y is predecessor of u so it will be relax before u and as per assumption u is the first vertex where $d[u] \neq \delta(s, u)$
- $\delta(s, y) \leq \delta(s, u)$ because y is the predecessor of u and at the the shortest path from s also there is no negative weights so $w(y, u) \geq 0$

$$d[y] = \delta(s, y)$$

$$\leq \delta(s, u)$$

$$\leq d[u]$$

- But because both vertices $y, u \notin S$ and when we extract u from Q then $d[u] \leq d[y]$
- $d[y] \leq d[u]$ and $d[u] \leq d[y]$ is only possible when $d[u] = d[y]$

Correctness of Dijkstra's algorithm IV

- So $d[y] = \delta(s, y) = \delta(s, u) = d[u]$ means out assumption is wrong.
- Hence proved that $d[u] = \delta(s, u)$ when u is added to set S

Bellman-Ford algorithm

- A graph with $|V|$ vertices can have maximum $|V| - 1$ edges and $|V|$ vertices in any simple path.
- If number of edges in the path is greater than $|V| - 1$ or vertices on the path is greater than $|V|$ including source and destination then there is cycle in path.
- Bellman works upon this fundamental and relaxes all edges $|V| - 1$ times to find the shortest path from single source.
- After $|V| - 1$ passes, there will be shortest path among nodes form source s, if exists.
- One more cycle of relax, if decrease the shortest path from source then it means there will be $|V|$ edges and $|V| + 1$ vertices and hence a negative weight cycle.
- The algorithm returns either TRUE if there is no negative weight cycle or FALSE if negative weight cycle

Algorithm

BELLMAN-FORD(G,w,s)

1. INITIALIZE-SINGLE-SOURCE(G, s)
2. for $i = 1$ to $|V| - 1$
3. for each edge $(u, v) \in E$
4. RELAX(u, v, w)
5. for each edge $(u, v) \in E$
6. if $d[v] > d[u] + w(u, v)$
7. return FALSE
8. return TRUE

Time complexity of the algorithm is $O(VE)$

Correctness of the Bellman-Ford algorithm I

Statement: If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, we have $d[v] = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree rooted at s. If G does contain a negative-weight cycle reachable from s, then the algorithm returns FALSE.

- Suppose that graph G contains no negative-weight cycles that are reachable from the source s.
- Now we have to prove that the algorithms will return TRUE.
- If vertex v is reachable from s, then at termination after $|V| - 1$ iterations $d[v] = \delta(s, v)$
- If vertex v is unreachable from s, then at termination after $|V| - 1$ iterations $d[v] = \delta(s, v) = \infty$

Correctness of the Bellman-Ford algorithm II

- After termination of $|V| - 1$ iterations, for all edge $(u, v) \in E, d[v] = \delta(s, v)$

  $d[v] = \delta(s, v)$

  $d[u] = \delta(s, u)$

  $d[v] = \delta(s, v) \leq \delta(s, u) + w(u, v)$-by the triangle inequality

  $d[v] \leq d[u] + w(u, v)$

- So $d[v] \leq d[u] + w(u, v)$ for all edge $(u, v) \in E$ in the predecessor subgraph G_π is a shortest path from source s
- So $d[v] > d[u] + w(u, v)$ of line no 6 of the algorithm will never returns TRUE and hence algorithm will return TRUE.
- Suppose that graph G contains negative-weight cycles that are reachable from the source s.

Correctness of the Bellman-Ford algorithm III

- Now we have to prove that the algorithms will return FALSE.
- Let $c = \langle v_0, v_1, \ldots, v_k \rangle$ is a negative weight cycle in the path p reachable from source s in graph G where $v_0 = v_k$
- $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \leftarrow$ because of negative weight cycle
- Now let contradiction that there is a negative weight cycle but still algorithm returns TRUE.
- So $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$.
- Taking summation over all the k values

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Correctness of the Bellman-Ford algorithm IV

- $d[v_1] + d[v_2] + \cdots + d[v_{k-1}] + d[v_k]$

  $\leq d[v_0] + d[v_1] + \cdots + d[v_{k-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$

- $0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i), \because v_0 = v_k$
- But $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$ because of negative weight cycle
- So our assumption of contradiction is wrong and the algorithm will return FALSE.

String Matching I

- $\sum$ is a set of characters.
- $T = T[1..n]$ and $P = P[1..m]$ is a string of characters and each character belongs to $\sum$ that is $T[i] \in \sum$ for $1 \leq i \leq n$ and $P[i] \in \sum$ for $1 \leq i \leq m$.
- T is a text of length n from which we want to find out the occurrences of the pattern P of length m.
- The length of the pattern may be less then or equal to length of the text i.e. $m \leq n$
- The pattern P may be any where in the text T i.e. $T[s+1..s+m] = P[1..m]$ or $T[s+j] = P[j]$ for $1 \leq j \leq m$
- s is know as shift and value of s may be from 0 to $n-m$; $n-m+1$ total values.
- When $s = 0$ the first m characters of text $T[1..m]$ will be checked with P.

String Matching II

- That is if s is the shift then the position of the text to be matched is $s+1$
- Pattern P occurs in text T with shift s or beginning at position $s+1$ if $0 \leq s \leq n-m$ and $T[s+1..s+m] = P[1..m]$ that is if $T[s+j] = P[j]$ for $1 \leq j \leq m$
- If P occurs with shift s in T , then we call s a valid shift; otherwise, we call s an invalid shift.
- So the string-matching problem is the problem of finding all valid shifts s with which a given pattern P occurs in a given text T.

$$s = \begin{cases} valid & \text{if } 0 \leq s \leq n-m \text{ and } T[s+1..s+m] = P[1..m] \\ invalid & \text{if } 0 \leq s \leq n-m \text{ and } T[s+1..s+m] \neq P[1..m] \end{cases}$$

Notation and terminology I

- $\sum^*$ is the set of all finite-length strings formed using characters from the alphabet $\sum$.
- The zero-length empty string, denoted ε is also belongs to $\sum^*$
- The length of a string x is denoted $|x|$.
- The concatenation of two strings x and y, denoted xy, has length $|x| + |y|$ and consists of the characters from x followed by the characters from y.
- String w is a prefix of a string x if $x = wy$ for some string $y \in \sum^*$ and represented as $w \sqsubset x$ and $|w| \leq |x|$
- String w is a suffix of a string x if $x = yw$ for some string $y \in \sum^*$ and represented as $w \sqsupset x$ and $|w| \leq |x|$
- The empty string ε is both a suffix and a prefix of every string x

Notation and terminology II

- For any strings x and y and any character a, we have $x \sqsupset y$ if and only if $xa \sqsupset ya$.
- P_k is the k-character prefix $P[1..k]$ of the pattern $P[1..m]$. $P_0 = \varepsilon, P_m = P = P[1..m]$
- T_k is the k-character prefix $T[1..k]$ of the text $T[1..n]$. $T_0 = \varepsilon, T_n = T = T[1..n]$
- Finding all shifts s in the range $0 \leq s \leq n - m$ such that $P \sqsupset T_{s+m}$ is the string-matching problem.

Overlapping-suffix lemma

Statement: Suppose x, y and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. Then 3 cases will be there:

1. If $|x| < |y|$ then $x \sqsupset y$
2. If $|y| < |x|$ then $y \sqsupset x$
3. If $|x| = |y|$ then $x = y$

Overlapping-prefix lemma

Statement: Suppose x, y and z are strings such that $x \sqsubset z$ and $y \sqsubset z$. Then 3 cases will be there:

1. If $|x| < |y|$ then $x \sqsubset y$
2. If $|y| < |x|$ then $y \sqsubset x$
3. If $|x| = |y|$ then $x = y$

Naive string-matching algorithm I

NAIVE-STRING-MATCHER(T,P)

1. $n = length(T)$
2. $m = length(P)$
3. for $s = 0$ to $n - m$
4. if $P[1..m] = T[s + 1..s + m]$
5. print pattern occurs at position $s + 1$.

Naive string-matching algorithm II

NAIVE-STRING-MATCHER(T,P)

1. $n = length(T)$
2. $m = length(P)$
3. for $s = 0$ to $n - m$
4. $\quad valid = 1$
5. $\quad$ for $j = 1$ to m
6. $\quad\quad$ if $P[j] \neq T[s + j]$
7. $\quad\quad\quad valid = 0$
8. $\quad$ if $valid = 1$
9. $\quad\quad$ print pattern occurs with sift s.

Complexity of the algorithm is $O((n - m + 1)m)$

Rabin-Karp algorithm I

- The algorithm depends on the fact that two numbers will be equal if and only if the modulus of these two numbers to a third number is equal.
- If two numbers are same then the modulus of these numbers to a third number will be same or if the modulus of the two numbers are not equal then these two numbers can not be same.
- If the modulus of the two numbers are equal then numbers may or may not be same.
- For string matching we will find out the modulus of text T and pattern P treating as numerical value to a third number and if the remainder is not same then no need to check the two strings.

Rabin-Karp algorithm II

- But if the remainder of the text T and pattern P to a third number is same then text T may or may be equal to pattern P. So we will check $P \sqsupset T_{s+m}$.
- Let p is numerical value of the pattern $P[1..m]$ in suppose radix-d(10).
- Let t_s denote the decimal value of the length-m substring of text $T[s+1..s+m]$ for $s = 0$ to $n - m$
- $t_s = p$ if and only if $T[s+1..s+m] = P[1..m]$
- So s is a valid shift if and only if $t_s = p$
- We can calculate t_0 and p in time $\Theta(m)$ using Horners rule:

$$p = P[m]+10(P[m-1]+10(P[m-2]+\cdots+10(P[2]+10P[1]))\dots)$$

$P[m]$ is the least order digit and $P[1]$ is the highest order digit.

$$t_0 = T[m]+10(T[m-1]+10(T[m-2]+\cdots+10(T[2]+10T[1]))\dots)$$

Algorithm I

RABIN-KARP-MATCHER(T,P,d,q)

1. $n = lenght(T)$
2. $m = lenght(P)$
3. $h = d^{m-1} mod\ q$
4. $p = 0$
5. $t_0 = 0$
6. for $i = 1$ to m
7. $\quad p = (dp + P[i]) mod\ q$
8. $\quad t_0 = (dt_0 + T[i]) mod\ q$
9. for $s = 0$ to $n - m$
10. $\quad$ if $p == t_s$
11. $\quad\quad$ if $P[1..m] == T[s + 1..s + m]$
12. $\quad\quad\quad$ print Pattern occurs with shift s

Algorithm II

13. if $s < n - m$

14. $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1])mod\ q$

Complexity of the algorithm is $\Theta(m)$ for preprocessing of the string P and T_m to find out p and t_0 respectively. For all $t_0, t_1, \ldots, t_{n-m}$, we check $P[1..m] = T[s+1..s+m]$ if modulus of p and t_i for $0 \leq i \leq n - m$ to a third number is same. So total time taken is $\Theta((n - m + 1)m)$.

String matching with finite automata I

Finite automata

A finite automaton A is a 5-tuple $(Q, q_0, F, \sum, \delta)$ where

- Q is a finite set of states
- $q_0 \in Q$ is the start/initial state
- $F \subseteq Q$ is a set of accepting/final states
- $\sum$ is a finite input alphabet
- δ is a transition function of A such that $Q \times \sum \rightarrow Q$ that is $\delta(q_i \in Q, c \in \sum) = q_j \in Q$

Final-state function ϕ is a sequence of transitions, the automata will be in a state after scanning string w that is $\phi(w) = q$ where $w \in \sum^*, q \in Q$. $\phi(\varepsilon) = q_0$ and $\phi(wa) = \delta(\phi(w), a)$ for $w \in \sum^*, a \in \sum$. A accepts a string w if and only if $\phi(w) \in F$.

String matching with finite automata II

String-matching automata

A finite automaton A is a 5-tuple $(Q, q_0, F, \sum, \delta)$ for pattern $P[1..m]$ and text $T[1..n]$ where

- $Q = \{0, 1, \ldots, m\}$ is a set of states
- $0 \in Q$ is the start state
- $m \subseteq Q$ is a accepting state
- $\sum$ is a finite input alphabet
- δ is a transition function of A such that $\delta(q, a) = \sigma(P_q a)$

Suffix function $\sigma(x)$ is the length of the longest prefix of P that is also a suffix of x:

$$\sigma(x) = \max\{k : p_k \sqsupset x\}$$

$$\phi(T_i) = \sigma(T_i)$$

when $\phi(T_i) = \sigma(T_i) = m$ it means $P_m \sqsupset T_i$ that is last m characters of T_i is same as P that is pattern matches at shift $i - m$.

Computing the transition function

COMPUTE-TRANSITION-FUNCTION($P, \sum$)

1. $m = length(P)$
2. for $q = 0$ to m
3. for each character $a \in \sum$
4. $k = \min(m, q + 1)$
5. while $P_k \sqsupset P_q a$
6. $k = k - 1$
7. $\delta(q, a) = k$
8. return δ

Time taken in creating the transition table using δ function for $|\sum|$ characters and finding the maximum k for $P_k \sqsupset P_m$ is $O(m^3|\sum|)$. It can be reduced to $O(m|\sum|)$

Algorithm

FINITE-AUTOMATON-MATCHER(T, δ, m)

1. $n = length(T)$
2. $q = 0$
3. for $i = 1$ to n
4. $\quad q = \delta(q, T[i])$
5. $\quad$ if $q == m$
6. $\quad\quad$ print Pattern occurs with shift $i - m$

Time complexity of the algorithm is $\Theta(n)$.

Knuth-Morris-Pratt algorithm

- This algorithm avoids computing the transition function δ altogether
- Its matching time is $\Theta(n)$
- Its preprocessing time is $\Theta(m)$
- The array π allows us to compute the transition function δ efficiently.
- π is prefix function for a pattern $P[1..m]$ defined as $\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m-1\}$ such that

  $\pi[q] = \max\{k : k < q, P_k \sqsupset P_q\}$

- $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q.
- $P_q \sqsupset T_{s+q}$
- $k = \pi[q]$
- $s = s + (q - k)$

Algorithm I

COMPUTE-PREFIX-FUNCTION(P)

1. $m = length(P)$
2. let $\pi[1..m]$ a new array
3. $\pi[1] = 0$
4. $k = 0$
5. for $q = 2$ to m
6. while $k > 0$ and $P[k+1] \neq P[q]$
7. $k = \pi[k]$
8. if $P[k+1] == P[q]$
9. $k = k + 1$
10. $\pi[q] = k$
11. return π

Algorithm II

KMP-MATCHER(T, P)

1. $n = length(T)$
2. $m = length(P)$
3. π =COMPUTE-PREFIX-FUNCTION(P)
4. $q = 0$
5. for $i = 1$ to n
6. while $q > 0$ and $P[q+1] \neq T[i]$
7. $q = \pi[q]$
8. if $P[q+1] == T[i]$
9. $q = q + 1$
10. if $q == m$
11. print Pattern occurs with shift $i - m$
12. $q = \pi[q]$

Test II ADA, IT 5^{th} *Sem*, Dated: 07^{th} *Nov*, 2019

Even numbered students will attempt even numbered questions and odd numbered students will attempt odd numbered questions.

Q1. Prove correctness of Huffman Code Algorithm

Q2. Prove the correctness of Bellman Ford algorithm for single source shortest path.

Q3. Explain Dijkstra algorithm with the help of an example.

Q4. Explain Kruskul algorithm with the help of an example.

Test II ADA, CSE 5^{th} *Sem*, Dated: 07^{th} *Nov*, 2019

Even numbered students will attempt even numbered questions and odd numbered students will attempt odd numbered questions.

Q1. Prove the correctness of Dijkstra's algorithm for single source shortest path.

Q2. Prove the correctness of Kruskul's algorithm for minimum spanning tree.

Q3. Given ten activities along with their start and finish time as $(A_1, 5, 6), (A_2, 3, 6), (A_3, 1, 4), (A_4, 5, 7), (A_5, 5, 8), (A_6, 7, 10), (A_7, 8, 9), (A_8, 9, 11), (A_9, 8, 11), (A_{10}, 10, 12)$. Compute a schedule where the largest numbers of activities take place.

Q4. Find optimal scheduling and total incurred penalty for following given problem: $a_i = (1, 2, 3, 4, 5, 6, 7)$, $d_i = (4, 2, 4, 3, 1, 4, 6)$, $w_i = (70, 60, 50, 40, 30, 20, 10)$.

ज्योतिर्वृणीत तमसो विजानन

Thank you

Please send your feedback or any queries to akyadav1@amity.edu, akyadav@akyadav.in or contact me on +91 9911375598