1. Matching Problem

   (a) Definitions

      i. Stable Situation: Self interest itself prevents offers from being retracted and redirected.
      ii. **Stable Outcome**: Individual self-interest will prevent any applicant/employer from being retracted and redirected
      iii. Matching: is a set of ordered pairs
      iv. **Perfect Matching**: is a matching with the property that each member of M and each member of W appears in exactly one pair in the matching.
      v. **Instability**: is pair (m, w') respect to S, such that $(m, w') \notin S$, but each if m and w' prefers the other to their partner in S.(Instability is a pair that not in the stable matching but it should be in)
      vi. **Stable Matching**: A matching if it is perfect and no instability with respect to S.
      vii. Valid Partner: w is a valid partner of m if there is a stable matching S and the pair (m, w) belongs to it.
      viii. Best Valid Partner: 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. $w = best(m)$
      ix. Worst Valid Partner: m is the Worst Valid Partner of w if m is a valid partner of w, and no man whom w ranks lower than m is a valid partner of hers.

   (b) Matching Algorithm: Gale-Shapley algorithm

```
Initially all $m \in M$ and $w \in W$ are free;
while there is a man m who is free and hasn't proposed to every woman do
    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 m';
        if w prefers m' to m then
            m remains free;
        else
            w prefers m to m';
            (m,w)become engaged;
            m' becomes free
        end
    end
end
```

   (c) Analyzing the G-S algorithm

      i. Assumption: There are equal number of man and woman, and each person only pair with one of person on his/her preference list.
      ii. It is possible to have more than on stable matching
      iii. The following hold when man propose and assume that the preference list is complete and correct
         A. w remains engaged after she receives her first proposal, the partners whom she is engaged gets better and better in terms of her preference list

B. m may alternate between being free and being engaged; the sequence of women to whom m proposes get worse and worse.

iv. The running time of G-S: The G-S algorithm terminates after at most n^2 iterations of the While loop.
Proof
Let $P(t)$ be the set of pairs (m,w) such that m has proposed to w by the end of iteration t. It is clear that the size of $P(t+1)$ is strictly greater than the size of $P(t)$. Since the number of man and the number of woman are n, there are only n^2 possible pairs of man and woman in total. Thus, the value of $P(\cdot)$ ca1u increase at most n^2 times

v. "The Set S returned at the termination of the algorithm is in fact a perfect matching" ≡ "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
1. the only way for the While loop to exit is fro there to be no free man.
2.

2. Problem: "Finding a_i has Property P_r" $\leftrightarrow P(a_i) = T$

   (a) Principle: Start at $a_i = a_1$. Check $P(a_i)$. If $P(a_i) = T$, then $a_{out} = a_1$; else $a_i = a_2$

   (b) Algorithm

```
Start with $i = 0$, $a_{out} = \emptyset$;
while $i \leq n$ and $a_{out} = \emptyset$ do
    if $P(a_{i+1}) = T$, then
        $a_{out} = a_{i+1}$
    else
        $i = i + 1$
    end
end
```

   (c) Analysis:

3. Data Structure

   (a) **Array**
   1-D
   2-D

   (b) **List**
   Linked List
   Double Linked List

   (c) **Queue**: FIFO

   (d) **Stack**: LIFO

4. Efficiency of Algorithm

   (a) Definition:
   An algorithm is efficient if it has polynomial running time

   (b) What is polynomial running time?
   The algorithm has a polynomial running time if

$$\exists c > 0,\ d > 0,\ \textit{so that}\ \forall N\ \textit{size, its running time is nounded by}\ cN^d\ \textit{primitive operations}$$

In the other words, when the input size increase by a constant factor, the algorithm should only slow down by some constant factor C

(c) What is primitive operations?
Primitive operations are basic computations performed by an algorithm

5. Asymptotic Order of Growth: the framework for analyze bound $f(n)$ on the running time of algorithm: a way that is insensitive to constant factors and low order terms

   (a) The n is the number of step in the algorithm.the number of step in each loop or the of the whole execution?

   (b) It is very hard to get explicit form of $f(n)$, and it is not necessary. means get precise bound is exhausting activity.

   (c) The goal is to analyze the broad class of algorithm with the similar behavior and compare or contrast with other classes. It is reasonable to get a coarser level of granularity

   (d) Getting the precise number of steps of algorithm is meaningless. Since the steps countered is base on higher level programming language. The actual number of steps depends on the computing architecture. There would be numbers of low-level machine instructions to perform one high-level operation.

   (e) Tools

      i. Asymptotic Upper Bounds

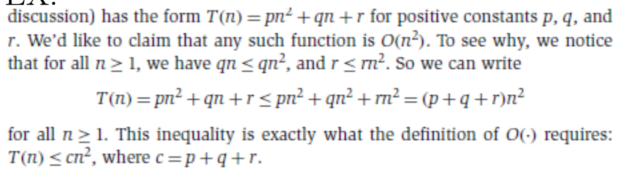
         A. Definition:

$$T(n)\ is\ O(f(n)),\ if\ \exists\, c > 0,\ n_0 > 0\ so\ that\ \forall n > n_0,\ T(n) \le c \cdot f(n)$$

         in this case, T is asymptotically upper-bounded by f.

         B. NOTES:
         exist c work for $\forall n$, c is not depends on n

         C. EX:



discussion) has the form $T(n) = pn^2 + qn + r$ for positive constants p, q, and r. We'd like to claim that any such function is $O(n^2)$. To see why, we notice that for all $n \ge 1$, we have $qn \le qn^2$, and $r \le rn^2$. So we can write

$$T(n) = pn^2 + qn + r \le pn^2 + qn^2 + rn^2 = (p + q + r)n^2$$

for all $n \ge 1$. This inequality is exactly what the definition of $O(\cdot)$ requires: $T(n) \le cn^2$, where $c = p + q + r$.

         $f(n) = n^2$ in this case

         D. There are more than one correct upper bound. We always choose the tightest upper bound

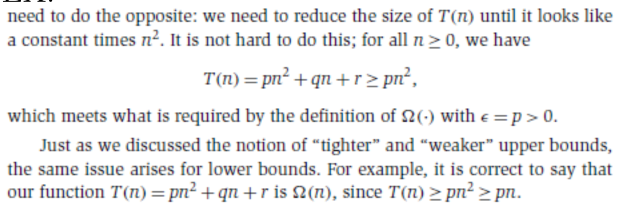
      ii. Asymptotic Lower Bound

         A. Why need? When we analyze an algorithm, we want to show that this upper bound is the best one possible. The lower bound is applied.

         B. Definition:

$$T(n) is \omega(f(n)), if \exists \epsilon > 0, \exists n_0 > 0 so\ that \forall n \ge n_0,\ we\ have T(n) \ge \epsilon \cdot f(n)$$

         C. EX:



need to do the opposite: we need to reduce the size of $T(n)$ until it looks like a constant times n^2. It is not hard to do this; for all $n \ge 0$, we have

$$T(n) = pn^2 + qn + r \ge pn^2,$$

which meets what is required by the definition of $\Omega(\cdot)$ with $\epsilon = p > 0$.

Just as we discussed the notion of "tighter" and "weaker" upper bounds, the same issue arises for lower bounds. For example, it is correct to say that our function $T(n) = pn^2 + qn + r$ is $\Omega(n)$, since $T(n) \ge pn^2 \ge pn$.

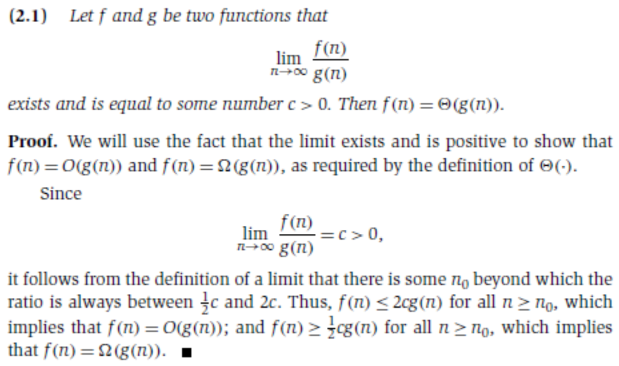
iii. Asymptotically Tight Bounds: The $T(n)$ is both $O(f(n))$ and $\omega(f(n))$

A. Definition:

$$T(n)\ is \Theta(f(n)),\ if\ T(n)\ is\ both\ O(f(n))\ and\ \omega(f(n))$$

B. This the asymptotically tight bound for worst case running time

C. Obtaining an asymptotically tight bound by computing a limit as n goes to infinity



(2.1) *Let f and g be two functions that*

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}$$

exists and is equal to some number $c > 0$. Then $f(n) = \Theta(g(n))$.

Proof. We will use the fact that the limit exists and is positive to show that $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, as required by the definition of $\Theta(\cdot)$.

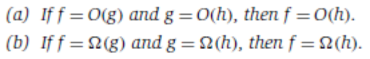
Since

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = c > 0,$$

it follows from the definition of a limit that there is some n_0 beyond which the ratio is always between $\frac{1}{2}c$ and $2c$. Thus, $f(n) \le 2cg(n)$ for all $n \ge n_0$, which implies that $f(n) = O(g(n))$; and $f(n) \ge \frac{1}{2}cg(n)$ for all $n \ge n_0$, which implies that $f(n) = \Omega(g(n))$. ■

(f) Properties of Asymptotic Growth Rates

i. Transitivity



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

(b) If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.

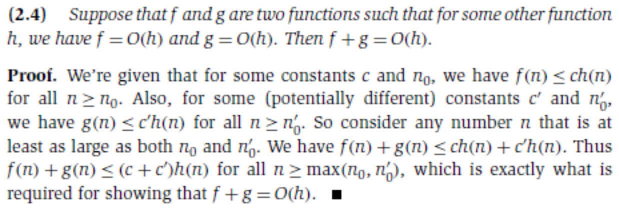
proof $\exists c\ and\ n_0$, we have $f(n) \le cg(n),\ \forall n \ge n_0$
Also, $\exists c'\ and\ n'_0$, we have $g(n) \le c'h(n),\ \forall n \ge n'_0$
Consider any number n at least as large as n_0 and n'_0, we have

$$f(n) \le cg(n) \le cc'h(n)$$

and so $f(n) \le cc'h(n)\ \forall n \ge max(n_0, n'_0)$

ii. Transitivity of Asymptotically tight bound
if $f = \Theta(g),\ and\ g = \Theta(h),\ then\ f = \Theta(h)$
proof
Since $f = O(g)$ and $g = O(h)$, by i-(a), we know, $f = O(h)$. Similarly, by i-(b), we know that $f = \Omega(h)$.

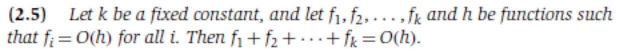
iii. Sum of Functions



(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 $f + g = O(h)$.*

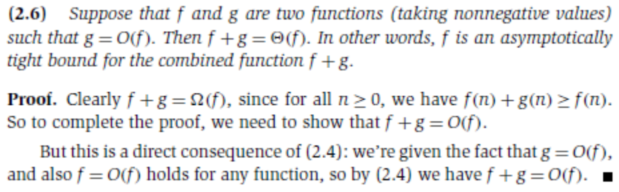
Proof. We're given that for some constants c and n_0, we have $f(n) \le ch(n)$ for all $n \ge n_0$. Also, for some (potentially different) constants c' and n'_0, we have $g(n) \le c'h(n)$ for all $n \ge n'_0$. So consider any number n that is at least as large as both n_0 and n'_0. We have $f(n) + g(n) \le ch(n) + c'h(n)$. Thus $f(n) + g(n) \le (c + c')h(n)$ for all $n \ge \max(n_0, n'_0)$, which is exactly what is required for showing that $f + g = O(h)$. ■

In generalization



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

$k \ge 2$



(2.6) *Suppose that f and g are two functions (taking nonnegative values) such that $g = O(f)$. Then $f + g = \Theta(f)$. In other words, f is an asymptotically tight bound for the combined function $f + g$.*

Proof. Clearly $f + g = \Omega(f)$, since for all $n \ge 0$, we have $f(n) + g(n) \ge f(n)$. So to complete the proof, we need to show that $f + g = O(f)$.

But this is a direct consequence of (2.4): we're given the fact that $g = O(f)$, and also $f = O(f)$ holds for any function, so by (2.4) we have $f + g = O(f)$. ■

If $g = \Omega(f)\ and\ h = \Omega(f)\ then\ g + h = O(f)$
Similar to Θ

iv. Property 1: If $g = O(f), then f = \Omega(g)$
Proof $\Omega(g) = \Omega(O(f)) = f$ or
$g = cf, f = \frac{1}{c}g$

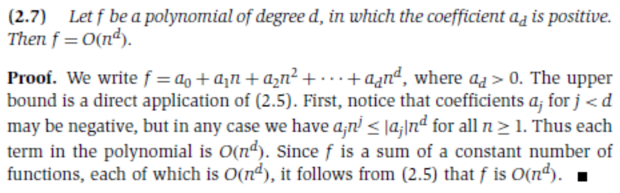
v. Corollary: if $g = O(f) and\ f = O(g), then g = \Theta(f)$
ProofBy property 1: If $g = O(f) then f = \Omega(g)$. Thus, $f = \Theta(g)$

vi. Property 2: If $g = \Theta(f) then f = \Theta(g)$
proof If $g = \Theta(f), then g = O(f) and g = \Omega(f); f = O(g) and f = \Omega(g). Thus f = \Theta(g)$

vii. Property 3: If $g = O(f), \forall k > 0, g = O(kf)\ and\ kg = O(f)$
Proof$\forall k > 0,\ By\ definiton\ of\ Big\ O(\cdot),\ \forall n > n_0, g \le cf(n).$
$Let\ d = kc,\ kg \le df(n).\ Thus\ kg = O(f)$
$Similarly,\ let\ d = \frac{c}{k},\ g \le cf = d(kf).\ Thus\ g = O(kf)$

(g) Bound examples of common functions

i. Polynomial



(2.7) *Let f be a polynomial of degree d, in which the coefficient a_d is positive. Then $f = O(n^d)$.*

Proof. We write $f = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d$, where $a_d > 0$. The upper bound is a direct application of (2.5). First, notice that coefficients a_j for $j < d$ may be negative, but in any case we have $a_j n^j \le |a_j| n^d$ for all $n \ge 1$. Thus each term in the polynomial is $O(n^d)$. Since f is a sum of a constant number of functions, each of which is $O(n^d)$, it follows from (2.5) that f is $O(n^d)$. ■

6. Graphs

(a) Definition:

i. Graph: consists of a collection V of nodes and a collection E of edges, each of which ”joins” two of the nodes

A. Directed Graph: consists of a set of nodes V and a set of directed edges E.–Each $e \in E$ is an ordered pair (u,v);the role of u and v are not interchangeable, and u is the tail of the edge and v the head.

B. Undirected Graph

ii. Path (in undirected graph): the sequence of vertex with the property that each consecutive pair $v_i,\ v_{i+1}$ is joined by an edge in the graph

A. Simple Path: all of the path’s vertices are distinct from one another

B. Cycle: a path $v_1, ..., v_k$ in which $k > 2$, the first $k-1$ nodes are all distinct, and $v_1 = v_k$.

C. Distance(short path): the minimum number f edges in a u-v path.

iii. Connectivity:

A. Connected: an undirected connected graph: for every pair of nodes u and v, there is a path from u to v

B. Directed Strongly Connected: for every two nodes u and v, there is a path from u to v and a path from v to u

iv. Tree(undirected acyclic graph): an connected undirected graph and does not contain a cycle.

A. Parent of v: to be the node u that directly precedes v on its path from r.

B. w is a Descendant of v: if v lies on the path from the root to w.

v. Directed Acyclic Graph(DAG): a directed graph has no cycle.

vi. Topological Ordering of G: an ordering of its nodes as $v_1, ..., v_n$ so that for every edge(v_i, v_j), we have $i < j$
Why need this order? It provides an order for tasks to performed, so that all independence are respected.

vii. Connected Component: A set R consist nodes that are reachable from the starting node s.

viii. Independent Set: $U \subseteq V$ to be independent set if $v,\ u \in U$(the node set), and $(u, v) \notin E$

ix. Degree: the number of edges that are incident to the vertex.

x. Spanning Tree: A subset $T \subseteq E$ of graph G if (V, T) is a tree.

(b) Property:

i. **Every n-node tree has exactly n-1 edges**

proof

We prove the result by using induction on n, the number of vertices. The result is obviously true for n = 1, 2 and 3. Let the result be true for all trees with fewer than n vertices. Let T be a tree with n vertices and let e be an edge with end vertices u and v. So the only path between u and v is e. Therefore deletion of e from T disconnects T. Now,$T \backslash \{e\}$ consists of exactly two components T1 and T2 say, and as there were no cycles to begin with, each component is a tree. Let n1 and n2 be the number of vertices in T1 and T2 respectively, so that $n_1 + n_2 = n$. Also, $n_1 < n$ and $n_2 < n$. Thus, by induction hypothesis, number of edges in $T1$ and $T2$ are respectively $n_1 - 1$ and $n_2 - 1$.Hence the number of edges in $T = n_1 - 1 + n_2 - 1 + 1 = n_1 + n_2 - 1 = n - 1$.

ii. **Let G be an undirected graph on n nodes. Any two of the following statements implies the third.**

A. G is connected.

B. G does not contain a cycle.

C. G has n-1 edges

proof

iii. **For any two nodes s and t in a graph, their connected components are either identical or disjoint.**[Think about forests. A group of trees.]

proof

Consider any two nodes s and t in a graph G with the property that there is a path between s and t.

We claim that the connected components containing s and t,are R_s and R_t, are the same set.

Consider a node $v \in R_s$, the statement $v \in R_t$ is also true.Since s and v are connected, s and t are connected, we can get v and t are connected.Thus a node is component of one if and only if it is in the component of the other.

On the other hand, if there is no path between t and s, there is no such node v in each component of each.For if there were such a node v, then, s and v are connected, v and t are connected, we get find a path that connects s and t.

Thus if there is no path between s and t, then their conncted components are disjoint.

iv. **G is DAG if and only if it has topological ordering**

A. **If F has a topological ordering, then G is a DAG**[The topological ordering of G show that G is acyclic.]

proof

Suppose, by way of contradiction, that G has a topological ordering and also a cycle C. Let v_i be the lowest-indexed node in C, and let v_j be the node on C just before v_i—- Thus (v_j, v_i) is an edge.

But by our choice of i, we have $i < j$, which contradicts the assumption of topological ordering in G

B. **In every DAG G, there is a node v with no incoming edges**

proof

Let G be a directed graph in which every nodes has a least one incoming edge.
We show how to find a cycle in G; this will prove the claim.
Pick a node v, go backward from v. After $n + 1$ steps, we will visit some node w twice, since every node has at least one incoming edges. Let C denote the sequence of nodes encountered between successive visits to w, then C is a cycle.

C. **If G is a DAG, then G has a topological ordering.**
proof
We claim by induction that every DAG has a topological ordering. In base case, it is clearly true for one or two node graphs.
Suppose it is true for n nodes DAG. Then, given DAG with $n + 1$ nodes, we choose the node v that has no incoming edge and place v first in topological ordering. Now $G\backslash\{v\}$ is a DAG, since it will not create a cycle that wasn't there previously. By induction, the n-nodes DAG has topological ordering. We append the nodes of $G\backslash\{v\}$ in this order after v. Thus G has topological order.

(c) Representation of Graph

i. Adjacency Matrix: is an $n \times n$ matrix A where A[u, v] is equal to 1 if the graph contains the edges (u,v) and 0 otherwise.
If the graph is undirected, the matrix A is symmetric, with A[u,v]=A[v,u] for all nodes $u, v \in V$

A. Advantage: Checking the existence of a edge (u,v), takes O(1) time.

B. Disadvantage:

- The representation takes $\Theta(n^2)$ space.
- To find all edges incident to a given node. It will take $\Theta(n)$, but many graphs have significantly fewer edges than n.

ii. Adjacency List: a record for each node v, containing a list of the nodes to which v has edges.[An array Adj, where Adj[v] is a record containing a list of all nodes adjacent to node v.]
for a undirected graph, each edge occurs on two adjacent lists

iii. Binary Relation: the subset R of $X \times Y$

(d) Find connectivity of a node.Exploring a Connected Component.

```
Start with R={v};
while $\exists e = (v, u) \in E, s.t. v \in R and u \notin R$ do
  | add u to R
end
```

(e) Depth-First Search(DFS)

i. DFS algorithm:

```
$DFS(u)$ : Mark u as "Explored" and add u to R
for each edge (u, v) incident to u do
  if v is not marked "Explored" then then
    Recursively invoke DFS(v)
  end if
end for
```

ii. Property:

A. Property 1:**For a given recursive call DFS(u), all nodes that are marked "Explored" between the invocation and end of this recursive call are descendants of u in T**[All nodes produce by calling DFS() on node u are descendants of u]

B. Property 2:**Let T be a depth-first search tree, let x and y be nodes in T, and let (x,y) be an edge of G that is not an edge of T. Then one of x or y is an ancestor of the other.**
proof Suppose that (x,y) is an edge of G that is not an edge of T, and suppose that x is reached first by the DFS algorithm.
When the edge(x,y) is examined during the execution of DFS(x), it is not added only because y is marked "Explored".
Since there is time that y was not marked "Explored" when DFS(x) first invoked, it is a node that was discovered between the invocation and end of recursive call DFS(x).
Thus, y is a descendant of x, by property 1.

(f) Breadth-First Search: explore outward from the root in all possible directions, adding nodes on "layer" at a time.

i. BFS algorithm:
Define the layers $L_1, L_2, ...$
Layer L_1 consists of all nodes that are neighbors of s. We define the L_0 to be the layer only consisting the root s
Assuming that we have defined layers $L_1, ..., L_i$, then layer L_{i+1} consists all nodes that don't belong to earlier layers and have an edges to a node in layer L_i

(g) Compare and Contrast BFS and DFS:

i. Similarity: they both build the connected component containing s, and they achieve similar levels of efficiency.

ii. Difference

A. DFS: the root-to-leaf paths are narrow and deep

B. BFS: the root-to-leaf paths are as short as possible

7. Greedy Algorithm

(a) Algorithm to Solve Interval Scheduling Problem.

```
Initially let R be the set of all request, and let A be empty
while R is not yet empty do
    choose a request $i \in R$ that has the smallest finishing time
    Add request i to A
    Delete all request from R that are not compatible with request i
end while
Return the set A as the set of accepted requests
```

i. A is a compatible set of request.
proof prove by induction, the base case is compatible, n-request case is hold, $n+1$ is hold because the algorithm will choose a request that compatible with the n-request case.

ii. For all indices $r \leq k$, we have $f(i_r) \leq f(j_r)$.[For each $r \geq 1$, the r^{th} accepted request in the algorithm's schedule finishes no later than the r^{th} request in the optimal schedule set O]
proof
Prove by induction

iii. **The greedy algorithm returns an optimal set A**
proof

prove by contradiction. Show that if the opposite is true then the algorithm will not terminate.

iv. The algorithm run in time $O(nlogn)$
Sorting the n request in ordering of finishing time==takes $O(nlogn)$
For each consecutive interval, add the request if compatible, else skip go to next.==take $O(n)$ time
$O(nlogn + n) = O(nlogn)$

8. Mergesort

9. Minimum Spanning Tree problem

   (a) Kruskal's Algorithm

      i. Principle:Start with empty, building a spanning tree by successively inserting edges from E in order of increasing cost. We insert each edges as long as it does not create a cycle when added to the edges we've already inserted. If create a cycle, discard it and continue.

   (b) Prim's Algorithm

      i. Principle: Initially $S = \{s\}$. In each iteration, add one node v that minimizes the attachment cost $min_{e=(u,v):u \in S} c_e$, and including the edge $e = (u, v)$

   (c) Reverse-delete algorithm

      i. Principle:Start with full graph (V,E), and begin deleting edges in order o decreasing cost. As we get to each edge e, we delete it as long as the graph is still connected

10. Divide and Conquer

   (a) Finding the Closest Pair of Points
   Given n points in the plane, find the pair that is closest together

      i. Algorithm:

```
Closest-Pair(p):
Construct $P_x$ and $P_y$ [takes $O(nlogn)$ time]
$(p_0^*, p_1^*)$ =Closest-Pair-Rec($P_x, P_y$)

Closest-Pair-Rec($P_x, P_y$)
if $|P| \leq 3$ then
   find closet pair by measuring all pairwise distances
end if

Construct $Q_x$, $Q_y$, $R_x$, $R_y$ [takes O(n) time]
$(q_0^*, q_1^*)$=Closest-Pair-Rec($Q_x, Q_y$)
$(r_0^*, r_1^*)$=Closest-Pair-Rec($R_x, R_y$)

$\delta = min(d(q_0^*, q_1^*), d(r_0^*, r_1^*))$
$x^* = maximum\ x - coordinate\ of\ a\ point\ in\ set\ Q$
$L = \{(x, y) : x = x^*\}$
$S = points\ in\ P\ whithin\ distance\ \delta\ of\ L$

Construct $S_y$ [takes O(n) time]
for each point $s \in S_y$ do
   compute distance from s to each of next 15 points in $S_y$
   Let $s$ , $s'$ be pair achieving minimum of these distance [takes O(n) time]
```

```
end for

if $d(s, s') < \delta$ then
   Return $(s, s')$
else if $d(q_0^*, q_1^*) < d(r_0^*, r_1^*)$ then
   Return $(q_0^*, q_1^*)$
else
   Return $(r_0^*, r_1^*)$
end if
```

11. Dynamic Programming

   (a) Principle:
   The dynamic programming solution based on a recurrence equation that expresses the optimal solution in therms of the optimal solution to smaller sub-problems

   (b) Weighted Interval Scheduling

      i. Goal: Selecting a subset $S \subseteq \{1, ..., n\}$ of mutually compatible intervals, so as to maximize the sum of the value of the selected intervals, $\sum_{i \in S} v_i$.

      ii. Assumptions:
         - A requests list $R = \{1, ..., n\}$
         - For each request i, $i = (s_i, f_i, v_i)$, s_i: starting time, f_i: finishing time, v_i: weight.
         - The requests are stored in order of non-decreasing finishing time
         - A request i comes before j if $i < j$, $\Leftrightarrow f_i < f_j$
         - Define $p(j) = max\{i \mid i < j,\ i \cup j = \emptyset\}$ the largest index i such that $i < j$ and intervals i and j are disjoint
         - Define $p(j) = 0$ if no request $i < j$ is disjoint from j
         - Define O_n to be optimal solution. Then a request n can be $n \in O_n$ or $n \notin O_n$
            - If $n \in O_n$, then all requests $\{p(n) + 1, ..., n - 1\} \not\subset O_n$, and $O_n \subseteq \{1, ...p(n)\}$, where $\{1, ...p(n)\}$ is the set of requests that are compatible with request n and earlier that it.
            - If $n \notin O_n$, then $O_n \subseteq \{1, ..., n - 1\}$
         - Define O_j to be the optimal solution of requests $\{1, ..., j\}$. Using this for reducing n requests problem to a j-request problem
         - $OPT(j)$ is the value of the optimal solution. And $OPT(0) = 0$. $OPT(n)$ is the value of the optimal solution O_n
            - If $j \in O_j$, then $OPT(j) = v_j + OPT(p(j))$ the weight of request j and all the earlier requests that are compatible with j
            - If $j \notin O_j$, then $OPT(j) = OPT(j - 1)$ which is $OPT(j - 1) = v_{j-1} + OPT(p(j - 1))$
            - We can get that $j \in O_j$ if and only if $v_j + OPT(p(j)) > OPT(j - 1)$
            - Since there is only tow possibilities of j, $OPT(j) = max\{v_j + OPT(p(j)), OPT(j-1)\}$

      iii. Algorithm: Compute-Opt

```
Compute-Opt(j)
if $j = 0$ then
   Return 0
else
   Return $max(v_j + Compute - Opt(j),\ Compute - Opt(j - 1))$
end if
```

iv. Correctness:
Compute-Opt(j) correctly computes OPT(j) for each j=1,...,n
Proof
By induction: $OPY(0) = 0$. Consider $j > 0$, and assuming that Compute-Opt(j) correctly compute OPT(j) for all $i < j$. We can know that $Compute-Opt(j) = OPT(p(j))$ [the algorithm return $max(v_j + Compute-Opt(j),\ Compute-Opt(j-1))$ at this point.] and $Compute-Opt(j-1) = OPT(p(j-1))$.
Thus $OPT(j) = max(v_j + Compute - Opt(j),\ Compute - Opt(j-1))$
$= Comput - Opt(j)$

v. Complexity:
exponential time in the worst case How to show it?
The recursive algorithm solves $n+1$ subproblems $Compute-Opt(0), ..., Compute-Opt(n)$.
The redundancy of computing these subproblems cause the complexity.

vi. Improvement: by memoizing the recursion
Store the value of Compute-Opt in a globally accessible place the first time we compute it and using it when need it.
Introducing array M[0,...,n]. M[0] is the value of empty.

vii. Algorithm the only give the total weight of the optimal set

```
M-Compute-Opt(j)
if $j = 0$ then
  Return 0
else if $M[j]$ is not empty then
  Return M[j]
else
  Define $M[j] = max(v_j + M - Compute - Opt(p(j)), M - Compute - Opt(j-1))$
  Return $M[j]$
end if
```

viii. Complexity:
The running time of M-Compute-Opt(n) is O(n) with assumption that the input intercals are soted by their finish times what happens if we remove this assumption?
Proof
The time spend in a single call to M-Compute-Opt is O(1).
Consider the progress of execution. Initially the number of M's entry is 0; but each time the procedure invokes the recurrence, issuing two recursive calls to M-Compute-Opt, it will fills a new entry. Since the size of M array is $n + 1$, thus there is at most O(n) calls

ix. A algorithm return optimal set

- Option1: explicit optimal set S
  Maintain an additional array S so that S[i] contains an optimal set of intervals. It would blow up the running time by an additional factor of O(n).
- Option2: implicit optimal set S
  recovering the optimal solution from values saved in array M
  By the theorem that **We can get that** $j \in O_j$ **if and only if** $v_j + OPT(p(j)) \leq OPT(j-1)$
  The algorithm Find-Solution(j)

```
Find-Solution(j)
if $j = 0$ then
  Output nothing
else
```

```
        **if** $v_j + M[p(j)] \leq M[j-1]$ **then**
            Output j together with the result of Find-Solution(p(j))
        **else**
            Output the result of Find-Solution(j-1)
        **end if**
    **end if**
```

- Complexity:
  O(n)

x. An algorithm uses iterative

```
Iterative-Compute-Opt(j)
$M[0] = 0$
**for** $j = 1, ..., n$ **do**
    $M[j] = max(v_j + M[p(j)], M[j-1])$
**end for**
```

- Correctness
  By definition $M[0] = 0$, For some $j > 0$, suppose that Iterative-Compute-Opt(j) correctly computes $M[i]$ for all $i < j$.
  By the induction hypothesis, we know that $Iterative - Compute - Opt(j) = M[j]$ and $Iterative - Compute - Opt(j-1) = M[j-1]$. Thus
  $M[j] = max(v_j + M[p(j)], M[j-1])$
  $= Iterative - Compute - Opt(j)$
- Complexity: O(n)