Module 1

# Proof

* A mathematical proof can be broken down into a number of sentences.
* For each sentence we must be able to determine whether it is true or not
  1. Example: we can determine the truth value of “1 + 1 = 2”, but we cannot determine the truth value of “she feels happy”
* Such sentences are called **statements.**
* The first way is to state an axiom or contradiction, and it is trivial to tell their truth values.
* The second way is to through specifying or quantifying parameters for open statement forms.
  1. Example: “she is a student of KU” is an open statement form called **predicate**
  2. It is open because we cannot know its truth value unless “who is she” is specified.
  3. Specifying parameters of a predicate
     + Once the **parameters** or **arguments** of a predicate are specified, a predicate becomes a statement.
     + Example: we will be able to determine the truth value of “Mary is a student of KU”
  4. A predicate can take multiple parameters.
     + Example: a predicate “x is a student of y” takes two parameters x and y and can be written as P(x, y)
  5. Quantifying parameters of a predicate:
     + Using either **universal quantifier** or the **existential quantifier**
     + For given predicate, multiple quantifiers on multiple parameters may apply.
       - can mean “there exists a woman who attends a university”.
     + in many cases, we may also specify the domain for parameters when using quantifiers,
     + we can **disprove a universal statement by counterexample.**
* When proving, we are trying to argue what we are trying to prove is correct, so we are constructing an **argument.** 
  + An argument is a collection of **ordered** statements.
  + The way we order our statement is called argument form
  + Example: module 1 slide 34
* **Premise** and **conclusion**
  + The **last statement** of an argument is called the conclusion.
  + **All** the other statements of an argument are called the premises.
* An argument form is valid if and only if for any combination of the truth value of the statements, if all the premises are true, then the conclusion is true.
* Proving the validity of an argument can be done using truth table
* Example:
  + Consider the argument form of
  + We identify the premises as
    - q
  + We identify the conclusion as
    - P

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| p | q |  | q | p |
| T | T | T | T | T |
| T | F | F | F | T |
| F | T | T | T | F |
| F | F | T | F | f |

* + The truth table is
  + We can see there is a case where all premises are true, but the conclusion is false, there the argument is invalid.
* A **sound** argument has two components.
  + All premises are true.
  + The argument form is valid
* An argument is unsound
  + **if it only has a valid argument form and not all of its premises are true.**
    - Example: module 1 slide 44
  + **if all of its premises are true but has an invalid argument form**
    - Example: module 1 slide 45
* Techniques
  + Direct proof
    - To construct an argument to show the conclusion is true
    - Constructive proof
      * Proof the existence / disprove the universal by example / counterexample.
      * Proof by exhaustion (countable and limited domain)
      * Prove by generalizing a generic particular
    - Non-constructive proof
  + Indirect proof
    - Tries to first prove some facts that are related to the conclusion, then indirectly prove the conclusion based on the proven facts
    - Use it because some cases, direct proof is difficult to construct
    - Prove by contradiction.
      * It attempts to prove the conclusion by showing that if the conclusion is false, some contradiction will happen
    - Prove by contraposition.
      * Instead of directly proving a conditional statement is true, we will indirectly prove its contraposition is true
  + Mathematical induction
    - Prove statements made on sequence of entities
    - Example1: prove for every positive n, is divisible by 5
      * Module 1 slide 66
    - Example2: prove that for all integers k >= 8, k cents can be obtained using 3-cent and 5cent coins
  + Common errors
    - Inverse error
      * The inference “p -> q, !p, then !q” is wrong
      * Can be proved by truth table
    - Converse error
      * The inference “p -> q, q, then p” is wrong
      * Arguing from examples
        + Only showing examples that fail to cover the entire domain is wrong
      * Ambiguous premises
      * Jumping into conclusion
      * Circular reasoning

Proof for GS algorithm for stable matching

* Once a woman becomes matched, she will never be free again: obviously from the algorithm, as once a woman becomes matched, no condition can make her free again.
* A matched woman only "trades up", meaning she will never get a less preferred partner than her current one: obviously from the algorithm, a woman rejects a proposal from a man if the current man she's engaged to is more preferred than him.

**(lemma1) Produce perfect matching “all men and women get matched upon the termination of the algorithm (perfection)”**

**Proof: (by contradiction)**

Let (m) be any unmatched man. Since each man can only pair with one woman, we also have a free woman (w). Because (w) is unmatched, (w) has yet to be proposed to.

However, the algorithm must terminate (based on the given premise), meaning all men will have proposed to all women. The while loop will break if there are no more free men or all men have proposed to all women. We know (m) is a free man, meaning all men must have proposed to all women.

This would mean our free man (m) would have had to propose to all women, contradicting that (w) has yet to be proposed to.

So, there exists no man nor woman being unmatched upon termination of the algorithm.

**(lemma) Produce stable matching “there exists no unstable matching upon the termination of the algorithm (stability)”**

**Proof: (by contradiction)**

Suppose there is an unstable pair ((m), (w')).

Upon program termination, (m) is paired with (w) and (m') is paired with (w'). Thus, we currently have the pairs ((m),(w)) and ((m'),(w')). The unstable pair ((m), (w')) means that (m) likes (w') better than their current partner (w), and (w') likes (m) better than their current partner (m').

In one case, (m) has not yet proposed to (w'). Men propose in decreasing order of preference, so (m) must prefer (w) over (w'). This contradicts that (m) prefers (w') over (w).

In another case, (m) has proposed to (w'). **2**Meaning, (w') prefers (m') over (m) because women only trade up**2**, regardless of when (m) is rejected. This case also contradicts that (w') prefers (m) over (m').

So, by contradiction, there exists no unstable matching upon the termination of the algorithm.

# Compound Statement

* multiple statements can be combined to form a new statement.
* 4 elementary logical operators to combine 2 statements.
  + Conjunction (AND)
  + Disjunction (OR)
  + Negation (NOT)
  + Condition (IF)
* A conjunction (AND) statement is true if both of its elementary statements are true.
* A disjunction (OR) statement is true if either of its elementary statements is true.
* A negation (NOT) statement is true if its elementary statement is false.
* A conditional statement (IF) consists of one statement as the condition, and another as the conclusion. A conditional statement is false only if the condition is true and the conclusion is false.
  + Note that if the condition is false, then no matter whether the conclusion is true or not, the conditional statement is true.
  + Example: if 1 + 1 = 0, a professor will be rich
    - Since you will never be able to know whether a professor is rich in a world where 1 + 1 = 0, you cannot claim the statement is false, hence it is true

# Truth Table

* Truth table summarize how we determine the truth values of the component statements, given different combinations of truth values for their elementary statements.
* The precedence of four elementary logical operators (high -> low)
  + NOT
  + AND
  + OR
  + IF
* Two statements are logically equivalent if for any given combination of values of the elementary statements, the truth values for both statements are identical. Denote logical equivalence with ”
* A set of laws can be used to establish logical equivalence between compound statements.
* De Morgan’s Law
  + and
  + Prove that

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| p | q | ¬p | ¬q | p v q | ¬(p v q) | ¬p ⋀ ¬q |
| T | T | F | F | T | F | F |
| T | F | F | T | T | F | F |
| F | T | T | F | T | F | F |
| F | F | T | T | F | T | T |

* + Prove that

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| p | q | ¬p | ¬q | p v q | ¬(p v q) | ¬p ⋀ ¬q |
| T | T | F | F | T | F | F |
| T | F | F | T | F | T | T |
| F | T | T | F | F | T | T |
| F | F | T | T | F | T | T |

* Commutative laws
  + and

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| p | q |  |  |  |  |
| T | T | T | T | T | T |
| T | F | T | T | F | F |
| F | T | T | T | F | F |
| F | F | F | F | F | F |

* Associative law
  + and
  + Prove

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| p | q | r |  |  |  |  |
| T | T | T | T | T | T | T |
| T | T | F | T | T | T | T |
| T | F | T | T | T | T | T |
| T | F | F | T | F | T | T |
| F | T | T | T | T | T | T |
| F | T | F | T | T | T | T |
| F | F | T | F | T | T | T |
| F | F | F | F | F | F | F |

* Distributive law
  + and
  + Prove

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| p | q | r |  |  |  |  |  |
| T | T | T | T | T | T | T | T |
| T | T | F | T | T | T | F | T |
| T | F | T | T | T | F | T | T |
| T | F | F | F | F | F | F | F |
| F | T | T | T | F | F | F | F |
| F | T | F | T | F | F | F | F |
| F | F | T | T | F | F | F | F |
| F | F | F | F | F | F | F | F |

* Identity law
  + and

|  |  |  |
| --- | --- | --- |
| p | t | p ∧ t |
| T | T | T |
| F | T | F |

|  |  |  |
| --- | --- | --- |
| p | c | p v t |
| T | F | T |
| F | F | F |

* Negation law
  + and

|  |  |  |  |
| --- | --- | --- | --- |
| p | ¬𝑝 | t | 𝑝 ∨ ¬𝑝 |
| T | F | T | T |
| F | T | T | T |

|  |  |  |  |
| --- | --- | --- | --- |
| p | ¬𝑝 | c | p ∧ ¬𝑝 |
| T | F | F | F |
| F | T | F | F |

* Contraposition law

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| p | q |  |  |  |  |
| T | T | T | F | F | T |
| T | F | F | F | T | F |
| F | T | T | T | F | T |
| F | F | T | T | T | T |

* Double negative law

|  |  |  |
| --- | --- | --- |
| p | ¬𝑝 | ¬(¬𝑝) |
| T | F | T |
| F | T | F |

* Idempotent law
  + and

|  |  |  |
| --- | --- | --- |
| p | p | p ^ p |
| T | T | T |
| F | F | F |

|  |  |  |
| --- | --- | --- |
| p | p | p v p |
| T | T | T |
| F | F | F |

* Universal bound law
  + and

|  |  |  |
| --- | --- | --- |
| p | t | p ∨ t |
| T | T | T |
| F | T | T |

|  |  |  |
| --- | --- | --- |
| p | c | p ∧ c |
| T | F | F |
| F | F | F |

* Absorption law
  + and

|  |  |  |  |
| --- | --- | --- | --- |
| p | q | 𝑝 ∧ 𝑞 | 𝑝 ∨ (𝑝 ∧ 𝑞) |
| T | T | T | T |
| T | F | F | T |
| F | T | F | F |
| F | F | F | F |

|  |  |  |  |
| --- | --- | --- | --- |
| p | q | 𝑝 ∨ 𝑞 | 𝑝 ∨ (𝑝 ∧ 𝑞) |
| T | T | T | T |
| T | F | T | T |
| F | T | T | F |
| F | F | F | F |

* Negation of t and c
  + and

|  |  |  |
| --- | --- | --- |
| c | t | ¬𝑡 |
| F | T | F |
| F | T | F |

|  |  |  |
| --- | --- | --- |
| c | t | ¬c |
| F | T | T |
| F | T | T |

Module 2

# Notation convention

* Use notation that is consistent with the meaning.
  + U (Universe) to represent all elements in a given domain.
  + i (index) to represent an index.
* do not use same notation for different things and overcomplicate the definition.
* A set of items are often represented by uppercase letter.
  + Says we have a set of different fruits F.
* Use superscript to distinguish between the sets if we have to use the same uppercase letter to represent them.
  + We define a set of red fruits as and set of green fruits as
* Use subscript to represent a specific item in an ordered set
  + Ith fruit would be written as

# Time Complexity

* time complexity measures how fast the running time grows with the input size. Basically it makes six fundamental assumptions:
  + the measures of input size
  + the relationship between the input size and the running time as a function
  + the ignorance of the constants in the function
  + focusing on the fastest-growing term
  + focusing on asymptotic behavior
  + focusing on the worst-case scenario
* in complexity analysis, the most used measure is the upper bound
  + denoted using O(.), and called big-O, ‘.’ Is the fastest growing term in T(n)
  + using upper bound, we will have a guaranteed of running time
* lower bound
  + use , called big-Omega notation.
  + it often viewed as a termination of inventing a theoretically more efficient algorithm for the problem.
* If T(n) is in both O(.) and ,with the same function ‘.’, we can say ‘.’ Is a tight bound for T(n) and use to indicate it and called big-theta notation

Module 3

# Greedy Algorithm

* At each stage, try to get closer to the mathematical objective as much as possible.
  + A myopic approach.
  + In some cases, it does solve the problem correctly.
  + In some cases, it does not
    - However, in this case, it could give a useful approximation of the optimal solution.
* Usually intuitive and simple
* Trying to achieve local optimality at each step while completely ignoring global optimality
* Often efficient and its time complexity is easy to analyze

# Knapsack Problem (Greedy algorithm)

* imagine an explorer enters a treasure cave with a small, limited capacity knapsack, and he tries to take the most valuable treasures with him
* you are given a total capacity C, a set of items I, each is assigned with a weight and a value . The problem seeks to pack a subset of items , such that and that the total value is maximized
* according to greedy design principle, we could try keep adding the most valuable item to the knapsack until it is full.

# Shortest Path problem

* Given a graph G= {V, E}, and the cost/length for each edge (denote the length of edge e as and for all ), find the shortest path from vertex u or v whose total length is minimized among all paths from u to v.

**Dijkstra’s Algorithm:**

Initialize S = {s} and d(s) = 0;

While S V:

Select a node v S with at least one edge from S for which

;

Add v to S, record d(v);

EndWhile

Prove of **Dijkstra’s Algorithm:**

**Proposition**: Given a graph G(V, E, L) with no negative edge, for any node v selected by the criteria , the quantity d(u) + corresponds to an s - v path, which is the shortest s - v path

**Proof: We will use mathematical induction for the proof.**

(Base case): The statement is clearly true when v = s, if we set d(s) = 0 and = 0 for arbitrary node i. That means the shortest path from s to s is null and its length is 0, and it is true per definition.

(induction step): Let S be a set of vertices that have been visited. We assume for any node u S we have d(u) being the shortest path length between s and u, and the corresponding shortest path is known

(Note: initially S = {s} and the hypothesis is true). In each algorithm run, we would pick the node v, where .

图示

描述已自动生成We compare the greedy path with a length of d(u) + and another arbitrary s – v path (generalization of a generic particular). Since the path connects

s S and v V – S, must contain an edge where x S and y V – S (Note: it is possible that x = u or y = v, but the equalities do not both hold at the same time because it would directly lead to d(u) + correspond to the shortest path). In per selection criterion, we have . Also, the graph has no negative-weighted edge, hence d() 0. Adding the term to the right-hand side we will get . Or (. Since is an arbitrary s – v path, it indicates that is shorter than or has the same length as any valid s – v path. Note that can be constructed by appending the edge to the path corresponding to d(u), which exists per induction hypothesis.

(conclusion): Therefore, for the node v selected by criteria , the shortest path s – v has a length of .

# The minimum spanning tree problem

* Given a graph G = (V, E, W) where G is connected and >= 0 for all e E, we would like to select a subset of edges T E such that all vertices can be connected through T and that the total cost is minimized
* The math objective of the problem is to minimize the sum of the weights of the edges contained in the subset T

**Kruskal’s Algorithm (G, W):**

Sort edges weights so that

T

For each (u make a set containing singleton u;

For i = l to m:

(u, v) =

If (u and v are in different sets):

T

Merge the sets containing u and v

Endif

Endfor

Return T

Proof of **Kruskal’s Algorithm:**

**Lemma:** Assume that all edge costs are distinct (will be relaxed later). Let S be any subset of vertices that is neither empty nor equal to all of V, and let edge e = (v, w) be the minimum cost edge with one end in S and the other end in V – S. Then every minimum spanning tree contains the edge s**. (cut property)**

Proof: argue that every edge selected by the greedy algorithm is at least as good as any other edge.

**We use prove by contradiction.**

图示, 工程绘图

描述已自动生成Proof: Consider the scenario shown in the figure, where the greedy algorithm picked edge e = (v, w), where v S and w V – S. We assume that the MST of the graph is T and it does not contain e. Per definition of MST, all nodes must be connected through it, including v and w. Since e T, there must exist another path in T that connects v, w. Since v S and w V – S, the path must contain an edge , where and . Note that . Let (replace with e in the MST). Obviously, per greedy selection criterion (also because we assumed that all edges have distinct weights). It is then clear that w() < w(T). We also argue that all nodes in G remained connected, since all connections in T that need to pass can now be redirected through . Specifically, let be the path that connects v, w, it follows that v, v’ are connected and w, w’ are connected. So, it is feasible to connect v’, w’ in T via in T’. We also claim that T’ contains no cycle. Note that T is an MST, and it contains no cycle per definition. It indicates that P is the unique path that connects v, w. After removing , no path can connect v, w anymore. After adding e, we can only create a unique connection between v, w. Since e is the only added edge in T’, the other parts of T’ remained the same as T and contain no cycle either. Taken together, T’ contains no cycle.

(conclusion): To summarize, we have shown that , T’ connects all nodes in G and T’ contains no cycle. In this case T’ is a spanning tree that has a lower cost than T, which contradicts with the assumption that T is an MST. Therefore, any MST for G must contain e.

Proposition: **Kruskal’s algorithm produces an MST of G.**

Proof: According to the proven cut property, each added edge by the Kruskal’s algorithm must be contained in an MST of G. We also claim that when the algorithm terminates, we have added sufficient amount of edges to form a spanning tree. This is true because the algorithm only terminates when all nodes are connected.

Proposition: **Kruskal’s algorithm is correct on graphs containing equally-weighted edges.**

Proof: In a special case where all edges have the same weight, any spanning tree will be an MST, since Kruskal’s algorithm identifies a spanning tree, it also identifies an MST. On the other hand, we have at least two edges that have different weights. Let the minimum difference between any two differently weighted edge be d. We can make an alternative graph G’, which has the identical node and edge set as G, but with its edge-weight ties broken by adding positive, arbitrarily small perturbations e. For example, consider 4 edges with weights 1.9, 2, 2, 2, then d = 0.1. We can set e = 0.01, and break the ties of the equally-weighted edges by reassigning their weights into 2(+0e), 2.01(+1e), 2.02(+2e). Note that we have at most |V| equally weighted edges, then we can set e < d / |V|^2 to satisfy the condition. We then run the Kruskal’s algorithm on G’ and denote the result as T’. since G’ contains no equally weighted edge, T’ is an MST of G’. Also denote that the true MST of G be T. Denote the weight of T’ evaluated on the graph G be . That is, the total weight after removing all the added perturbations. There are three cases to discuss:

1. When < . This is impossible since it contradicts with the assumption that T is the MST of G
2. When = . Then T’ is also an MST of G, so Kruskal’s algorithm is correct.
3. When > . In this case, - < d. Otherwise - d, we will have Simplify the inequation we get . Also note that since we only add positive perturbations to form G’. Taken together, we have . It indicates that T will be an MST of G’, contradicting with the fact that T’ is an MST of G’. Since d is defined as the smallest difference between any two differently weighted edges, T’ and T are identical.

In all possible cases, T’, and T are identical, hence the Kruskal’s algorithm correctly identifies an MST on graphs containing equally weighed edges.

**Prim’s Algorithm (G, W):**

While S

Select a node v with at least one edge from S for which

Add v to S;

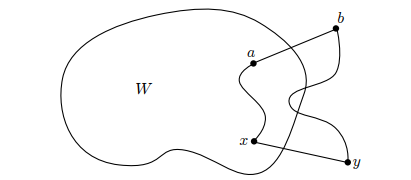
Add to T;

Endwhile

Return T

Proof that the **Prim’s algorithm is correct**

The proof is by contradiction, so assume that S is not minimum weight. Let ES = (e1, e2, · · · , en−1) be the sequence of edges chosen (in this order) by Prim’s algorithm, and let U be a minimum-weight spanning tree that contains edges from the longest possible prefix of sequence ES.

Let ei = {x, y} be the first edge added to S by Prim’s algorithm that is not in U, and let W be the set of vertices immediately before {x, y} is selected. Notice that it follows that U contains edges e1, e2, · · · , ei−1 but not edge ei . There must be a path x ❀ y in U, so let {a, b} be the first edge on this path with one endpoint (a) inside W, and the other endpoint (b) outside W, as in the following picture:

Now we define the set of edges T = U + {{x, y}} − {{a, b}}, and notice that T is a spanning tree for graph G. Consider the three possible cases for the weights of edges {x, y} and {a, b}:

1. w({a, b}) > w({x, y}): In this case, in creating T we have added an edge that has smaller weight then the one we removed, and so w(T) < w(U). However, this is impossible, since U is a minimum-weight spanning tree.
2. w({a, b}) = w({x, y}): In this case w(T) = w(U), so T is also a minimum spanning tree. Furthermore, since Prim’s algorithm hasn’t selected edge {a, b} yet, that edge cannot be one of e1, e2, · · ·, ei−1. This implies that T contains edges e1, e2, · · · , ei , which is a longer prefix of ES than U contains. This contradicts the definition of tree U.
3. w({a, b}) < w({x, y}): In this case, since the weight of edge {a, b} is smaller, Prim’s algorithm will select {a, b} at this step. This contradicts the definition of edge {x, y}.

Since all possible cases are contradictions, so the original assumption must be invalid, thus

Prim’s algorithm is correct.

Module 4

# Interval scheduling (greedy algorithm)

* Given a single-threaded CPU server, and a set of jobs J that need to be scheduled. Each job has an associated start time and finishing time . Find the maximal subset such that for any two jobs either or .
* Two jobs cannot be executed at the same time since we only have a single-threaded server, and schedule as many jobs as possible
* Intuitively, we can first sort the jobs based on their starting time, once the jobs are sorted by time, we can then schedule tasks one-by-one
* Mathematical objectives
  + At each given time point, we may need to choose from a set of mutually conflicting jobs
  + Since the final math objective is to maximize |J’|, picking any job from the mutually conflicting set will contribute equally to our objective.
  + In this case we should probably pick the task that ends as early as possible, such that we can free up the server as early as possible for more jobs.

Naïve algorithm

**Let J be the set of all jobs, and let A be empty;**

**Sort jobs in J with ascending finishing time;**

**For each is not empty:**

**Add to A;**

**Delete all request from J that conflict with**

**Endwhile**

**Return A**

Let A = {j1, j2 … } be the set of jobs produced by the greedy algorithm, and let B = {j’1, j’2…} be an arbitrary solution that is non-greedy

**Lemma**: For each I <= |B|, , that is to say, the time that needs to finish the first i jobs in A is shorter than the time that needs to finish the first i jobs in B.

Proof by induction:

(Base case): The statement is true when i=1, this is because the greedy algorithm always selects the job that finishes the earliest.

(induction hypothesis): we assume it is also true when i=k, where 1 <= k <= |B|. That is, we assume. Because B is a valid schedule, we must have . In this case, is also a valid choice to be included in A by the greedy algorithm. If the greedy algorithm picks as its next job, then . If it does not pick , it means that there exists another valid such that in either case, we have .

(conclusion): Hence, for each I <= |B|,

Proposition: the job set A returned by the greedy algorithm is optimal

Proof by contradiction:

If A is not an optimal set, then it means |A| < |B| for some conflict-free job set B. let |A| = n, it means that there exists a job . By the previous lemma, we know that . Therefore, does not conflict with and should have been included in A, which contradicts with the fact that |A| = n. Therefor, A must be an optimal set.

The overall time complexity is O(nlogn)

# Minimized Maximum Lateness

* Given a single-threaded CPU server, and a set of jobs J that need to be scheduled. Each job has an execution time and a deadline .
* Given a conflict-free schedule represented by a set of start time S = {s1, s2…}, define the lateness for as .Find the schedule S to minimize . **We want the schedule to be as fair as possible.**
* Differs from the interval scheduling problem in terms of
  + We have to schedule all tasks, rather than a subset of them
  + The scheduling of each job is more flexible, does not need to be started at a specific time
  + The math object is to minimize the lateness

**Lemma1**: A finite list of jobs can be sorted via finite number of adjacent swaps

Proof by induction:

(base case): The lemma is true when we have 2 jobs, if the two jobs are an inversion, we can convert them into a sorted list with an adjacent swap.

(induction hypothesis): We assume that we can sort k >= 2 jobs with only adjacent swaps. Given k + 1 jobs, we will first sort the first k jobs with only adjacent swaps. Then, we will keep performing adjacent swaps on the last job until we find its correct position among the first k jobs. The number of adjacent swaps we need is clearly bounded by k.

(conclusion): Hence, any job can be sorted via a finite number of adjacent swaps

**Lemma2:** There is an optimal solution with no idle time

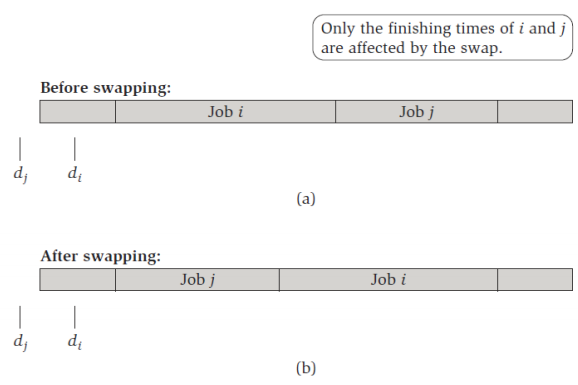
Proof: Assume that there is an optimal solution with idle time. We can remove the idle time by scheduling the rest of the jobs earlier. No job will be finished later than the original optimal schedule. Hence, the resulted schedule without idle time is also an optimal solution.

**Lemma3:** All schedules with no inversions and no idle time have the same maximized lateness.

Proof: For two schedules with no inversion and no idle time, they may only differ from each other in the execution order of jobs with the same deadline. Because no inversion is allowed, all jobs with the same deadline will be executed consecutively. The maximum lateness of these jobs thus only depends on the finish time of the last job and is independent of the ordering of these jobs. In this case, all ordering of these jobs will result in the same maximized lateness.

**Lemma4:** There exists an optimal schedule that has no inversions and no idle time

Proof by contradiction: Let’s assume that all optimal schedules have either some inversions or idle time. Per the previous lemma, we can simple remove the idle time for any schedule without increasing its associated maximum lateness. Hence, we only discuss optimal schedules that contain only inversions, let O be an optimal schedule with at least one inversion. We can clearly sort O into a schedule A with no inversion with only swapping adjacent inversions. Now we will prove that swapping adjacent inversions does no increase the maximum lateness.

Since i, j are adjacent inversion, we have per definition. With the swap, j is scheduled earlier and we will not increase the maximum lateness if it is attributed by j. The only concern is that job j is postponed and the new schedule may increase its lateness. Let the new lateness of i be . Note that in the new schedule, job I will be finished the same time as j in the old schedule. Now, we have = . It means that the lateness of job I in the new schedule remains smaller than the lateness of job j in O. Inn this case, after swapping, neither the lateness of jobs I nor j will surpass . So the swapping will not increase the maximal lateness of O. It further means that the schedule A converted from O will not have a higher maximum lateness than O, and A should also be an optimal schedule. Clearly, A is sorted and will not contain any inversion, m hence A will be an optimal schedule with no inversion and no idle time.

Proposition: The greedy algorithm finds an optimal schedule that has the minimized maximum lateness:

Proof: since there exists an optimal schedule with no inversion nor idle time, and all schedules with no inversion nor idle time have the same maximized lateness, all schedules with no inversion nor idle time have the minimized maximum lateness. The greedy algorithm produces a schedule with no inversion and no idle time, which also has the minimized maximum lateness.

Module 5

# Fibonacci number (Divide and Conquer)

* Consider the computation of Fibonacci number:
  + f(n) = f(n-1) + f(n-2) for n > 2
  + f(1) = f(2) = 1
* the computation of f(n) depends on the solution of a smaller problem in size: f(n-1) and f(n-2)
* the smaller problems are called subproblems

**Fibo(n)**

**If n < 2 return 1;**

**Return Fibo(n-1) + Fibo(n-2)**

# Divide and Conquer

* the identification of subproblem can lead to another intuitive design principle called **recursion** or **divide and conquer.**
* The idea of design principle consists three main steps
  + Partition the larger problem into a number of subproblems
  + Solve the subproblem (most likely recursively)
  + Use the results of the subproblem to solve the larger problem
* The correctness and efficiency analysis of divide an conquer algorithm could be different than those for the greedy algorithm
  + Is often much simpler
  + The efficiency analysis could be more involving
* In many case we do not need to prove the correctness of divide and conquer algorithm, if the correctness is obvious.

# Merge Sort

* A sorting algorithm developed under the divide an conquer schema
* The entire set of number is divided into two equal-sized subsets. Each subset is sorted individually, and then the two sorted subsets are merged into a larger sorted list of numbers

**MergeSort(data, n sorted)**

**If n <= 1 return InsertionSort(data, n)**

**DivideInput(data, n smallData, n/2, 2)**

**MergeSort(smallData[0], n/2, smallSorted[0])**

**MergeSort(smallData[1]. n/2, smallSorted[1])**

**Sorted = combindSorted(smallSorted[0], smallSorted[1])**

**Return**

* You can argue the correctness of the merge sort by showing
  + The division is clearly correct since the union of the two subsets is exactly the original set of number, and the intersection of the two subsets is an empty set
  + The merge of two sorted list in to a larger sorted list is also correct, since all numbers remained in the array A nd B must be larget than the number that is put into the array C
* Time complexity: O(nlogn)

Let the time required for sorting n numbers be T(n), and we have the following recursion form

We can write the recursive form into a set of formula with different input sizes:

……….

Multiply for the ith equation.

………..

We can add up all equations and simplify them, the sum becomes:

Note that each term on the right hand side evaluates to O(n) and there are terms. So the overall time complexity is .

# Master’s theorem

* We can write recursive form of algorithm as:
  + a = how many subproblems the large problem is divided into
  + b = how much smaller the subproblem is compared to the larger problem
  + c = what is the exponent of the polynomial time complexity required by the result merge process
* Once we get a, b, c, we will be able to get the closed form time complexity directly.
* only apply to divide and conquer algorithm divides the larger problem into equal-sized smaller problems.

**Proof**

For a-nary tree: now consider the ith layer in the a-nary tree

* We will have
* Each subproblem will have a size of
* Obtaining the solution of the subproblem will need time, where r is a constant term that converts the time complexity term into the actual time

Now the time required to compute the solution for all subproblems in the ith layer becomes:

Rearranging the term, we get:

Then, the overall time for the algorithm is simply the sum of the per-layer time overall all layers. Note that we have layers, which leads to an overall time of:

By analyzing the term , we realize that r, n, and c are constants for a given algorithm and input instance. The only term that is variable between different layers is .

While computing the sum of a series, it is important to know whether it is an increasing or a decreasing series.

* We need to discuss whether the term is increasing , invariable or decreasing as I increases

**Case1: an increasing series**

**Case2: an invariable series**

**Case3: an decreasing series**

Note: assumption that , so if

# Median Finding

* Find the median of a set of unsorted numbers
* Naïve solution: sort numbers and find the median
  + Bound by sorting algorithm efficiency which is O(nlogn) (fastest sorting)
* Divide and conquer algorithm for median finding by Blum, Floyd, Pratt, Rivest and Tarjan.

**Select(A, i)**

**Divide the n items into groups of 5**

**Find the median of each group of 5**

**Use Select recursively to fine the median of there n/5 medians**

**Partition the n items around x, and let k be the ranking of x**

**If I = k then return x**

**Else if i < k use Select recursively by calling Select(A[1…k-1], i)**

**Else if i > k use Select recursively by calling Select(A[k+1 … A], i-k )**

**Endif**

Proposition: The divide-and-conquer median-finding algorithm runs in O(n) time

Proof by induction:

(base case): For an input size of 1, we can complete the median finding in time c by making it sufficiently large, such as T(1) <= c \* 1 = c

(Induction): We assume running time of the algorithm T(k) <= ck for 1 <= k <= n. Now, substitute the assumption to the running time formula and we get:

Since a is a constant which associated with the running time for the partitioning process around x, we can make a large-enough constant where c > 10a and

n be negative. If n is negative, then . Which is consistent with our hypothesis at the beginning.

(conclusion): Hence the algorithm runs in O(n) time.

**Proof cannot be done any faster:**

So we will guess a lower time complexity say O(1)

Assuming the algorithm runs in O(1) and T(n) <= c for some constant c, we have:

The formula clearly indicates an O(n) time complexity, no matter how we manipulate the constant a and c, the O(n) complexity is inconsistent with what we have guessed, hence, the algorithm dos not run in O(1).

**Why divide into subproblem of size of 5**

Case1: Assume d is an odd number, then we can write the running time as:

To make a proper upper bound, we shall make negative, in other word, should be positive.

It is clear that d > 3 and since we assumed d is an odd number , the smallest value d can take is 5. When is positive, we can make c large enough such that becomes negative, and thus make the algorithm remain in O(n) time.

Case2: assume d is an even number

We must have d > 4 if d is an even number, so d = 5 works as the smallest choice that make the algorithm runs in O(n) time if combine two situations.

Module 6

# Integer Multiplication

* Multiplication of two large numbers, each with n digits
* Naïve algorithm
  + Simply multiply each digit of the second number with each digit of the first number and add up the results
  + Takes O(n^2) time
* Divide and conquer version
  + Subproblem
    - Partition each number into two subsets of digits, half digits that are more significant and half digits that are less significant
    - Example: 1200 \* 3612, we partition first number as 12 ( it should be multiplied with 100), and 00, and second number as 36 and 12.
    - Then perform 4 multiplication: 12 \* 36, 00 \* 36, 12 \* 12, 00 \* 12
    - And we know if we have two number 12 and 36 should be raise the result with \*10000, and if we have one red number, raise result with \*100
  + Generalize the problem
    - Consider multiplication of two integers x and y each with n digits
    - We can make x = and y =
    - Then xy =
  + Notice that we only need the sun of but not necessarily nor . And so - .
  + We know that we need to compute and as subproblems and we will also compute as the third subproblem.
  + We can approximate the computation of as a subproblem with size n/2, since the sum of two n/2-digit number can have at most n/2 + 1 digits. And all additional additions and subtractions can be done in O(n) time.
  + Now we revise the analysis of time complexity:
    - a=3, since we need to compute 3 multiplications as subproblems
    - b=2, the size of the subproblem is halved
    - c=1, since adding the results of the 4 multiplications clearly takes liner time
    - , which is slightly faster than the naïve algorithm.

# Counting inversions

**Merge-and-count(A,B)**

**Maintain a current pointer into each list, initialized to pointer to the front elements**

**Maintain a variable count for the number of inversion, initialized to 0**

**While both lists are noneempty:**

**Let ai, aj be the elements pointed to by the Current pointer**

**Append the smaller of these two to the output list**

**If bj is the smaller element then**

**Increment count by the number of elements remaining in A**

**Endif**

**Advance the current pointer in the list from which the**

**Smaller element was selected**

**Endwhile**

**Case**

Consider the following problem: we wish to study which courses in computer science are strongly tied to EECS 660 Algorithms.

– we consider two courses are strongly tied if students do well in EECS560 also tend to do well in the other class

– for example, students do well in EECS 660 also tend to do well in EECS 560 Data structure, but the tie could be weaker in, say, Physics classes

– of course, we can quantify the similarity in statistical sense, such as using Pearson’s or Spearman’s correlation

– but here, we would like to use another measure, called rank inversions

Say we have exactly the same set of students who have taken EECS 560 and EECS 660, and we have finished the grading with the students’ rankings in the two courses.

– for example, assume the ranking of EECS 660 is: Mary (1st), Tom (2nd), and Jacob (3rd)

– and the ranking of EECS 560 is: Tom, Jacob, and Mary.

– we can associate the ranking with the students in EECS 660, and map the rankings to EECS 560

• Mary is 1, Tom is 2, and Jacob is 3; In EECS 560, Tom (2) goes first, so we have 2 as the first mapped ranking; Jacob (3) goes second, so we have 3 as the second,… eventually we get the mapped ranking as 2, 3, 1

Then, we define an inversion in a list A as the number of cases where ai > aj if I < j. Here, ai is the ith element of A

* For example, consider the mapped ranking in EECS 560 which is 2, 3, 1
* For any combination of two elements {(2,3),(2,1),(3,1)} we know that (2,1) and (3,1) are inversion per the definition
* Thus the number of inversions we have in EECS 560’s mapped ranking is 2
* If two lists are identical the new have 0 inversions, if two lists are completely reversed, then for any pair of elements it is an inversioninversions in total
* In this case, we can use the number of inversions to quantitfy the similarity of two ranking.

Counting Inversion: Given an unsorted list of n numbers 1, 2 ……n where each number occurs exactly once, count the number of inversions in the list

To solve this problem using divide an conquer, note that we can divide the entire set of numbers into two subsets, and the total number of inversions we have in the entire list is the sum of the number of inversion in the first list, the number of inversion sin the second list, and the number of inversion between the two lists

* Using the naïve O(n^2) algorithm, we know that each of these three numbers can be computed in T(n) /4 and the sum of them is 3T(n) /4, so we should be getting some efficiency gain here
* However, per the master’s theorem, time complexity for the merge phase is in fact a lower found for the entire algorithm, we have that complexity being O(n^2), hence the naïve divide an conquer algorithm has no gain in theoretical efficiency

Hence, to improve the time complexity, we shall do better in merging the results of two subproblems.

Let A1 = {2, 6, 1} and A2 = {3, 5, 4} we can sort them individually into A1’ = {1,2,6} and A2’={3,4,5}

Per definition, we know that all numbers in A2’ should be larger than those in A1’, if there is no inversion.

So, for any number in A2, the number of inversions associated with it is exactly the number of numbers in A1, that are larger that it.

Consider A1’ = {1,2,6} and A2’={3,4,5}

* 3 is smaller than 6, 1 inversion
* 4 is smaller than 6, 1 inversion
* 5 is smaller than 6, 1 inversion
* 3 between-subset inversions in toal

We can implement the idea in a way similar to merge sort

* We set up three pointer (input A1’, A2’ output A’)
* When we need to move any number in A2’ into A’, indicates that the number is smaller than all of remaining numbers in A1’. Hence it will cause that many inversion
* This process takes O(1) time

The overall running time become:

And per the master’s theorem, time complexity is .

# Closest pair of points

**Closest-Pair(P)**

**Construct px and py**

**(P0\*, P1\*) = Closest-Pair-rec(px, py)**

**Closest-Pair-rec(px, py)**

**If |P| <= 3 then find closest pair by measuring all pairwise distances**

**Endif**

**Construct Qx, Qy, Rx, Ry**

**(q0\*, q1\*) = Closest-Pair-rec(Qx, Qy)**

**(r0\*, r1\*) = Closest-Pair-rec(Rx, Ry)**

**= min(d((q0\*, q1\*), (r0\*, r1\*)))**

**X\* = maximum x-coordinate of a point in set Q**

**L = {(x, y) : x = x\*}**

**S = points in P within distance of L**

**Construct Sy**

**For each point s Sy, compute distance from s to each of next 15 points in Sy, let s, s’ be pair achieving minimum of these distances**

**If d(s, s’) < then**

**Return(s, s’)**

**Else if d(q0\*, q1\*) < d(r0\*, r1\*) then**

**Return (q0\*, q1\*)**

**Else**

**Return (r0\*, r1\*)**

**Endif**

* For the 1-dimensional case, we can solve this problem in O(n logn) time by sorting their points according to their 1-dimensional coordinates and scan the sorted points to find the closest pair
* For the 2D case, each point is associated with a coordinate (x,y),We must divide the points. Each point on our 2D plane can be split into two subspaces so each subspace contains half of the points. To simplify this process, we cut based on the x coordinate. More specifically, we cut the plane with a vertical line x = x\*, where half of the points have their x coordinates less than x\*. We denote the line x = x\* as L and the two subsets of points as Q and R.
* Let the closest pair of points within Q or R as 𝛿. This will be found by recursively solving the problem on each subset
* Let 𝛿 be the upper bound for the between-subset distance. If the distance is larger, we return 𝛿 instead.

**Proposition**: If there exists a q in Q and r in R for which d(q, r) < 𝛿, then each of q and r lies within a distance 𝛿 of L, where L is the line that separates Q and R

Proof: Denote the x coordinate of L as x\*. If q and r exist, denote their coordinates as q = (x­q, y­­q) and r = (xr, yr). By definition of the division of subproblems, we have xq <= x\* <= xr.Thus, we have the following:

x\*-xq <= xr-xq <= d(q, r) = 𝛿

xr-x\* <= xr-xq <= d(q, r) = 𝛿

So, each of q and r has an x coordinate within 𝛿 of x\* and hence lies within distance 𝛿 of L.

* Let all points within the band be S
* Let s be a point within S. All points that could lead to a distance smaller than 𝛿 with s must satisfy the following two conditions:
  + All points must not be within the same subset as s and must be within the circle.
* Within the circle, we can only hold a limited number of points within the same subset.
  + Thus, we claim that the max number of points that can exist within the circle is upper bounded by a constant.
* Assume that we have sorted all points in S by y coordinates. Let Sy be this sorted list.

**Proposition**: If s and s’ are in S and have the property of d(s, s’) < 𝛿, then s and s’ are within 15 position of each other in the sorted list Sy

Assume we have a band and split it into small boxes into a 𝛿/2 \* 𝛿/2 grid. This way, each box is of size (𝛿^2)/4. It is not possible that two points can be within the same box, since the farthest distance between two points within the box is (sqrt(2)\* 𝛿)/2, from the diagonal of the box. This contradicts with the minimum between-subset distance being 𝛿. So, the worst case of this method places one point in each box. So, if two points differ from each other more than 15 positions, then their distances will be at least (3𝛿)/2, which is greater than 𝛿. As a result of this claim, we need only to compute the distance between s and it’s next 15 points for any point s in Sy.

Module 7

# Fibonacci Number (Dynamic programming)

**Fibo(n)**

**If n < 2 return 1;**

**Return Fibo(n-1) + Fibo(n-2)**

* We can simply compute and memorize the results of these subproblems, which will lead to a very simple algorithm that runs in O(n)
* We simply use a one-dimensional array to store the subproblems f(1), f(2) ….f(n-1), when computing f(i), we simply add up its previous two numbers f(i-1) and f(i-2)
* Simply avoid the re-computation of overlapping subproblems

# Dynamic programming

* The idea of identifying overlapping subproblems, solve each of them only once and record the result to avoid re-computation, is key to the higher efficiency of dynamic programming algorithm
* Trade space for time in dynamic programming, so we need to analyze both time and space complexity
* Which property qualifies a proper subproblem
  + The sole property that properly defines a subproblem is called optimal substructure
  + Also called subproblem optimality
  + Some subproblems have optimal substructure, some subproblems do not
  + The algorithm design schema we have learnt so far, including greedy algorithms, divide and conquer, and dynamic programming should all have optimal substructure
    - But they do have different substructure in terms of their subproblems
    - When one problem can be solved by greedy algorithms, it implies that solving a larger problem of it only requires the solution of one subproblem.
  + In terms of generalizability, divide and conquer > dynamic programming > greedy
  + In terms of efficiency, greedy > dynamic programming > divide and conquer
* For the same problem, some definition of subproblems does not have optimal substructure, and some definitions do.
* Steps
  + Define recursion
  + Initialization
  + traceback

# Knapsack problem (Dynamic programming)

* Define subproblem
  + We could define the subproblem as the same capacity with a subset of items
  + We could define the subproblem as the same set of items with a smaller capacity
  + We could define the subproblem as a smaller capacity with a subset of items
* Case1
  + Imagine the item set contains an item I, where wi = C, vi = infinity, and the subset of items defining the subproblem does not contain the item i
  + No matter what we have chosen for that subproblem, the solution is useless
    - Because the solution to the subproblem clearly does not contain i, while the optimal answer clearly does
* Case2
  + Imagine the item set contains an item I, where wi = C and vi = infinity, the weight of item I is larger than the capacity defining the subproblem
  + No matter that we have chosen for that subproblem the solution is useless
    - Because the solution to the subproblem clearly does not contain I, while the optimal answer clearly does
* Case3
  + Consider another larger problem (C + wi, I’ + i)
  + The solution to this problem, conditioned on that we take item I, is nothing else than including I tin to the solution obtained for (C’, I’)
    - Because under that condition, we are left with a capacity C’, and an item set I’, which define the original subproblem
* Hence the subproblem with a smaller capacity and subset of items does have an optimal substructure.
* The recursive function as the follows
* why consider item i first, but not j
  + it does matter because putting item I into the knapsack first followed by item j has the same effect as because putting item j into the knapsack first followed by item i
  + so the oder of the items does not matter
* Design
  + 2D matrix (dynamic programming table)
  + According the data dependency in the dynamic programming table, conceptually we should first compute the “northwestern” data blocks before marching ‘southeast”
  + If we compute the first column and the first row, we should be able to complete the table row-by-row
    - The first row and first column would correspond to special cases of the problem
    - The first row indicates the value we can have if the knapsack has 0 capacity, the answer would obviously 0
    - The first column indicates the scenario when we only have one item in the set, clearly, for cells with capacity less than w0, we cannot take the item and the value should be 0, for cells with capacities equal or greater than w0, we can take the item an the value should be v0
  + However, the algorithm only tells us the maximum value can achieve, but it does not tell which items should be taken to achieve that maximum value
    - We need include additional step called traceback to retrieve the evidence
    - Note that we are trying to reconstruct the evidence that corresponds to the optimal solution of the problem, therefore, we should **begin with the cell that stores the optimal solution.**

**Correctness**

* Similar to divide and conquer algorithm, the algorithm is correct if the recursion is correct

**Efficiency**

* The time complexity is the for filling up + time for trace back
* Filling up the table takes O(C \* |I|) entries, for each entry, we need O(1) time
* For trace back, we are moving either left or up by at least one position, so time complexity is O(C + |I|)
* Space complexity is clearly O(C|I|)
* Note on time complexity
  + Note that C is a single number rather than a set of items, which need O(n) bits to represent
  + So the time complexity is really which is exponential instead of polynomial
* Algorithm whose time complexity can be written in the form of polynomial but is in fact exponential, are called pseudo-polynomial algoritjhms
  + They are in theory classified as exponential algorithm

# Weighted Interval scheduling (Dynamic programming)

* Try to define a proper subproblem
  + There are basically two input restrictions that limit us from obtaining infinity profit, a limited amount of time and a limited set of jobs
  + And we should be defining the subproblem via reducing these two factors
    - An instance with a shorter time
    - An instance with a subset of jobs
    - An instance with both a shorter time and a subset of jobs
  + First investigate the subproblem defined by subset of jobs
    - Turns out unlikely to work
    - Because the solution we found for a subset of jobs could become invalidate if some conflicting new jobs added in
    - Also explains why the subproblem defined on both a shorter time and a subset of jobs does not work
  + We only choice remains is to use a shorter time as the subproblem
    - We still need to investigate whether it has an optimal substructure
    - Lets devote V(t’) as the maximum profit we can obtain up to time t’
    - The solution to the subproblem is obviously invariant, since for a given set of jobs and a given amount of time, the maximum profit once can make is fixed
  + A final restriction we should be placing on subproblem is that we expect the time cutoff to be finishing time of jobs
    - This is because no job can be scheduled between the finishing time of the last job and the time cutoff to increase profit
    - We may just shrink the time cutoff to the finishing time
    - And it discretize time and lead to only a linear number of subproblems
  + Next we shall see how can we construct solution for larger problem using solutions for smaller subproblems
  + For a given problem V(t), we know from the previous lemma that t=fi for some job ji, to define the set of subproblems, we should pre-sort the jobs based on their finishing time. The time needed for this step is O(nlogn) for n jobs
  + Now consider a larger problem V(t), and its associated job ji, there will be two cases we need to discuss
    - Either that ji is scheduled
    - Or that ji is not scheduled
  + Case 1
    - If ji is scheduled, we are sure that no more job can be scheduled after ji have started, so the subproblem we should be using is how much profit can we get using the time before ji is started, in other words, the subproblem V(si)
    - While si does not necessarily correspond to the finishing time for another job ji, we can shrink the time si to the latest time point t’ such that t’ = ji
    - If we define the last job whose ending time is before a given time point t as pre(t), then we can simply write down the recursive function for his scenario as:
  + Case 2
    - If ji is not scheduled, then it will not invalidate any jobs due to time conflict, and we can simply jump tp its immediately previous job
    - Then it is easy to write the recursive function for the second scenario where ji is not scheduled:
  + Taken together, we have the complete recursive function as:
  + Note on the function pre(.)
    - We can obtain a list of sorted finishing time (we need to sort the jobs based on their finishing time)
    - Then, for a given a time t, we will just perform binary search of t against the list to find the latest job shoes finishing time is before t (needs O(nlogn) time)
  + Now set up the dynamic programming table and the detailed algorithm
    - The table we need is one-dimensional, since we only have one variable (time), when defining subproblems
  + Traceback
    - We simply need to record Y or N for each cell, Y indicates that the maximum profit comes from the first scenario where the corresponding job is taken and N indicates the other scenario where it is not taken
    - Takes O(n) time
  + Time complexity
    - We have one-dimensional table to fill, which has a size of O(n)
    - To compute each cell, we need O(logn) time
    - Taken together, the overall time complexity is O(nlogn)
  + Space complexity
    - O(n)

# Sequence alignment

**Alignment(X,Y)**

**Array A[0 … m, 0… n]**

**Initialize A[i, 0]= iδ for each i**

**Initialize A[0, j]= jδ for each j**

**For j = 1, . . . , n**

**For i = 1, . . . , m**

**Use the recurrence (Rec) to compute A[i, j]**

**Endfor**

**Endfor**

**Return A[m, n]**

**Proposition:** Let M be any alignment of X and Y. If (m,n) ∉ M, then either the m­­­th position of X or the nth position of Y is not matched in M.

**Proof:** Suppose by way of contradiction that (m,n) ∉ M, and there are number i < m and j < n such that the pair (m,j) ∈ M and the pair (i, n) ∈ M. However, this contradicts our definition of alignment: we have (i,n), (m,j) ∈ M with i < m, but n > i so the pairs (i, n) and (m, j) cross.

There is an equivalent way to write the proposition that exposes three alternative possibilities and leads directly to the formulation of a recurrence:

In an optimal alignment M, at least one of the following is true:

* Case1: (m,n) ∈ M
* Case2: The mth position of X is not matched
* Case3: The nth position of Y is not matched.

Now, let OPT(i,j) denote the minimum cost of an alignment between x1­­­­x2…xi and y1y2…yi.

* If case 1 holds, then we pay (the mismatch cost for lining up xm and yn) and then also align x1x2…xm-1 as well as possible with y1y2…yn-1. We get OPT(m,n) = + OPT(m – 1, n – 1).
* If case2 holds, then we pay a gap cost of δ because the mth position of X is not matched. Then, we align x1x2…xm-1 as well as possible with y1y2…yn. In this way, we get OPT(m,n) = δ + OPT(m – 1, n).
* if case 3 holds, then we get OPT(m,n) = δ + OPT(m, n – 1).

Therefore, since all three cases hold, we have proven that if (m,n) ∉ M, then either the m­­­th position of X or the nth position of Y is not matched in M, where M is any alignment of X and Y.

Using the same argument for the subproblem of finding the minimum-cost alignment between x1x2…xi and y1y2…yj, we get the following fact:

The minimum alignment costs satisfy the following recurrence for i >= 1 and j >= 1:

OPT(i,j) = min[+ OPT(i – 1, j – 1), δ + OPT(i – 1, j), δ + OPT(i, j – 1)].

Moreover, (i,j) is in an optimal alignment M for this subproblem iff the minimum is achieved by the first of these values.

The dynamic programming algorithm is now clear: We build up the values of OPT(i,j) by using the recurrence. There are only O(mn) subproblems, and OPT(m,n) is the value that we are seeking.

We now specific the algorithm to compute the value of the optimal alignment. For initialization, we not that OPT(i,0) = OPT(0,j) = iδ for all i, because the only way to line up an i-letter word with a 0-letter word is to use i gaps.

For traceback, we can trace back through the array A using the second part of recurrence to construct the alignment itself.

**Analyzing the Algorithm**

The correctness of the algorithm follows directly from recurrence. The running time is O(m\*n), since the array A has O(m\*n) entries, and at worst we spend constant time on each. Assume specified what GXY is first

**Proposition:** Let f(i, j) denote the minimum cost of a path from (0,0) to (i, j) in GXY. Then for all i, j, we have f(i, j) = OPT(I, j).

**Proof:**

(base case) when i = j = 0, we have i = j = 0, and indeed f(i,j) = OPT(i,j) = 0.

(induction)Now consider the arbitrary values of i and j. Also, suppose the statement is true for all pairs (i’,j’) with i’ + j’ < i + j. The last edge on the shortest path to (i,j) is either from (i – 1, j – 1), (i – 1,j) or (i, j – 1). Therefore, we have

f(i,j) = min[ + f(i – 1,j – 1), δ + f(i – 1,j), δ + f(i,j – 1)]

= min[ + OPT(i – 1, j – 1), δ + OPT(i – 1,j), δ + OPT(i,j – 1)]

= OPT(i,j),

where we pass from the first line to the second by using the induction hypothesis, and we pass from the second to the third by using Recurrence.

(conclusion)Hence, we have proven that for all i, j, we have f(i, j) = OPT(i, j), when we let f(i, j) denote the minimum cost of a path from (0,0) to (i,j) in GXY.

The value of the optimal alignment is the length of the shortest path in GXY from (0, 0) to (m, n). (We’ll call any path in GXY from (0, 0) to (m, n) a corner-to-corner path.)