Prepared by,

Chetan Shetty
Assistant Professor
Department of CSE
MSRIT
Bangalore

UNIT – 1

INTRODUCTION

1. Some Representative Problems
   - A First Problem: Stable Matching:
     - The Problem
     - Designing the Algorithm
     - Analyzing the Algorithm
     - Extensions
2. Five Representative Problems:
   - Interval Scheduling
   - Weighted Interval Scheduling
   - Bipartite Matching
   - Independent Set
   - Competitive Facility Location
3. Computational Tractability:
   - Some Initial Attempts at Defining Efficiency
   - Worst-Case Running Times and Brute-Force Search
   - Polynomial Time as a Definition of Efficiency
4. Asymptotic Order of Growth:
   - Properties of Asymptotic Growth Rates
   - Asymptotic Bounds for Some Common Functions
5. Implementing the Stable Matching Algorithm
   - Using Lists and Arrays: Arrays and Lists,
   - Implementing the Stable Matching Algorithm
6. A Survey of Common Running Times:
   - Linear Time
   - $O(n \log n)$ Time
   - Quadratic Time
   - Cubic Time
   - $O(nk)$ Time
   - Beyond Polynomial Time
   - Sub linear Time.

UNIT 1

INTRODUCTION

1. A first problem: Stable Matching

1.1 The problem:

- Designing a college admission process or job recruiting process that is self-enforcing.
- All juniors in college majoring in computer science begin applying to companies for summer internships.
- Application process is the interplay between two different types of parties.
  1. Companies (the employers)
  2. Students (the applicants)
- Each applicant has a preference ordering on companies and each company forms a preference ordering on its applicants.
- Based on these preferences, companies extend offers to some of their applicants, applicants choose which of their offers to accept.
- Gale and Shapely considered the sorts of things that could start going wrong with the process.
  1. "**Raj**" accepted job at company "**CluNet**".
  2. **"WebExodus"** offers job to **"Raj"**.
  3. **"Raj"** now prefers **"WebExodus"** and rejects **"CluNet".**
  4. **"Kiran"** gets an offer from **"CluNet"**.
  5. **"Kiran"** already had accepted offer from **"BabelSoft"**.
  6. **"Kiran**" accepts offer from **"CluNet"** and rejects **"BabelSoft"**.
  7. **"Deepa"** who has accepted the offer from **"BabelSoft"**calls up**"WebExodus"** to join them (i.e. she preferred WebExodus over BabelSoft.
  8. **"WebExodus"** rejects **"Raj"** and accept **"Deepa"** (i.e. WebExodus preferred Deepa over Raj).
- Situation like this creates chaos and both applicants and employers endup unhappy with the process as well as outcome as the process is not self-enforcing and people are not allowed to act in their self-interest.

- According to Gale and Shapley: Given a set of preferences among employers and applicants, we can assign applicants to employers so that for every employer E, and every applicant A who is not scheduled to work for E, at least one of the following two things should hold:
  1. "E" prefers every one of its accepted applicants to "A".
  2. "A" prefers her current situation over working for employer "E".

  If this holds, the outcome is stable.
  - Individual self-interest will prevent any applicant/employer deal from being made behind the scene.

1.2. Formulating the problem:

- Each applicant is looking for a single company. Each company is looking for many applicants. Each applicant does not typically apply to every company.
- Each of **napplicants** applies to each of **ncompanies** and each company wants to accept a single applicant.

  (OR)
- **n**men and **n** women can end up getting married, in this case everyone is seeking to be paired with exactly one individual of opposite gender.
  - M is a set of n men, M={m_1,m_2,……,m_n}
  - W is a set of n women, W={w_1,w_2,……w_n}
  - M*W, is the set of all possible ordered pairs of form (m,w), where mϵM and wϵW

- **Matching:** A matching "S" is a set of ordered pairs, each from M*W, with the property that each member of M and each member of W appears in at most one pair in **S**.

Perfect matching

- A perfect matching S^1 is a matching with the property that each member of M and each member of W appears in exactly one pair in S^1.
- A perfect match is a way of pairing men with the women in such a way that everyone ends up married to somebody and nobody is married to more than one person. (i.e. neither singlehood nor polygamy).

Instability:

- Say there are 2 pairs (m,w) and (m^1,w^1) in S with property that
  - m prefers w^1 to w
  - w^1 prefers m to m1

The pair (m,w^1) is an instability with respect to S: (m,w^1) does not belong to S.

- Our goal is a set of marriages with no instabilities. A matching S is stable if:
  1. **It is perfect.**
  2. **There is no instability with respect to S.**

- **Example 1:**

  We have a set of two men, {m, m^1} and a set of two women {w, w^1}. The preference lists are:

  m prefers w to w^1
  m^1 prefers w to w^1
  w prefers m to m^1
  w^1 prefers m to m^1

  - There is a unique stable matching, consisting of pairs (m,w) and (m^1,w^1).
  - (m^1,w) and (m,w^1) would not be a stable match, because the pair (m,w) would form an instability with respect to this matching.

- **Example 2:**
  m prefers w to w^1
  m^1 prefers w^1 to w
  w prefers m^1 to m
  w^1 prefers m to m^1

  - (m,w) and (m^1,w^1) is stable, because both men are happy as neither would leave their matched partners.
  - (m^1,w) and (m,w^1) is stable as both women are happy.
  - So its possible for an instance to have more than one stable matching.

1.3 Designing the Algorithm:

Basic steps:

1. Initially, everyone is unmarried.
   - If an unmarried man **m** chooses woman **w** who ranks highest on his preference list and proposes her.
   - A mam **m^1** whom **w** prefers, **w** may or may not receive a proposal from **m^1.**

- So **w** prefers to go to an intermediate state (**m**,**w**).

2. Suppose we are now at a state in which some men and women are engaged and some of them are not engaged.
   - An arbitrary free man **m** chooses the highest ranks woman **w** and proposes her.
   - If **w** is free, then **m** and **w** become engaged.
   - Otherwise, **w** is already engaged to some other man $\mathbf{m^1}$ i.e. she determines which of **m** or $\mathbf{m^1}$ ranks higher on her preference list.

3. Finally algorithm will terminate when no one is free.

Algorithm

Initially all mϵM and wϵW are free.
While there is a man **m** who is free and hasn't proposed to every woman
Choose such a man **m**

Let **w** be the highest ranked woman in **m**'s preference list to whom **m** has not yet proposed.

If **w** is free then
(**m**,**w**) become engaged.
else
w is currently engaged to $\mathbf{m^1}$

if**w** prefers $\mathbf{m^1}$ to **m** then
m remains free
Else
w prefers **m** to $\mathbf{m^1}$
(**m**,**w**) become engaged.
$\mathbf{m^1}$ becomes free.
End if
End if
End while
Return the set S of engaged pair.

1.4 Analyzing the Algorithm

1. **w remains engaged from the point at which she receives her first proposal, and a sequence of partners to which she is engaged gets better and better(in terms of her preference list).**

2. **The sequence of women to whom m proposes gets worse (in terms of his preference list).**

3. **The G-S algorithm terminates after at most n^2 iterations of the while loop.**

PROOF:

- Let **P(t)** denote the set of pairs(m,w) such that **m** has proposed to **w** by the end of iteration **t** .
- For all **t**,the size of **P(t+1)** is strictly greater than the size of **P(t)**.
- There are only n^2 possible pairs of men and women in total. So the value of **P(.)** can increase at most n^2 times over the course of the algorithm.
- So it follows that there can be at most n^2iterations.

- **Two things are important to note :**
  1. There are executions of the algorithm that can involve close to n^2 iterations.
  2. There are many quantities that would not have worked well as a progress measure for algorithm,since they need not strictly increase in each iteration.
- **Example:** Number of free individuals could remain constant from one iteration to the next, as could the number of engaged pairs. So these quantities could not be used directly in giving an upper bound on maximum possible number of iterations.

4. **If m is free at some point in the execution of the algorithm, then there is a woman to whom he has not yet proposed.**

PROOF

- Suppose there comes a point when **m** is free but has already proposed to every woman.
- Then by 1, each of 'n' women is engaged at this point in time.
- Since the set of engage pairs forms a matching, there must also be 'n' engaged men at this point in time.
- But there are only 'n' men total,and **m** in not engaged,so this is a contradiction.

5. **The set S returned at termination is a perfect matching.**

PROOF

- A set of engaged pairs always forms a matching.
- Let us suppose that the algorithm terminates with a free man **m**.
- At termination, it must be the case that **m** had already proposed to every woman, for otherwise the while loop would not have exited.
- But this contradicts (4) which says that there cannot be a free man who has proposed to every woman.

6. **Consider an execution of the G-S algorithm that returns a set of pairs S. the set S is a stable matching.**

PROOF

- We have seen from (5) that S is a perfect matching.
- To prove S is a stable matching, we will assume that there is an instability with respect to s and obtain a contradiction.
- Instability would involve 2 pairs(m,w) and(m^1,w^1),in s with the properties that
  1. **m** prefers $\mathbf{w^1}$ to **w**,and
  2. $\mathbf{w^1}$ prefers **m** to $\mathbf{m^1}$
- In the execution of algorithm that produced S,**m'**s last proposal was,by definition to **w**.
- Did **m** propose to $\mathbf{w^1}$ at some earlier point in the execution point in the execution?

- If he didn't then **w** is higher on **m**'s preference list than $\mathbf{w^1}$,contradicting our assumption that **m** prefers $\mathbf{w^1}$ to **w**.
- If he did, then he was rejected by $\mathbf{w^1}$ in favour of some other man $\mathbf{m^{11}}$,whom $\mathbf{w^1}$ prefers to **m**.
- $\mathbf{m^1}$ is final partner to $\mathbf{w^1}$,so either $\mathbf{m^{11}}=\mathbf{m^1}$ or $\mathbf{w^1}$prefers final partner $\mathbf{m^1}$to $\mathbf{m^{11}}$,either way this contradicts our assumption that $\mathbf{w^1}$prefers **m** to $\mathbf{m^1}$**.**
- It follows that S is a stable matching.

1.5 Extensions

All extensions yield the same matching:

- Uniquely characterize the matching that is obtained and then show that all executions result in matching with this characterization.
- We will show that each man ends up with the best possible partner.
- A woman **w** is a valid partner of man **m** if there is a stable matching that contains the pair(m,w).
- We will say that **w** is the best valid partner of **m** if **w** is a valid partner of **m**,and no woman whom **m** ranks higher than **w** is a valid partner of his.
- We will use best(m) to denote the best valid partner of **m**.
    - Now let S* denote the set of pairs{(m,best(m)):mϵM}.
- Each man ends up with the best possible partner if all men prefer different women.

7. Every execution of the G-S algorithm results in the set S*:

PROOF

- Let us suppose by the way of contradiction, that some execution E of the G-S algorithm results in matching s in which some man is paired with woman who is not his best valid partners.
- Since men propose in decreasing order of preference,this means that some man is rejected by a valid partner during the execution E of the algorithm.
- So consider the first moment during the execution E in which some man, say **m**,is rejected bya valid partner **w**.

- Since men propose in decreasing order of preference,E since this is the first time such a rejection has occurred,it must be that **w** is **m**'s best valid partner best(m).
- The rejection of **m** by **w** may have happened either because
    1. **m** proposed and was turned down by **w**,because of existing engagement.
    2. **w** broke engagement to **m** in favour of better proposal.
- Either way,**w** forms or continues an engagement with a man $\mathbf{m^1}$ whom she prefers to **m**.
- Since **w** is a valid partner of **m**, there exists a stable matching S^1 containing the pair(m,w).
- Since the rejection of **m** by **w** was the first rejection of a man by a valid partner in the execution E,it must be that $\mathbf{m^1}$ has not been rejected by any valid partner at the point in E when he became engaged to **w**.
- Since he proposed indecreasing order of preference, and since $\mathbf{w^1}$ is clearly a valid partner of **m**,it must be that $\mathbf{m^1}$ prefers to **w** to $\mathbf{w^1}$.
- But we have already seen that **w** prefers $\mathbf{m^1}$ to **m**,for in execution E she rejected **m** in favour of $\mathbf{m^1}$.
- Since(m^1,w) does not belong to S^1 ,it follows that (m^1,w) is an instability in S^1.
- This contradicts our claim that S^1 is stable and hence contradicts our initial assumption.

8. In the stable matching S*, each woman is paired with her worst partner with her worst valid partner.

PROOF:

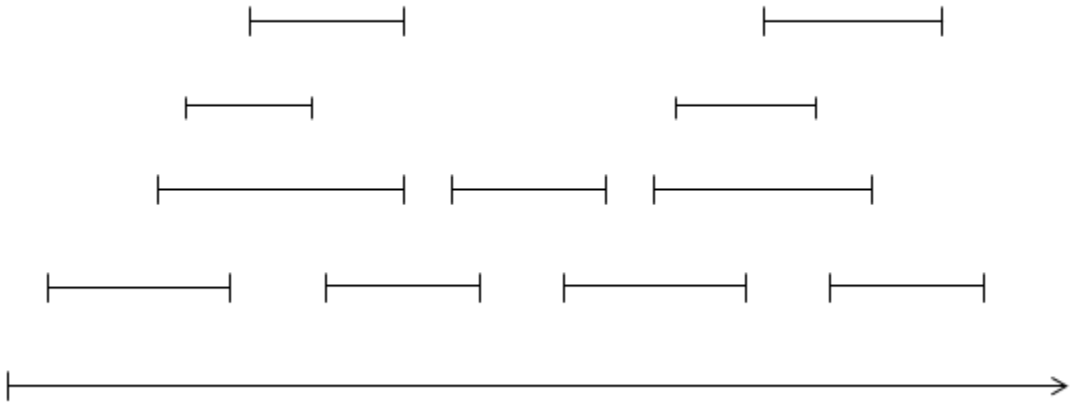
- Suppose there were a pair (m,w) in S* such that **m** is not the worst valid partner to **w**.
- Then there is a stable matching S^1 in which **w** is paired with a man **m** whom she likes less than **m**.
- In S^1, **m** is paired with a woman $\mathbf{w^1}\neq\mathbf{w}$; since **w** is the best valid partner of **m** and $\mathbf{w^1}$ is a valid partner of **m,** we see that m prefers **w** to $\mathbf{w^1}$.
- But from this it follows that (m,w) is an instability in S^1, contradicting the claim that S^1 is stable and hence contradicting our initial assumption.

2.Five Representative Problems:

1. Internal scheduling:

Scheduling problem:

- You have aresource say lecture room and many people request to use the resources for periods of time.
- We will assume that the resource can be used at most one person at a time.
- A scheduler wants to accept a subset of these requests rejecting all others, S_i that the accepted request donot overlap in time.
- The goal is to maximize the number of requests accepted.
- Example:



A single compatible set of size 4, and this is the largest compatible set.

- There will be 'n' requests labeled 1,2,….n, with each request ispacing start time S_i and a finish time f_i, such that $S_i<f_i$ for all i.
- 2 requests i and j are compatible if the requested intervals do not overlap. i.e. either 'i' is for an earlier time interval than request 'j' ($f_i<=S_j$) or otherwise ($f_j<=S_i$).
- A subset A of requests is compatible if all pairs of requests i,jϵA.
- The goal is to select a compatible subset of requests of maximum possible size.
- This problem can be solved by algorithm that orders the set of requests according to certain heuristic and then "greedily" process them, selecting as large compatible subset as it can.

2. **Weighted Interval Scheduling:**
   - In this, each request interval 'i' has an associated value, or weight, "$V_i>0$", we can picture this as amount of money we will make from the i^{th} individual if we schedule his or her request.
   - Goal is to find a compatible subset of intervals of maximum total value.
   - If $V_i=1$ foe each 'i' is simply the basic interval scheduling.
   - Appearance of arbitrary values changes the nature of maximization problem.
   - **Example:** If V_1 exceeds the sum of all other "V_i", then the optimal solution must include interval 1regardless of the configuration of full set of intervals. So any algorithm for this problem must be very sensitive to the values.
   - We employ "dynamic programming" technique that builds up the optimal value overall possible solution in a compact tabular way, that leads to efficient algorithm.

3. **Bipartite Matching :**
   - We can express the concept of "stable matching problem" more generally in terms of graphs, and in order to do this it is useful to define the notation of "bipartite graph".
   - A graph G=(V,E) is bipartite, if its node set "V" can be partitioned into sets X and Y in such a way that every edge has one end in 'X' and the other end in 'Y'.
   - In case of bipartite graphs, the edges are pairs of nodes, so we say that matching in a graph G=(V,E) is a set of edges M proper subset of E with the property that each node appears in at most one edge of M.
   - M is a perfect matching if every node appears in exactly one edge of M
   - For stable matching, consider a bipartite graph G1 with a set 'X' of 'n' men, a set Y of n women, and an edge from every node in 'X' to every node in 'Y'.
   - In stable matching we added preferences, here we add a different source of complexity.
   - There is not necessarily an edge from every $x\epsilon X$ to every $y\epsilon Y$, so the set of possible matching has quite complicated structure.

(OR)

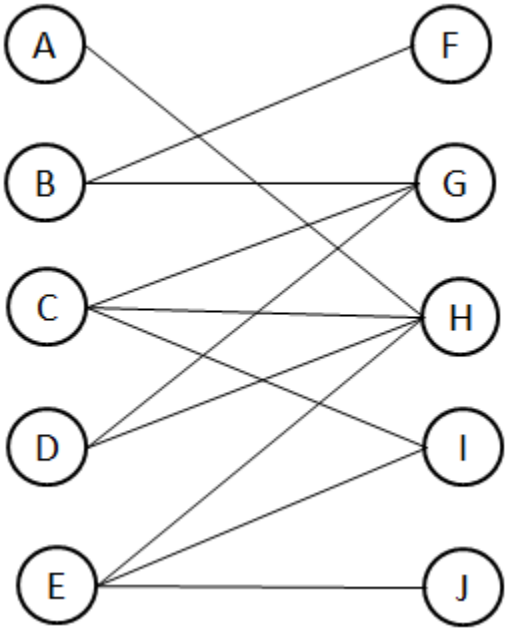
Only certain pairs of men and women are willing to be paired off and figure out how many people in a way that is consistent with this.

- **Example:** X is the set of professors in the department.

  Y is the set of offered courses.

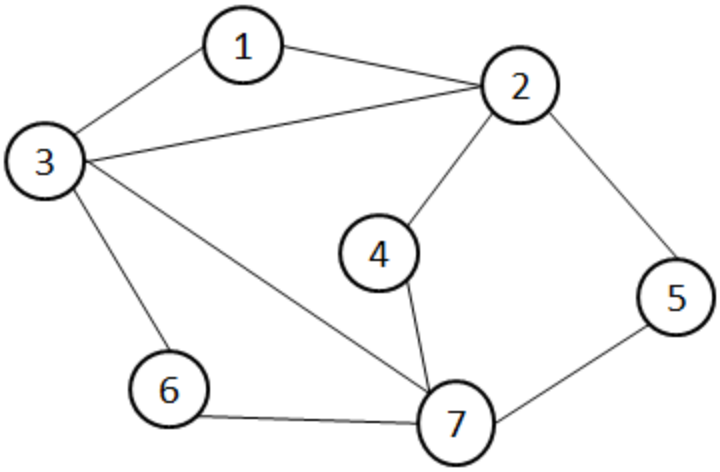
  (X_i, Y_j) is an edge that indicates, professor X_i be of teaching course Y_j.

- A perfect matching consists of an assignment of each professor to a course that he/she can teach, in such a way that every course is covered.
- To solve this problem we have an efficient algorithm called "Augmentation".



Bipartite graph

4. **Independent Set:**
   - Given a graph G=(V,E), we say a set of nodes S which is proper subset of V is independent if no two nodes in S are joined by an edge.
   - Independent set problem is, given a graph G, find an independent set that is as large as possible.



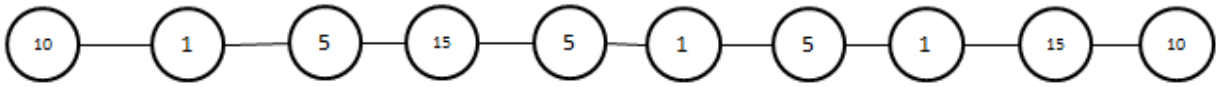
Example: The maximum size of an independent set in the graph is 4. i.e. {1, 4, 5, 6}

- Independent set problem encodes situation in which you have 'n' friends and some pairs don't get along. How large a group of friends can be invited to dinner if you don't want any interpersonal tensions, L would be the largest independent set in the graph whose nodes are your friends, with an edge between each conflicting pair.
- Interval scheduling and bipartite matching are special cases of independent set problem.
- For interval scheduling, define a graph G= (V, E) in which the nodes are the intervals and there is an edge between each pair of them that overlap.
- The independent sets in **G** are then just the compatible subsets of intervals.
- Given a bipartite graph $G^1 = (V^1, E^1)$, the objects being chosen are edges, and the conflicts arise between two edges that share an end.
- So we define a graph G= (V, E) in which the node set V is equal to the edge set E^1 of G^1.
- We define an edge between each pair of elements in V that corresponds to edges of G^1 with a common end.
- We can now check that the independent sets of G are precisely the matchings of G^1.

5. <u>Competitive Facility Location</u>

- Consider 2 large companies that operate café franchise across the country and they are currently competing for market share in a geographical area.
- First "Café Coffee Day" opens a franchise, then "Coffee World" opens a franchise and so on.

- Suppose they deal with zoning regulations that require no two franchises be located too close together, and each is trying to make its locations as convenient as possible.
- The geographical region is divided into "n" zones, labeled 1,2,3……,n. Each zone "i" has a value "b_i", which is the revenue obtained by either of the companies if it opens a franchise there.
- Finally certain pairs of zones (i, j) are adjacent, and local zoning laws prevent two adjacent zones from each containing a franchise, regardless of which company owns them.
- We model these conflicts via a graph G= (V, E), whose "V" is a set of zones, and (i, j) is an edge in E if the zones "i" and "j" are adjacent.
- Zoning requirement says that the full set of franchises opened must form an independent set in **G**.



- Our game consists of 2 players, P1 and P2, alternately selecting nodes in G, with P1 moving first.
- The set of all selected nodes must form an independent set in G.
- P2 target bound is B=20. Then P2 has a winning strategy.
- If B=25, then P2 does not have a winning strategy.
- If there is a strategy for P2 so that no matter how P1 plays, P2 will be able to select a set of nodes with a total value of atleast B. We call this an instance of "competitive location problem".

****

CHAPTER 2

COMPUTATIONAL TRACTABILITY

- Primary focus will be on the efficiency in running time. i.e. we want algorithms to run quickly.
- Algorithms must be efficient in their use of other resources as well. i.e space used by an algorithm.

Some initial attempts of defining Efficiency

- **Definition:** An algorithm is efficient if, when implemented, it runs quickly on real input instances.
- **Disadvantages in definition**:
    1. Omission of where we implement algorithm and howwell we implement algorithm.
        a. Even bad algorithm will run quickly when applied to small test cases on extremely fast processors.
        b. Even good algorithm runs slowly when they are coded sloppily.
    2. We don't know the full range of input instances that will be encountered in practice.
    3. Does not consider how well or badly an algorithm may scale as problem sizes grow to unexpected levels.
- Situation where 2 very different algorithms may perform comparably on input size 100; multiply the input size tenfold and one will still run quickly while other consumes a huge amount of time.
- We need a concrete definition of efficiency that is
    - Platform independent.
    - Instance independent
    - Predictive values w.r.t increasing input size.
- Example: Stable matching problem.
    - N → Input size (total size of preference lists)
    - n→ Number of men and women.
    - There are 2n preference lists, each of length "n".
    - $N=2n^2$ (2 preference lists each of length n)

- **Worst-Case Running times & Brute-Force Search**
  - In analysis of worst-case running time, we will look for a bound on largest possible running time the algorithm could have over all inputs of a given size "N" & see how it scales with "N".
  - Brute-Force search can tell us whether a running time bound is impressive or weak over search space.
  - Example: Stable matching problem.
  - When the size of a stable matching input instance is relatively small, the search spaceit defines is enormous (there are n! possible perfect matching between n men & n women), and we need to find a matching that is stable.
  - Brute force algorithm for this problem would plow through all perfect matchings by enumeration, checking each to see if it is stable.
  - In this we need to spend time proportional only to N in finding a stable matching from this stupendously large space of possibilities.

- **Proposed definition of Efficiency:** An algorithm is efficient if it achieves qualitatively better worst-case performance at an analytical level, than brute-force search.
- **Note:** It is very hard to express full range of instances that arise in practice. So how a random input should be generate, as same algorithm can perform very well on one class of random inputs & very poorly on another.

Polynomial time as definition of Efficiency.

- Search spaces for natural combination problems tend to grow exponentially in the size N of the input.
- If input size increases by 1, the number of possibilities increases multiplicatively.
- Algorithm for such a problem to have a better scaling property when input size increases by a constant factor (a factor of 2) the algorithm should only slow down by some constant factor 'c'.
- Arithmetically we can formulate this scaling property as say: There are absolute constants c>0 & d>0 so that on every input instance of size "N" its running time is bounded by cN^d primitive computational steps.

- If running time bound holds, for some c & d, then we say that the algorithm has a polynomial running time.
- If input size increases from N to 2N, the bound of running time increases from cN^d to $c(2N)^d = c.2d.N^d$, which is a slow-down by a factor of 2^d. Since d is a constant so is 2^d.
- Lower-degree polynomials exhibit better scaling behavior than higher-degree polynomials.

Proposed definition of Efficiency:

- An algorithm is efficient if it has a polynomial running time.
- Problems for which polynomial-time algorithm exists almost invariably turnout to have algorithms with running time proportional to very moderate growing polynomials like n, n log n, n^2 or n^3.
- Exponential functions 2^n& n! grow fast that their values become astronomically large even for small values of n.
- Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes.

Asymptotic order of growth

1. **Asymptotic upper bounds:**
   - Let T(n) be a function- the worst case running time of a certain algorithm on an input size "n".
   - Another function f(n), we say that T(n) is O(f(n)) if, for sufficiently large n, the function T(n) is bounded above by a constant multiple of f(n).

     (OR)

     T(n)= O(f(n))
   - T(n) is O(f(n)) if there exist constant c > 0 and n_0>= 0 so that for all n >= n_0, we have T(n) <c.f(n). In this case, we will say that T is asymptotically upper bounded by f.
   - **Example:** Consider an algorithm whose running time is of the form $T(n)=pn^2+qn+r$ for positive constant p,q and r.
     - We would like to claim that any such function is $O(n^2)$.
     - $T(n)=pn^2+qn+r <= pn^2+qn^2+rn^2 =(p+q+r)n^2$ for all n>=1
     - This inequality is exactly is exactly what the definition of O(.) requires:

       $T(n)<=c.n^2$ where c=p+q+r

- O(.) expresses only an upper bound.
- So we claim that the function $T(n)=pn^2+qn+r$ is $O(n^2)$, it is also correct to say that its $O(n^2)$.

2. **Asymptotic Lower bounds**:
   - For arbitrarily large input sizes n, the function T(n) is at least a constant multiple of some specific f(n).
   - Then we say that T(n) is Ω(f(n)) if there exist constants c>0 and n_0>0. So that for all n>=n_0, we have T(n)>= c.f(n)
   - So we will refer to T in this case as being asymptotically lower bounded by f.
   - The constant "c" must be fixed, independent of "n".
   - Example: $T(n)=pn^2+qn+r$, where p,q and rare positive constants.
   - We need to reduce the size of T(n) until it looks like a constant times n^2.
   - $T(n)=pn^2+qn+r>=pn^2$ for all n>=0
   - So $T(n)=pn^2+qn+r$ is Ω(n), since $T(n)>=pn^2>=pn$.

3. **Asymptotically Tight bound:**
   - If a function T(n) is both O(f(n)) and Ω(f(n)), we say that T(n) is θ(f(n)). In this case, we say that f(n) is asymptotically tight bound for T(n).
   - So for example our analysis shows that $T(n)= pn^2+qn+r$ is $\theta(n^2)$
   - Asymptotically tight bound characterize the worst-case performance of an algorithm precisely up to constant factor.

Properties of Asymptotic Growth rates

1. **Let f and g be 2 functions that $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)$ exists and is equal to some number c>0. Then f(n)=θ(g(n))**

Proof:

- We will use the fact that the limit exists and is positive to show that f(n)= O(g(n)) and f(n)= Ω(g(n)) as required by the definition of θ(.)

Since$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)$ c >0

- It follows from the definition of limit that there is some n0 beyond which the ration is always between ½ c and 2c.
- Thus f(n) <= 2c.g(n) for all n>=n_0, which implies that f(n)=O(g(n)) and f(n) >= ½ c.g(n) for all n>= n_0, which implies that f(n)= Ω(g(n))
- Hence f(n)=θ(g(n)).

2. **<u>Transitivity</u>:If a function f is asymptotically upper bounded by a function g and if g in turn is asymptotically upper bounded by a function h, then f is asymptotically upper bounded by h.**

2.1. If f=O(g) and g=O(h), then f=O(h)

<u>Proof</u>:

f(n)= O(g(n)),

$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)$ c_1=0 or c_1>0 for all n_0>0 → (1)

g(n)= O(f(n)),

$\lim_{n\to\infty}\left(\frac{g(n)}{h(n)}\right)$ c_2=0 or c_2>0 for all n_0^1>0 → (2)

Multiply (1) and (2)

$\lim_{n\to\infty}\frac{f(n)}{g(n)}\times\frac{g(n)}{h(n)}$ c_1c_2=0 or c_1c_2>0

$\lim_{n\to\infty}\left(\frac{f(n)}{h(n)}\right)$ or f(n)= c.h(n)

Hence f(n) = O(h(n))

2.2. If f=Ω(g) and g= Ω (h), then f= Ω (h)

<u>Proof:</u>

f(n)= Ω (g(n)),

$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)$ c_1=∞ or c_1>0 for all n_0>0 → (1)

g(n)= Ω (f(n)),

$\lim_{n\to\infty}\left(\frac{g(n)}{h(n)}\right)$ $c_2=\infty$ or $c_2>0$ for all $n_0^1>0$ → (2)

Multiply (1) and (2)

$\lim_{n\to\infty}\frac{f(n)}{g(n)}\times\frac{g(n)}{h(n)}$ $c_1c_2=\infty$ or $c_1c_2>0$

$\lim_{n\to\infty}\left(\frac{f(n)}{h(n)}\right)$ or f(n)= Ω (h(n))

2.3. If f=θ(g) and g= θ (h), then f= θ (h)

Proof:

f(n)= θ (g(n)),

$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)$ $c_1>0$ for all $n_0>0$ → (1)

g(n)= θ (f(n)),

$\lim_{n\to\infty}\left(\frac{g(n)}{h(n)}\right)$ $c_2>0$ for all $n_0^1>0$ → (2)

Multiply (1) and (2)

$\lim_{n\to\infty}\frac{f(n)}{g(n)}\times\frac{g(n)}{h(n)}$ $c_1c_2>0$

$\lim_{n\to\infty}\left(\frac{f(n)}{h(n)}\right)$ or f(n)= c.h(n)

Hence f(n) = θ (h(n))

2.4. Suppose that f and g are two functions such that for some other function h we have f=O(h) and g=O(h). Then f+g= O(h)

Proof:

- We are given that for some constants c and n_0, we have f(n) <= c_1.h(n) for all n>=n_0^1
- So consider any number n that is at least as large as both n_0 and n_0^1.

- We have,

$f(n)+g(n) <= c.h(n) + c_1.h(n)$

$f(n)+g(n)<=(c+c_1).h(n)$ for all $n>= \max(n_0, n_0^1)$

- Which is exactly what is required for showing that f+g=O(h).

2.5. Let k be a fixed constant, and let f_1, f_2, f_3,…..,f_k and h be functions such that f_i=0(h) for all i. Then $f_1+f_2+f_3+……+f_k$ = O(h)

- Consequence of (2.4)
- We are analyzing an algorithm with two high level parts and it is easy to show that one of the two parts is slower than the other.
- The running time of the whole algorithm is asymptotically comparable to the running time of the slow part.
- Since overall running time is a sum of 2 functions, result on asymptotic bounds for sums of functions are directly relevant.

2.6. Suppose that f and g are two functions such that g=O(f). Then f+g=θ(f). in other words, f is an asymptotically tight bound for the combined function f+g.

Proof:

g(n) = O(f(n)

g(n)<= f(n)

or

f(n)>=g(n) →Ω

f(n) + g(n) >= Ω(f(n)) for all n>=0

i.e. f(n) + g(n) >=f(n)

We have already shown that f+g=O(f)

Hence if functions belongs to O(n) and Ω(n), then it has to belong to θ(n)

2.7. Let f be a polynomial of degree k, in which the coefficient a_k is positive. Then f= O(n^k)

Proof:

$f(n)= a_kn^k + a_{k-1}n^{k-1} +…… +a_1n + a_0$

The highest order in polynomial that dominates as n→∞

Choose n_0=1 and $c=|a_k| + |a_{k-1}| + …..+ |a_1| + |a_0|$

We need to show that $f(n)=c.n^k$ for all n>=1.

We have for every n>=1, we replace coefficients with absolute value.

$f(n)<=|a_k|n^k + |a_{k-1}|n^{k-1} +.....+ |a_1|n + |a_0|$

Replace lower power of n with higher powers of n^k.

$f(n)<= |a_k|n^k + |a_{k-1}|n^k +.....+ |a_1|n^k + |a_0|n^k$

$f(n)<= (|a_k| + |a_{k-1}| +.....+ |a_1| + |a_0|)n^k$

$f(n)<=c.n^k$

or

$f(n)=O(n^k)$

2.8. For every b>1 and every x>0, we have $\log_b n = O(n^x)$

Proof:

One can translate directly between logarithms of different bases using identity.

$\log_a n = \frac{\log_b n}{\log_b a}$

$\log_a n = \frac{1}{\log_b a} * \log_b n$

$\log_a n = C. \log_b n$

$\log_a n = \theta(\log_b n)$

Note: Base of the algorithm is not important when writing bounds using asymptotic notation.

2.9. For every r>1 and every d>0, we have $n^d = O(r^n)$

- For different bases r>s>1, it is never the case that $r^n=\theta(s^n)$ [unlike in log].
- This would require that for some constant c>0, we would have $r^n<=c.s^n$ for all sufficiently large **n**.

$r^n<=c.s^n$

Divide by s^n

$r^n/s^n<=c$ for all sufficiently large "n"

(or)

$(r/s)^n<= c$

Since r>s, the expression $(r/s)^n$ tends to infinity with "n", and so it cannot possibly remain bounded by a fixed constant c.

A Survey of Common Running Times:

1. Linear time:

- An algorithm that runs in O(n), or linear, time has a very natural property: its running time is at most a constant factor times the size of the input.
- Example 1: Computing the maximum
  - Numbers are provided as input in either list or array.
  - Each time we encounter a number a_i, we check whether a_i is larger than current maximum, and if so we update the estimate to a_i.

- **Algorithm:**

  max=a_1
  fori=2 to n
  ifa_i>max then
  Set max=a_i
  End if
  End for
- f(n)=O(n)

Example 2: Merging two sorted lists

- Suppose we are given two lists of n numbers each, a_1,a_2,a_3, …… , a_n and b_1, b_2, b_3,…… , b_n and each is arranged in ascending order
- Merge them into a single list c_1, c_2, c_3, …… c_{2n} that is also arranged in ascending order.

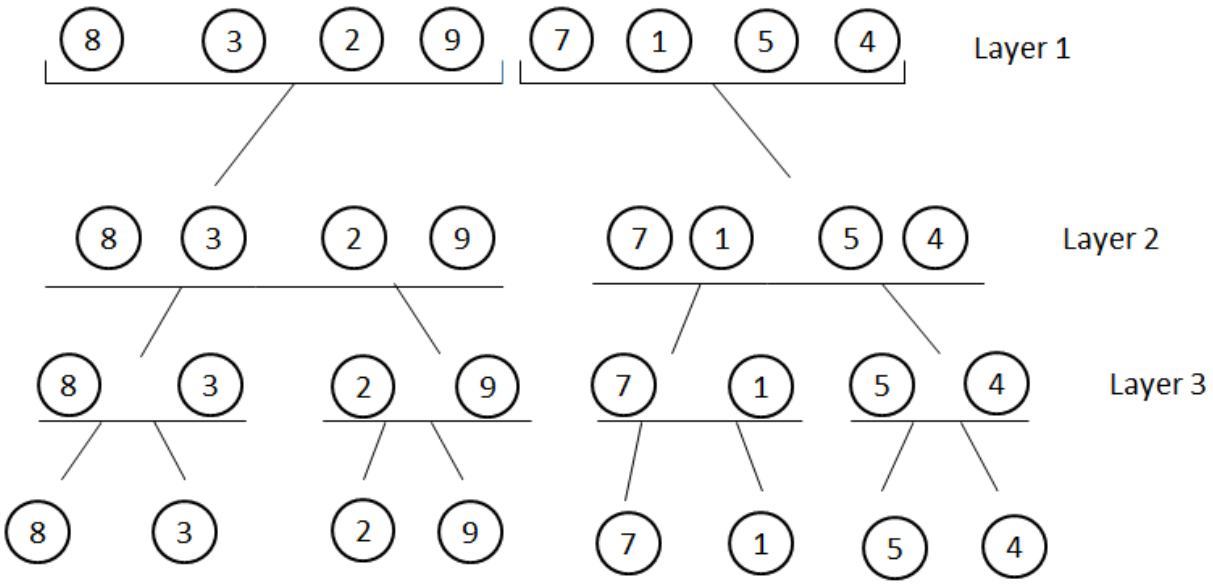
- **Algorithm:**

  To merge sorted lists A=a_1, a_2, ….., a_n and B= b_1, b_2,….. b_n,
  Maintain a current pointer into each list, initialized to point to the front elements
  While both lists are non-empty
  Let a_i and b_j be the elements pointed to by the current pointer.
  Append the smaller of these two to the output list.
  Advance the current pointer in the list from which the smaller element was selected.
  End while
  Once one list is empty, append the remainder of the other list to the output.

- Order of growth is O(n).

2. O(n log n) time

Mergesort: Algorithm that splits its inputs into 2 equal sized pieces, solve each piece recursively & combines the set in linear time.



Note: We are not considering 4th layer because we are not doing any processing here, we split and we go back.

- Layer 1 has 8 elements, so as layer 2 and layer 3.
  8+8+8=24
- 3 levels → 8 elements per level.
- So total operations = 8*3= n* ?
- $8 \leftrightarrow 3$
- $\log_2 8 \leftrightarrow 3$
- $\log_2 2^3 \leftrightarrow 3$
- $3\log_2 2 \leftrightarrow 3$
- $3*1 \leftrightarrow 3$
- Or $\log_2 8 \leftrightarrow 3$
- i.e. $8*\log_2 8$
- **O(n log n)**

3. Quadratic time

- **Example:** Suppose given n points in a plane, each specified by (x,y) coordinates, we would like to find the pair of points that are closest together.

Solution: The number of pairs of points can be calculated by using formula $\frac{n(n-1)}{2}$

The distance between point (x_i,y_i) and (x_j,y_j) can be computed by the formula,

d= $\sqrt{(xi-xj)^2+(yi-yj)^2}$ in constant time.

So total number of comparison $\frac{n(n-1)}{2}$= $O(n^2)$

- **Algorithm:**

For each input point (x_i,y_i)

For each output point (x_j,y_j)

Compute distance d= $\sqrt{(xi-xj)^2+(yi-yj)^2}$

If d is less than the current min, update min to d.

End For

End For

4. Cubic Time

- Sets of nested loop often leads to algorithm that run in $O(n^3)$ time.
- **Example:** Given sets S_1, S_2,….., S_n, each of which is a subset of {1,2,3,…..,n} and we would like to know whether some pair of these sets is disjoint.

- **Algorithm:**

For each set S_i

For each set S_j

For each element P of S_i

Determine whether P also belongs to S_j

End For

If no element of S_i belongs to S_j then

Report that S_i and S_j are disjoint.

End if

End For

End For

- Each set has maximum size O(n), so the innermost loop takes time O(n), looping over set S_j involves O(n) iterations and looping over set S_i involves O(n).
- Multiplying these three factors of "n" together we get the running time of $O(n^3)$

5. $O(n^k)$ time

- **Example:** Finding independent sets in a graph.
- For some fixed constant 'K', we would like to know if given n-node input graph 'G' has an independent set of size 'K'

Algorithm:

For each subset S of K nodes
Check whether S constitutes an independent set
If S is an independent set then
Stop and declare success
End if
End for
If no K-node independent set was found then
Declare failure
End if

Running time:

- For running time consider "Total number of K-element subsets in n-element set.
- $\binom{n}{k} \frac{n(n-1)(n-2).....(n-k+1)}{k(k-1)(k-2)(k-3).....(2)(1)} \leq \frac{n^k}{k!}$
- Since we treat 'K' as constant, this quantity is $O(n^k)$. So outer loop will run for $O(n^k)$ iterations as it tries all K-node subsets of the 'n' nodes of the graph.
- Inside loop we need to test a given set 'S' of 'K' nodes constitutes an independent set.
- Definition of independent set says that we need to check for each pair of nodes, whether there is an edge or not.

  It is same as searching over pair of points which took $O(k^2)$ time.

Therefore, the total running time is $O(k^2n^k)$.

Since we are treating 'K' as a constant we write the running time as $O(n^k)$.

6. Beyond polynomial time

- Two kinds of bounds that come up very frequently 2^n and n!
- **Example:** Given a graph we want to find an independent set of maximum size.
- **Algorithm**

  For each subset S of nodes

  Check whether S constitutes an independent set

  If S is larger independent set than the largest seen so far then,

  Record the size of 'S' as the current maximum

  End if

  End for
- We iterate over all subsets of graph. So the total number of subsets of n-element set is 2^n. → (1)
- Outer loop executes 2^n iterations as it tries all subsets.
- Inside loop checks all pairs from set 'S' that can be as large as 'n' nodes will take $O(n^2)$ time. → (2)
- Multiply (1) and (2)

  $O(n^2 2^n)$ is the total running time for this algorithm.

⇨**n! grows rapidly than 2^n**

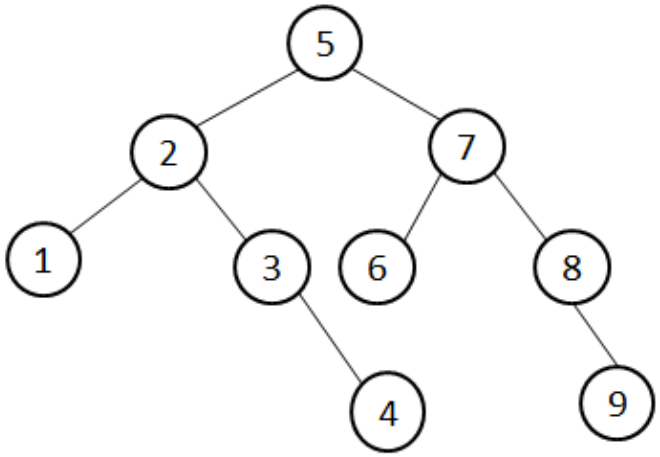
- n! is the number of ways to match up 'n' items with 'n' other items.
- Example : Number of possible perfect matchings of 'n' men and 'n' women in an instance of the stable matching problem.
- First man will always have 'n' choices, once he selects a women from other set, the second man will select from (n-1) women left.
- Multiplying all these choices we get

  n(n-1)(n-2)…….(2)(1) = n!

7. Sub linear time

- Cases where running times are asymptotically smaller than linear.
- Goal is to minimize the amount of query that must be done.
- Example: Binary Search

To search 1:



- To search 1 →Firsti will choose 5, then 2 and finally 1, i.e. i will do 3 searches to get an element.

  The total number of elements is 9, searches 3.

  9⇔3 need to establish a relationship between 9 & 3.

  $\log_2 9 = 3$

  $\log_2 9 \approx \log_2 8$

  $\log_2 8 = 3$

  $\log_2 2^3 = 3$

  $3\log_2 2 = 3$

  3=3

  Or $\log_2 9 = 3$

  i.e. 9 is the number of elements so replace 9 by n.

  $O(\log_2 n)$ is the worst case running time for binary search.

- When 22 is being searched

  n= 9 No. of searches =4

  4↔9

  1+3↔9

  $1+\log_2 9 = 1+ \log_2 2^3$

  $1+\log_2 9 = 1+ 3\log_2 2$

  $1+\log_2 9 = 1+ 3$

  $1+\log_2 9 = 4$

  Or $1 + \log_2 n$

  i.e. $O(\log_2 n)$ is the worst case running time.

- For successful search it is either log n or less than that.

****