INT 102

Algorithmic Foundations and Problem Solving

By Baby Rui Cry Cry

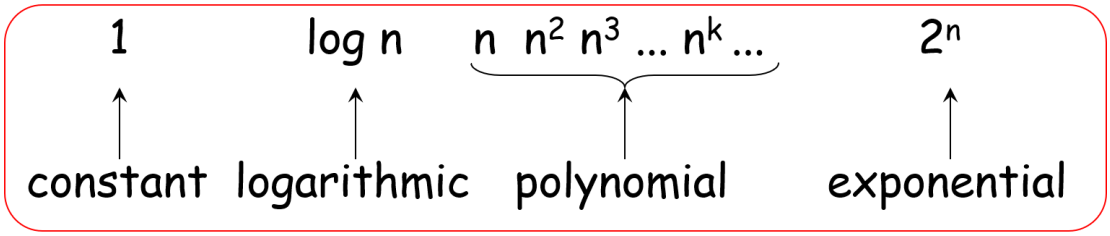
2021.06.02

Week1

Hierarchy of functions

Definition:

We can define a hierarchy of functions each having a greater order of magnitude than its predecessor:



We can further refine the hierarchy by inserting $n \log n$ between n and n^2, $\mathrm{n}^2 \log n$ between n^2 and n^3, and so on.
Important: as we move from left to right, successive functions have greater order of magnitude than the previous ones.
As n increases, the values of the later functions increase more rapidly than the values of the earlier ones. (Relative growth rates increase)
For example, $f(n) = 2n3 + 5n2 + 4n + 7$
The term with the highest power is $2n^3$.
The growth rate of $f(n)$ is dominated by n^3.
This concept is captured by Big-O notation.

Big-O notation

Definition:

$f(n) = O(g(n))$ [read as $f(n)$ is of order $g(n)$]
Roughly speaking, this means $f(n)$ is at most a constant times $g(n)$ for all large n.
Function on left and function on right are said to have the **same order of magnitude**.

Formal definition:

There exists a constant c and n_0, such that $f(n) \leq cg(n)$ for all $n > n_0$.
Examples:
$2n^3 = O(n^3)$
$3n^2 = O(n^2)$
$2n \log n = O(n \log n)$
$n^3 + n^2 = O(n^3)$

Time complexity:

Usually, we are only interested in the asymptotic time complexity, i.e., when n is large.
$\boldsymbol{O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(2^n)}$
Example:

(From Problem Session 1, Question 4)

List the following functions from lowest order to highest order of magnitude:

$O(1), O(n^2), O(n^3), O(n), O(\log n), O(\log^2 n), O(n \log n), O(\sqrt{n}), O\ (n^2 \log^4 n), O(n^{\frac{3}{2}}), O(n^{\frac{10}{3}}), O(n^4)$.

Solution:

$O(1), O(\log n), O(\log^2 n), O(\sqrt{n}), O(n), O(n \log n), O(n^{\frac{3}{2}}), O(n^2), O\ (n^2 \log^4 n), O(n^3), O(n^{\frac{10}{3}}), O(n^4)$.

Proof of order of magnitude:

Example:

(From Problem Session 1, Question 5)

Prove that the function $f(n) = 3n^2 \log n + 2n^3 + 3n^2 + 4n$ is $O(n^3)$.

(That is, show that there exist constants c and n_0 such that $f(n) \leq cn^3$ for all $n > n_0$.)

Solution:

To prove that $f(n) = 3n^2 \log n + 2n^3 + 3n^2 + 4n$ is $O(n^3)$, we show that there exists a constant c and n_0 such that for any integer $n > n_0$,

$3n^2 \log n + 2n^3 + 3n^2 + 4n \leq cn^3$

- $3n^2 \log n \leq 3n^3\ \forall n \geq 1$
- $2n^3 \leq 2n^3$
- $3n^2 \leq 3n^3\ \forall n \geq 1$
- $4n \leq 3n^3\ \forall n \geq 1$

As a result, $3n^2 \log n + 2n^3 + 3n^2 + 4n \leq 12n^3\ \forall n \geq 1$.

Since 12 is a constant, the function $3n^2 \log n + 2n^3 + 3n^2 + 4n$ is $O(n^3)$.

(**N.B.:** Should give reasons for the inequalities as shown above.)

Alternatively, we can prove in the following way:

- $3n^2 \log n \leq n^3\ \forall n \geq 16$
- $2n^3 \leq 2n^3\ \forall n$
- $3n^2 \leq n^3\ \forall n \geq 3$
- $4n \leq n^3\ \forall n \geq 2$

As a result, $3n^2 \log n + 2n^3 + 3n^2 + 4n \leq 5n^3\ \forall n \geq 16$.

Week2

Linear Search

Definition:

Input:
a sequence of n numbers $a_0, a_1, \dots, a_{n-1}$; and a number X.
Output:
determine whether X is in the sequence or not.
Algorithm:
1. Starting from $i = 0$, compare X with a_i one by one as long as $i < n$.
2. Stop and report "Found!" when $X = a_i$.
3. Repeat and report "Not Found!" when $i \geq n$.

Pseudocode:

```
i = 0
while i < n do
begin
    if X == a[i] then
        report "Found!" and stop
    else
        i = i+1
end
report "Not Found!"
```

Time Complexity:

Important operation of searching: **Comparison.**
Best case:
X is the 1st no. 1 comparison, $O(1)$.
Worst case:
X is the last no. or X is not found, n comparisons, $O(n)$.

Binary search

Definition:

More efficient way of searching when the sequence of numbers is **pre-sorted**.
Input:
a sequence of n **sorted** numbers $a_0, a_1, \dots, a_{n-1}$ in ascending order and a number X.
Algorithm:
1. Compare X with number in the middle.
2. Then, focus on only the first half or the second half (depend on whether X is smaller or greater than the middle number).
3. Reduce the amount of numbers to be searched by half.

Pseudocode:

```
first = 0, last = n-1
while (first <= last) do
begin
    mid = ⌊(first+last)/2)⌋
    if (X == a[mid])
          report "Found!" & stop
    else
        if (X < a[mid])
            last = mid-1
        else
            first = mid+1
end
report "Not Found!"
```

Time Complexity:

Best case:
X is the number in the middle => 1 comparison, $O(1)$
Worst case:
at most $|\log_2 n| + 1$ comparisons, $O(\log n)$
Every comparison reduces the amount of numbers by at least half.
E.g., 16 => 8 => 4 => 2 => 1

Binary search vs Linear search

Time complexity:

Time complexity of linear search is $O(n)$
Time complexity of binary search is $O(\log n)$
Therefore, binary search is **more efficient** than linear search.

String Matching

Definition:

Given a string of n characters called the text and a string of m characters ($m \leq n$) called the pattern. We want to determine if the text contains a substring matching the pattern.
Algorithm:

1. The algorithm scans over the text position by position.
2. For each position i, it checks whether the pattern $P[0..m-1]$ appears in $T[i..i+m-1]$.
3. If the pattern exists, then report found.
4. Else continue with the next position $i+1$.
5. If repeating until the end without success, report not found.

Pseudocode:

```
for i = 0 to n-m do
begin
    j = 0
    while (j < m && P[j] == T[i+j]) do
        j = j + 1
    if (j == m) then
        report "found!" & stop
end
report "Not found!"
```

Time complexity:

Best case:
pattern appears in the beginning of the text, $O(m)$.
Worst case:
pattern appears at the end of the text OR pattern does not exist, $O(nm)$.

Example:

(From Problem Session 1, Question 6)
Consider a string T of n characters $T[0..(n-1)]$. Design and write a pseudo code of an algorithm to determine if a given character, denoted by C, appears uniquely in $T[0..(n-1)]$, i.e., whether the character C appears exactly once in T.

Solution:

```
Algorithm: CC1(c, T[0..n-1])
//Determine if a given character, denoted by C, appears uniquely
//in T[0..n-1]
//Input: an array of n numbers T[0..n-1]
//Output: "Yes" if c appears uniquely in T[0..n-1], otherwise
"No"
    counter = 0
    for i = 0 to n-1
        if c == T[i] then
            counter = counter + 1
    if counter == 1 then
        return "Yes"
    else
        return "No"
```

OR

```
Algorithm: CC2(c, T[0..n-1])
//Determine if a given character, denoted by C, appears uniquely
//in T[0..n-1]
//Input: an array of n numbers T[0..n-1]
//Output: "Yes" if c appears uniquely in T[0..n-1], otherwise
//"No"
```

```
    counter = 0
    while counter < 2 do
        if c == T[i] then
            counter = counter + 1
    if counter == 1 then
        return "Yes"
    else
        return "No"
```

Selection Sort

Definition:

Input:
a sequence of n numbers $a_0, a_1, \ldots, a_{n-1}$.
Output:
arrange the n numbers into ascending order, i.e., from smallest to largest.
Algorithm:

1. Find minimum key from the input sequence.
2. Delete it from input sequence.
3. Append it to resulting sequence.
4. Repeat until nothing left in input sequence.

Pseudocode:

```
for i = 0 to n-2 do
begin
    min = i
    for j = i+1 to n-1 do
        if a[j] < a[min] then
        min = j
    swap a[i] and a[min]
end
```

Time complexity:

Total number of comparisons
$= (n-1) + (n-2) + \cdots + 1$
$= n(n-1)/2$
Time complexity is $O(n^2)$.

Bubble Sort

Definition:

Algorithm:

1. Starting from the last element, swap adjacent items if they are not in ascending order.

2. When first item is reached, the first item is the smallest.
3. Repeat the above steps for the remaining items to find the second smallest item, and so on.

Pseudocode:

```
for i = 0 to n-2 do
    for j = n-1 downto i+1 do
        if (a[j] < a[j-1])
            swap a[j] & a[j-1]
```

Time complexity:

Total number of comparisons
$= (n-1) + (n-2) + \cdots + 1$
$= n(n-1)/2$
Time complexity is $O(n^2)$.

Insertion Sort (optional)

Definition:

Algorithm:
1. Look at elements one by one.
2. Build up sorted list by inserting the element at the correct location.

Pseudocode:

```
for i = 1 to n-1 do
begin
    key = a[i]
    pos = 0
    while (a[pos] < key) && (pos < i) do
        pos = pos + 1
    shift a[pos], …, a[i-1] to the right
    a[pos] = key
end
```

Time complexity:

Worst case input is that input is sorted in descending order. Then, for $a[i]$ finding the position takes i comparisons.
Total number of comparisons
$= 1 + 2 + \cdots + (n-1)$
$= n(n-1)/2$
Time complexity is $O(n^2)$.

Selection, Bubble, Insertion Sort

Time complexity:

All three algorithms have time complexity $O(n^2)$ in the worst case.
The time complexity of the fastest comparison-based sorting algorithm is $O(n \log n)$.

Week3

Recurrence

Definition:

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

E.g.,

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

To solve a recurrence is to derive asymptotic bounds on the solution.

Example:

1. $\boldsymbol{T(n) = T(n/2) + 1, T(n)\ is\ O(\log n)}$
2. $\boldsymbol{T(n) = 2T(n/2) + 1, T(n)\ is\ O(n)}$
3. $\boldsymbol{T(n) = 2T(n/2) + n, T(n)\ is\ O(n \log n)}$

Solution:

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

Solution (Substitution method):

Make a guess, $T(n) \leq 2 \log n$.

We prove statement by MI.

Assume true for all $n' < n$ [assume $T(n/2) \leq 2 \log(n/2)$]

$T(n) = T(n/2) + 1$

$\leq 2 \log(n/2) + 1$ (by hypothesis)

$\leq 2(\log n - 1) + 1$ ($\log(n/2) = \log n - \log 2$)

$< 2 \log n$

Therefore, $T(n) \leq 2 \log n$.

Solution (Iterative method):

(From Assessment 1, Problem 6)

$T(n) = T(n/2) + 1$

$T(n) = T(n/2^2) + 2$

$T(n) = T(n/2^3) + 3$

$T(n) = T(n/2^k) + k$

Assume that $n/2^k = 1$, we get $k = \log n$.

$T(n) = T(1) + log(n) = log(n) + 1$

Therefore, $T(n) = O(log(n))$.

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

Solution (Substitution method):

Make a guess, $T(n) \leq 2n - 1$.

Assume true for all $n' < n$ [assume $T(n/2) \leq 2(n/2) - 1$]

$T(n)$

$\leq 2(2 * (n/2) - 1) + 1$

$= 2n - 2 + 1$

$= 2n - 1$

Therefore, $T(n) \leq 2n-1$.

Solution (Iterative method):

(From Problem Session 2, Question 4)

$T(n) = 2T(n/2) + 1$

$= 2(2(T(n/2^2)) + 1) + 1$

$= 2^2T(n/2^2) + 2 + 1$

$= 2^3T(n/2^3) + 2^2 + 2 + 1$

… …

$= 2^kT(n/2^k) + 2^{k-1} + \cdots + 2^2 + 2 + 1$

… …

$= 2^{\log(n)} + 2^{\log(n)-1} + 2^2 + 2 + 1$

$= 2^{\log(n)+1} - 1$

$= 2n - 1$

(Note that $2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$)

Therefore, $T(n)\ is\ O(n)$.

3. $\boldsymbol{T(n) = 2T(n/2) + n}$

Solution (Substitution method):

Make a guess, $T(n) \leq 2n\log n$.

Assume true for all $n' < n$ [assume $T(n/2) \leq 2(n/2)\log(n/2)$]

T(n)

$\leq 2(2(n/2)\log(n/2)) + n$

$= 2n(\log n - 1) + n$

$= 2n\log n - 2n + n$

$\leq 2n\log n$

Therefore, $T(n) \leq 2n\log n$.

Solution (Iterative method):

(From Problem Session 2, Question 3)

$T(n) = 2T(n/2) + n$

$= 2(2(T(n/2^2)) + n/2) + n$

$= 2^2T(n/2^2) + 2n$

$= 2^2(2T(n/2^3) + n/2^2) + 2n$

$= 2^3T(n/2^3) + 3n$

… …

$= 2^kT(n/2^k) + kn$

$= 2^kT(1) + kn$

$= 2^k + kn$

$= n + n\log n < n\log n \quad for\ n > 1$

Therefore, $T(n)\ is\ O(\log n)$.

Divide and Conquer

Idea:

- A problem instance is divided into several smaller instances of the same problem, ideally of about same size.

- The smaller instances are solved, typically recursively.
- The solutions for the smaller instances are combined to get a solution to the original problem.

Recursive Binary Search (RBS)

Definition:

(See Week2, Binary Search, Definition)

Pseudocode:

```
RecurBinarySearch(A, first, last, X)
begin
    if (first > last) then
        return false
    mid = ⌊(first + last)/2⌋
    if (X == A[mid]) then
        return true
    if (X < A[mid]) then
        return RecurBinarySearch(A, first, mid-1, X)
    else
        return RecurBinarySearch(A, mid+1, last, X)
end
```

Time complexity:

Let $T(n)$ denote the time complexity of binary search algorithm on n numbers.

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

Time complexity is $O(\log n)$.

Merge sort

Definition:

Algorithm:

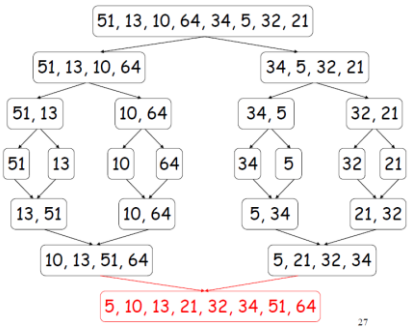
1. Using divide and conquer technique.
2. Divide the sequence of n numbers into two halves.
3. Recursively sort the two halves.
4. Merge the two sort halves into a single sorted sequence.

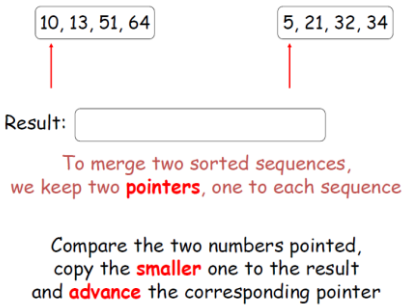
Divide:

Dividing a sequence of n numbers into two smaller sequences is straightforward.

Conquer:

Merging two sorted sequences of total length n can also be done easily, at most $n - 1$ comparisons.





Pseudocode:

```
Algorithm Mergesort(A[0..n-1])
    if n > 1 then begin
        copy A[0..⌊n/2⌋-1] to B[0..⌊n/2⌋-1]
        copy A[⌊n/2⌋..n-1] to C[0..⌈n/2⌉-1]
        Mergesort(B[0..⌊n/2⌋-1])
        Mergesort(C[0..⌈n/2⌉-1])
        Merge(B, C, A)
end
```

```
Algorithm Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])
    Set i=0, j=0, k=0
    while i<p and j<q do
    begin
        if B[i] <= C[j] then set A[k] = B[i] and increase i
        else set A[k] = C[j] and increase j
        k = k+1
    end
    if i == p then copy C[j..q-1] to A[k..p+q-1]
    else copy B[i..p-1] to A[k..p+q-1]
end
```

Time complexity:

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

Time complexity is $O(n \log n)$.

Undirected Graphs

Definition:

An **undirected** graph $G = (V, E)$ consists of a set of vertices V and a set of edges E. Each edge is an **unordered** pair of vertices. (E.g., $\{b, c\}$ & $\{c, b\}$ refer to the same edge.)

Type:

Simple graph:

At most one edge between two vertices, no self loop (i.e., an edge from a vertex to itself).

Multigraph:

Allows more than one edge between two vertices.

Pseudograph:

Allows a self loop.

Terminology:

In an undirected graph G, suppose that $e = \{u, v\}$ is an edge of G.

1. u and v are said to be **adjacent** and called **neighbors** of each other.
2. u and v are called **endpoints** of e.
3. e is said to be **incident** with u and v.
4. e is said to **connect** u and v.
5. The **degree** of a vertex v, denoted by $\boldsymbol{deg(v)}$, is the number of edges incident with it (a loop contributes twice to the degree).

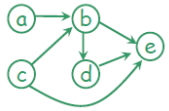
Directed graph

Definition:

A **directed** graph $G = (V, E)$ of a set of vertices V and a set of edges E. Each edge is an **ordered** pair of vertices. (E.g., (b, c) refer to an edge from b to c.)

In/Out degree

The **in-degree** of a vertex v is the number of edges **leading to** the vertex v.
The **out-degree** of a vertex v is the number of edges **leading away** from the vertex v.



	in-deg(v)	out-deg(v)
a	0	1
b	2	2
c	0	2
d	1	1
e	3	0
sum:	6	6

Representation

Definition:

An undirected graph can be represented by **adjacency matrix**, **adjacency list**, **incidence matrix** or **incidence list**.

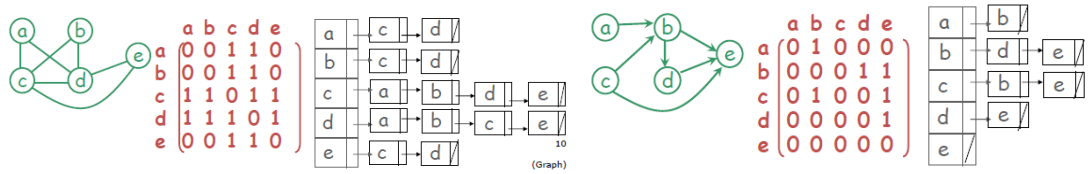
Adjacency matrix:

M for a simple undirected/directed graph with n vertices:

1. M is an $n \times n$ matrix.
2. $M(i, j) = 1$ if vertex i and vertex j are adjacent. **(Undirected graph)**
3. $M(i, j) = 1$ if (i, j) is an edge. **(Directed graph)**

Adjacency list:

1. Each vertex has a list of vertices to which it is adjacent. **(Undirected graph)**
2. Each vertex u has a list of vertices pointed to by an edge leading away from u. **(Directed graph)**



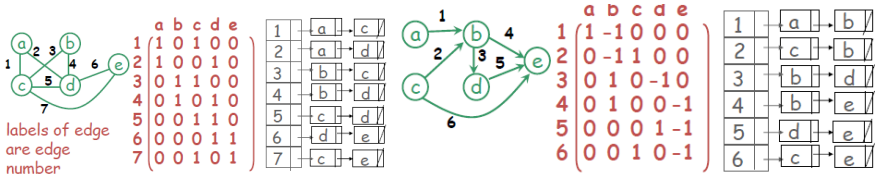
Incidence matrix:

M for a simple undirected/directed graph with n vertices and m edges:

1. M is an $m \times n$ matrix.
2. $M(i,j) = 1$ if edge i and vertex j are incidence. **(Undirected graph)**
3. $M(i,j) = 1$ if edge i is leading away from vertex j. **(Directed graph)**
4. $M(i,j) = -1$ if edge i is leading to vertex j. **(Directed graph)**

Incidence list:

1. Each edge has a list of vertices to which it is incident with. **(Undirected graph)**
2. Each edge has a list of two vertices (leading away is 1st and leading to is 2nd) **(Directed graph)**



Paths, circuits (in undirected graphs)

Definition:

1. In an undirected graph, a path from a vertex u to a vertex v is a sequence of edges $e_1 = \{u, x_1\}, e2 = \{x_1, x_2\}, \dots e_n = \{x_{n-1}, v\}$, where $n \geq 1$.
2. The length of this path is n.
3. Note that a path from u to v implies a path from v to u.
4. If $u = v$, this path is called a circuit (cycle).
5. If every pair of vertices have a path, then Graph is called connected.

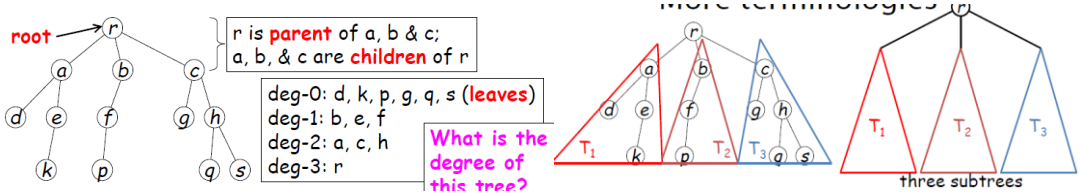
Tree

Definition:

An undirected graph $G = (V, E)$ is a tree if G is connected and acyclic (i.e., contains no cycles).

Terminology:

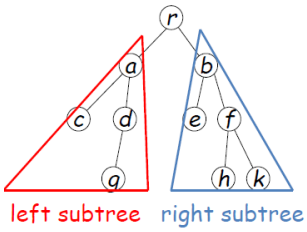
1. Topmost vertex is called the **root**.
2. A vertex u may have some **children** directly below it, u is called the **parent** of its children.
3. **Degree** of a vertex is the no. of children it has. (N.B. it is different from the degree in an unrooted tree.)
4. Degree of a tree is the max. degree of all vertices.
5. A vertex with no child (degree-0) is called a **leaf**. All others are called **internal vertices**.
6. We can define a tree **recursively**.
   a) A single vertex is a tree.
   b) If $T_1, T_2, \ldots, T_k$ are **disjoint** trees with roots $r_1, r_2, \ldots, r_k$, the graph obtained by attaching a new vertex r to each of $r_1, r_2, \ldots, r_k$ with a single edge forms a tree T with root r.
   c) $T_1, T_2, \ldots, T_k$ are called **subtrees** of T.



Binary tree

Definition:

1. A tree of degree at most **TWO**.
2. The two subtrees are called left subtree and right subtree (may be empty).



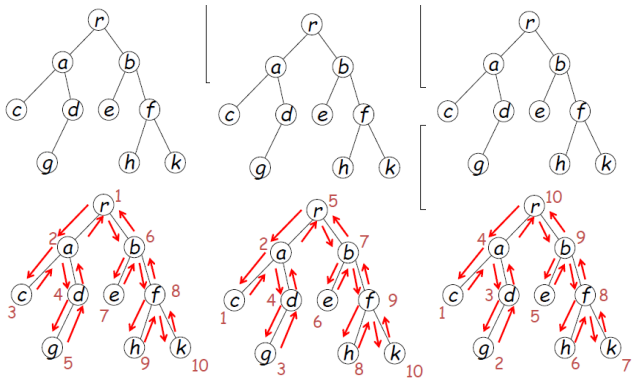
Traverse:

1. Preorder traversal:
   vertex, left subtree, right subtree
2. Inorder traversal:
   left subtree, vertex, right subtree
3. Postorder traversal:
   left subtree, right subtree, vertex

Example:

1. Preorder traversal:
   r → a → c → d → g → b → e → f → h → k
2. Inorder traversal:
   c → a → g → d → r → e → b → h → f → k

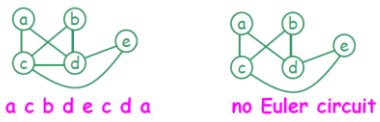
3. Postorder traversal:
   c → g → d → a → e → h → k → f → b → r



Euler circuit

Definition:

1. A **simple** circuit visits an edge **at most once**.
2. An **Euler** circuit in a graph G is a circuit visiting every edge of G **exactly** once. (NB. A vertex can be repeated.)



Condition:

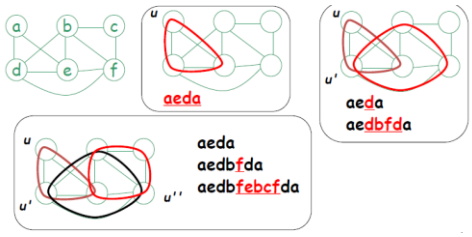
Trivial condition:

1. An undirected graph G is said to be **connected** if there is a path between **every pair** of vertices.
2. If G is **not** connected, there is no single circuit to visit all edges or vertices.
3. Even if the graph is connected, there may be no Euler circuit either.

Necessary and sufficient condition:

Let G be a connected graph.

Lemma: G contains an Euler circuit if and only if every vertex has **even** degree.



Euler path

Definition:

1. Let G be an undirected graph.
2. An **Euler path** is a path visiting every edge of G exactly once.

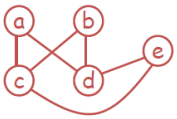
Condition:

Necessary and sufficient condition:

1. An undirected graph contains an Euler path if it is connected and contains **exactly two vertices of odd degree**.

This graph has no Euler circuit, but has an Euler path

c b d a c e d



Hamiltonian circuit / path

Definition:

Let G be an undirected graph.

1. A **Hamiltonian circuit** (path) is a circuit (path) containing every vertex of G exactly once.
2. Note that a Hamiltonian circuit or path may **NOT** visit all edges.
3. Unlike the case of Euler circuits / paths, determining whether a graph contains a Hamiltonian circuit (path) is a very **difficult** problem. (**NP-hard**)

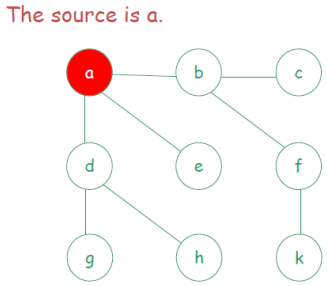
Week4

Breadth-first search (BFS)

Definition:

Given $G = (V, E)$, and a distinguished source vertex s, BFS systematically explores the edges of G such that all vertices at **distance** k from s are explored **before** any vertices at **distance** $k + 1$.

Example:



Order of exploration: a, b, e, d, c, f, h, g, k

Pseudocode:

```
unmark all vertices
choose some starting vertex s
mark s and insert s into tail of list L
while L is nonempty do
begin
    remove a vertex v from front of L
    visit v
    for each unmarked neighbor w of v do
    mark w and insert w into tail of list L
end
```

Formal one:

```
ALGORITHM BFS(G)
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = <V, E>
//Output: Graph G with its vertices marked with consecutive
//integers in the order they are visited by the BFS traversal
//mark each vertex in V with 0 as a mark of being "unvisited"
count ←0
for each vertex v in V do
    if v is marked with 0
        bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are visited
```

```
//via global variable count
count ←count + 1; mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ←count + 1; mark w with count
            add w to the queue
    remove the front vertex from the queue
```

Depth First Search (DFS)

Definition:

Depth-first search searches **"deeper"** in the graph whenever possible.

1. Edges are explored from the most recently discovered vertex v that still has unexplored edges leaving it.
2. When all edges of v have been explored, the search **"backtracks"** to explore edges leaving the vertex from which v was discovered.

Pseudocode:

```
ALGORITHM DFS(G)
//Implements a depth-first search traversal of a given graph
//Input: Graph G = <V, E>
//Output: Graph G with its vertices marked with consecutive
//integers in the order they are first encountered by the DFS
//traversal mark each vertex in V with 0 as a mark of being
//"unvisited"
count ←0
for each vertex v in V do
    if v is marked with 0
        dfs(v)

dfs(v)
//visits recursively all the unvisited vertices connected to
//vertex v by a path and numbers them in the order they are
//encountered via global variable count
count ←count + 1; mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)
```

Week5

Greedy methods

Definition:

How to be greedy?

1. At every step, make the best move you can make.
2. Keep going until you're done

Advantages:

1. Don't need to pay much effort at each step.
2. Usually finds a solution very quickly.
3. The solution found is usually not bad.

Possible problem:

1. The solution found may NOT be the best one.

Design:

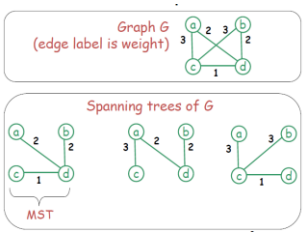
1. Cast the optimization problem as one in which we make a choice and are left with **one** subproblem to solve.
2. Demonstrate that, having make the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Minimum Spanning tree (MST)

Definition:

1. Given an undirected connected graph G, the edges are labelled by weight.
2. Spanning tree of G is a tree containing all vertices in G.
3. Minimum spanning tree is a spanning tree of G with minimum weight.

Example:



Theorem:

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weights on edges. Let A be a subset of E that is included in some minimal spanning tree for G, let S be a subset of V and contain V_A, where $V_A = \{v : there\ is\ an\ edge\ e = (u, v)\ in\ A\}$. If (u^*, v^*) is one of the edges with minimum weight among the edges connecting S and $V - S$, then $A \cup \{(u^*, v^*)\}$ is contained in a minimum spanning tree.

Prim's algorithm

Pseudocode:

```
// Given a weighted connected graph G=(V,E)
pick a vertex v0 in V
VT = {v0}
ET =  ∅
For i=1 to |V|-1 do
    pick an edge e = (v*, u*) with minimum weight among all the
    edges (v, u) such that v is in V1 and u is in V-VT
    VT = VT∪{v*}
    ET = ET∪{e*}
Return ET
```

Correctness:

Observation (loop invariant):
Prior to each loop, E_T is a subset of a minimal spanning tree.

Complexity:

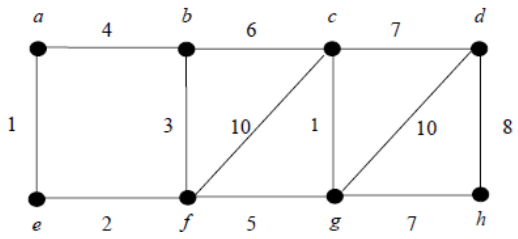
Given $G = (V, E)$ and if the following part is implemented by min-heap (for priority queue), then the answer is $O(|E| \log |V|)$.

Pseudocode (with data structure):

```
MST-PRIM(G, w, root) //G=(V,E)
for each u in V
    do key[u] <- ∞
       π[u] <- NIL
    key[root] <- 0
    Q <- V //implement by min-heap, O(|V|)
    while Q is not empty
       do u <- Extract-min(Q) //O(lg(|V|) by heap
          for each v in Adj[u]
          do if v in Q and w(u,v) < key[v]
             then π[v] <- u
                key[v] = w(u,v)
```

Example:

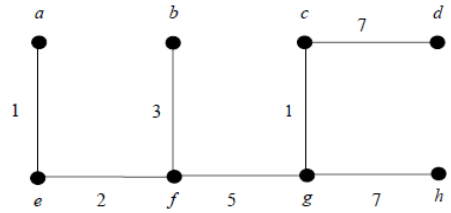
(From Problem Session 3, Question 4.1)
Consider the following graph G. The label of an edge is the cost of the edge.



Using Prim's algorithm, draw a minimum spanning tree (MST) of the graph. Also write down the change of the priority queue step by step and the order in which the vertices are selected. Is the MST drawn unique? (i.e., is it the one and only MST for the graph?)

Solution:

Order Selected	$a(0,-)$	$b(-,\infty)$	$c(-,\infty)$	$d(-,\infty)$	$e(-,\infty)$	$f(-,\infty)$	$g(-,\infty)$	$h(-,\infty)$
$a(0,-)$		$b(a,4)$	$c(-,\infty)$	$d(-,\infty)$	$e(a,1)$	$f(-,\infty)$	$g(-,\infty)$	$h(-,\infty)$
$e(a,1)$		$b(a,4)$	$c(-,\infty)$	$d(-,\infty)$		$f(e,2)$	$g(-,\infty)$	$h(-,\infty)$
$f(e,2)$		$b(f,3)$	$c(f,10)$	$d(-,\infty)$			$g(f,5)$	$h(-,\infty)$
$b(f,3)$			$c(b,6)$	$d(-,\infty)$			$g(f,5)$	$h(-,\infty)$
$g(f,5)$			$c(g,1)$	$d(g,10)$				$h(g,7)$
$c(g,1)$				$d(c,7)$				$h(g,7)$
$d(c,7)$								$h(g,7)$
$h(g,7)$								



This is the only MST.

Kruskal's algorithm

Definition:

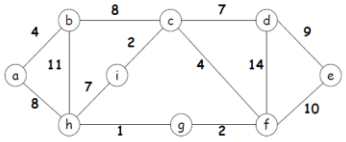
Kruskal's algorithm is greedy in the sense that it always attempts to select the smallest weight edge to be included in the MST.

Example:

1. Arrange the edges from smallest to largest weight.
2. Choose the minimum weight edge.
3. Choose the next minimum weight edge.
4. Continue as long as no cycle forms.
5. …
6. (h,i) cannot be included, otherwise, a cycle is formed.
7. (a,h) cannot be included, …
8. (f,e) cannot be included, …

9. (d, f) cannot be included, …
10. MST is found when all edges are examined.

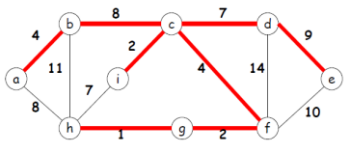
(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14



Arrange the edges from smallest to largest weight

21

(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
(h,i)	7
(b,c)	8
(a,h)	8
(d,e)	9
(f,e)	10
(b,h)	11
(d,f)	14



MST is found when all edges are examined

35

Pseudocode:

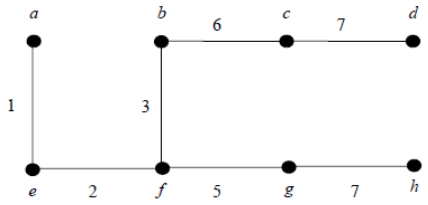
```
// Given an undirected connected graph G=(V,E)
pick an edge e in E with minimum weight
T = {e} and E' = E-{e}
while E' ≠ ∅ do
begin
    pick an edge e in E' with minimum weight
    if adding e to T does not form cycle then
        T = T∪{e}
    E' = E'-{e}
End
```

Time complexity is $O(nm)$.

Example:

(From Problem Session 3, Question 4.2)
(See Week5, Prim's algorithm, Example)
Using Kruskal's algorithm, draw a minimum spanning tree (MST) of the graph G. Write down the order in which the edges are selected.
Is the MST drawn unique? (i.e., is it the one and only MST for the graph?)
Solution:

(a, e)	1
(c, g)	1
(e, f)	2
(b, f)	3
~~(a, b)~~	~~4~~
(f, g)	5
(b, c)	6
(c, d)	7
(g, h)	7
~~(d, h)~~	~~8~~
~~(c, f)~~	~~10~~
~~(d, g)~~	~~10~~

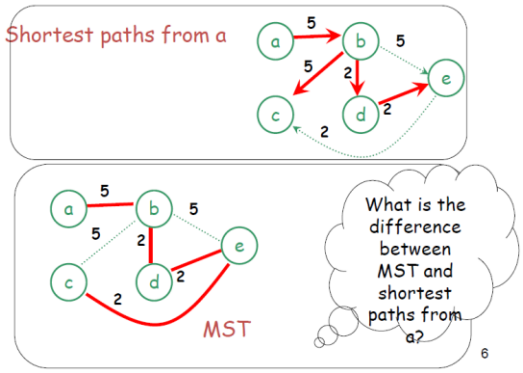


Single-source shortest-paths

Definition:

1. Consider a (un)directed connected graph G where the edges are labelled by weight.
2. Given a particular vertex called the **source**.
3. Find **shortest paths** from the source to all other vertices (shortest path means the total weight of the path is the smallest)

Example:



Optimal substructure:

A shortest path between two vertices contains other shortest paths within it.

Dijkstra's algorithm

Definition:

1. Input: A directed connected weighted graph G and a source vertex s.
2. Output: For every vertex v in G, find the shortest path from s to v.
3. Dijkstra's algorithm runs in iterations:
   a) In the i-th iteration, the vertex which is the i-th closest to s is found.
   b) For every remaining vertices, the current shortest path to s found so far (this shortest path will be updated as the algorithm runs).

Pseudocode:

Each vertex v is labelled with two labels:

1. a numeric label $d(v)$ indicates the length of the shortest path from the source to v found so far.
2. another label $p(v)$ indicates next-to-last vertex on such path, i.e., the vertex immediately before v on that shortest path.

```
// Given a graph G=(V,E) and a source vertex s
for every vertex v in the graph do
    set d(v) = ∞ and p(v) = null
set d(s) = 0 and V_T = ∅
while V-V_T ≠ ∅ do   // there is still some vertex left
begin
    choose the vertex u in V-V_T with minimum d(u)
```

```
    set V_T = V_T ∪ {u}
    for every vertex v in V-V_T that is a neighbour of u do
        if d(u) + w(u,v) < d(v) then  // a shorter path is found
            set d(v) = d(u) + w(u,v) and p(v) = u
end
```

Correctness:

Observation (loop invariant):
Prior to each loop, for each $v \in S$, $d[v]$ is the length of the shortest path from s to v.

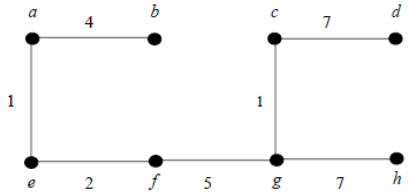
Complexity:

Depends on the data structure to represent the graph $G = (V, E)$ and the implementation of priority queue.
if G is represented by adjacency list and priority queue is implemented by min-heap., then the answer is $O(|E| \log |V|)$.

Example:

(From Problem Session 3, Question 4.3)
(See Week5, Prim's algorithm, Example)
Referring to the same graph above, find the shortest paths from the vertex a to all other vertices in the graph G using Dijkstra's algorithm. Show the changes of the priority queue step by step and give the order in which edges are selected.
Solution:

Order Selected	$a(0,-)$	$b(-,\infty)$	$c(-,\infty)$	$d(-,\infty)$	$e(-,\infty)$	$f(-,\infty)$	$g(-,\infty)$	$h(-,\infty)$
$a(0,-)$		$b(a,4)$	$c(-,\infty)$	$d(-,\infty)$	$e(a,1)$	$f(-,\infty)$	$g(-,\infty)$	$h(-,\infty)$
$e(a,1)$		$b(a,4)$	$c(-,\infty)$	$d(-,\infty)$		$f(e,3)$	$g(-,\infty)$	$h(-,\infty)$
$f(e,3)$		$b(a,4)$	$c(f,13)$	$d(-,\infty)$			$g(f,8)$	$h(-,\infty)$
$b(a,4)$			$c(b,10)$	$d(-,\infty)$			$g(f,8)$	$h(-,\infty)$
$g(f,8)$			$c(g,9)$	$d(g,18)$				$h(g,15)$
$c(g,9)$				$d(c,16)$				$h(g,15)$
$d(c,16)$								$h(g,15)$
$h(g,15)$								



Week6

Fibonacci numbers

Definition:

Fibonacci number F(n)

$$F(n) = \begin{cases} 1 & if\ n = 0\ or\ 1 \\ F(n-1) + F(n-2) & if\ n > 1 \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9	10
$F(n)$	1	1	2	3	5	8	13	21	34	55	89

Pseudocode:

```
Procedure F(n)
    if n==0 or n==1 then
        return 1
    else
        return F(n-1) + F(n-2)
```

```
Procedure F(n)
    Set A[0] = A[1] = 1
    for i = 2 to n do
        A[i] = A[i-1] + A[i-2]
    return A[n]
```

Complexity:

Recursive version:

$f(n) = f(n-1) + f(n-2) + 1$

$= [f(n-2) + f(n-3) + 1] + f(n-2) + 1$

$> 2f(n-2)$

$> 2[2f(n-2-2)] = 2^2 f(n-4)$

$> 2^2[2f(n-4-2)] = 2^3 f(n-6)$

...

$> 2^k f(n-2k)$

If n is even, $f(n) > 2^{n/2} f(0) = 2^{n/2}$

If n is odd, $f(n) > f(n-1) > 2^{(n-1)/2}$

Time complexity is $O(2^n)$.

Dynamic Programming version:

Time complexity is $O(n)$.

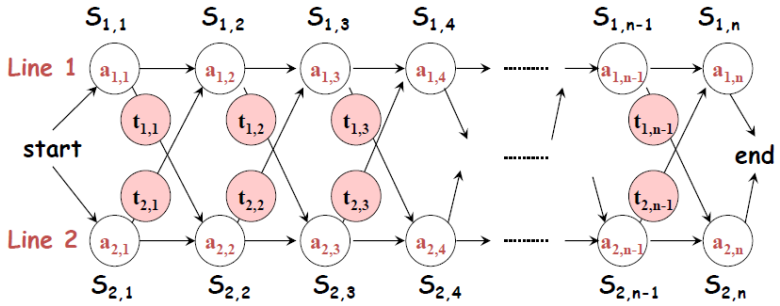
Assembly line scheduling

Definition:

1. 2 assembly lines, each with n stations ($S_{i,j}$: line i station j) $S_{1,j}$ and $S_{2,j}$ perform

same task but time taken is different.

2. $a_{i,j}$: assembly time at $S_{i,j}$.
3. $t_{i,j}$: transfer time after $S_{i,j}$.
4. Problem: To determine which stations to go in order to **minimize** the total time through the n stations.



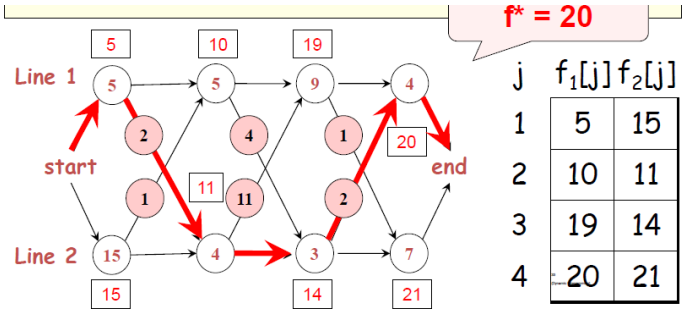
Dynamic Programming

$$f_1[j] = \begin{cases} a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j > 1 \end{cases}$$

$$f_2[j] = \begin{cases} a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j > 1 \end{cases}$$

$f^* = \min(f_1[n], f_2[n])$

Example:



j	$f_1[j]$	$f_2[j]$
1	5	15
2	10	11
3	19	14
4	20	21

Pseudocode:

```
set f1[1] = a1,1
set f2[1] = a2,1
for j = 2 to n do
begin
    set f1[j] = min(f1[j-1]+a1,j, f2[j-1]+t2,j-1+a1,j)
    set f2[j] = min(f2[j-1]+a2,j, f1[j-1]+t1,j-1+a2,j)
end
set f* = min (f1[n], f2[n])
```

Time complexity is $O(n)$.

3 or more lines

In general, m assembly lines: use multi-dimensional arrays.

1. $a[i][j]$: represents assemble time of station j on line i.
2. $t[i][j][k]$: represents transfer time from station j on line i to station $(j+1)$ on line k

$(t[i][j][i] = 0)$.

3. $f[i][j]$: represents the best so far way of going to station j on line i.

$f[i][j] = min_{1 \le k \le m}(f[k][j-1] + t[k][j-1][i] + a[i][j])$

Week8

Space-for-time tradeoffs

Type:

1. **Input-enhancement:** Preprocess the input (or its part) to store some info to be used later in solving the problem.
2. **Pre-structuring:** Preprocess the input using a data structure to make accessing its elements easier.

Sorting

Definition:

Input:
a sequence of n numbers a_0,a_1,…,a_(n-1).
Output:
arrange the n numbers into ascending order, i.e., from smallest to largest.

Complexity:

Insertion sort: $O(n^2)$
Selection sort: $O(n^2)$
Bubble sort: $O(n^2)$
Merge sort: $O(n \log n)$
Quick sort: $O(n \log n)$
All the sorting algorithms above are **comparison sorts**.
In fact, we can prove that **any comparison sorting** takes at least $\boldsymbol{O(n \log n)}$ time in the worst case.

Counting sort

Pseudocode:

```
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1   # C[i] = |{key = i}|
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]    # C[i] = |{key ≤ i}|
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```

Complexity:

```
O(k)   for i ← 1 to k
         do C[i] ← 0
O(n)   for j ← 1 to n
         do C[A[j]] ← C[A[j]] + 1
O(k)   for i ← 2 to k
         do C[i] ← C[i] + C[i–1]
O(n)   for j ← n downto 1
         do B[C[A[j]]] ← A[j]
            C[A[j]] ← C[A[j]] – 1
O(n + k)
```

Time complexity is $O(n+k)$.

Example:

(From Problem Session 4, Question 2)

Assuming that the set of possible list is $\{a, b, c, d\}$, sort the following list in alphabetical order by the counting algorithm: b, c, d, c, b, a, a, b.

Solution:

Auxiliary array C:

a	b	c	d
2	3	2	1

a	b	c	d
2	5	7	8

Changes in Output array

1	2	3	4	5	6	7	8
				b			

1	2	3	4	5	6	7	8
	a			b			

1	2	3	4	5	6	7	8
a	a			b			

1	2	3	4	5	6	7	8
a	a		b	b			

1	2	3	4	5	6	7	8
a	a		b	b		c	

1	2	3	4	5	6	7	8
a	a		b	b		c	d

1	2	3	4	5	6	7	8
a	a		b	b	c	c	d

1	2	3	4	5	6	7	8
a	a	b	b	b	c	c	d

Changes in Auxiliary array C

a	b	c	d
2	4	7	8

a	b	c	d
1	4	7	8

a	b	c	d
0	4	7	8

a	b	c	d
0	3	7	8

a	b	c	d
0	3	6	8

a	b	c	d
0	3	6	7

a	b	c	d
0	3	6	7

a	b	c	d
0	3	5	7

Horspool's Algorithm

Definition:

1. Preprocesses pattern to generate a shift table that determines how far to shift the pattern when a mismatch occurs.
2. Always makes a shift based on the text's character c aligned with the last character in the pattern according to the shift table's entry for c.

Example:

The character is not in the pattern

```
.....c..................... (c not in pattern)
BAOBAB
      BAOBAB
```

The character is in the pattern (but not the rightmost)

```
.....O..................... (O occurs once in pattern)
BAOBAB
   BAOBAB
.....A..................... (A occurs twice in pattern)
BAOBAB
 BAOBAB
```

The rightmost characters do match

```
.....B.....................
BAOBAB
  BAOBAB
```

Shift Table:

For a pattern $P[0..m-1]$, shift size $s(c)$ of a letter c in the text can be precomputed as following:

1. If c is in P$[0..m-2]$:
   $s(c) =$ The number of characters from c's rightmost occurrence in $P[0..m-2]$ to the right end of the pattern $P[0..m-1]$.
2. If c is not in $P[0..m-2]$:
   $s(c) =$ The length m of the pattern $P[0..m-1]$

$s(c)$ is stored in a so-called shift table indexed by text and pattern alphabet.

Example:

Let the text and pattern consists of alphabet $\{A, \dots, Z\}$

Consider the pattern $P[0..5]$ = BAOBAB

Shift table:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	-
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

OR

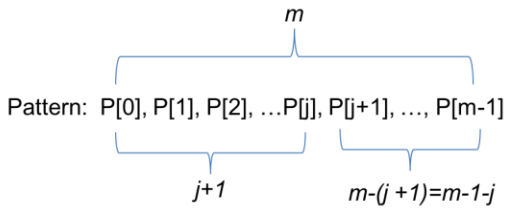
A	B	O	*
1	2	3	6

Pseudocode:

```
Algorithm ShiftTable(P[0..m-1])
    //Fills the shift table used by Horspool's algorithm
    //Input: Pattern P[0..m-1] and an alphabet of possible
    //characters
    //output: Table[0..size-1] indexed by the alphabet's
    //characters and filled //with the shift sizes computed as
    //before.

    Initialize all the elements of Table with m

    for j=0 to m-2 do Table[P[j]]=m-1-j
    Return Table
```




```
Algorithm HorspoolMatching(P[0..m-1], T[0..n-1])
    //Inmplements Horspool's algorithm for string matching
    //Input: Pattern P[0..m-1] and text T[0..n-1]
    //Output: The index of the left end of the first matching
    //substring or -1 if there are no matches
    ShiftTable(P[0..m-1])   //generate Table of shifts
    i ← m-1                 //position of the pattern's right end
    while i ≤ n-1 do
        k ← 0               //number of matched characters
        while k ≤ m-1 and P[m-1-k] = T[i-k] do
            k ← k+1
        if k=m
            return i-m+1
        else i ← i+Table[T[i]]
    return -1
```

Complexity:

1. The worst-case complexity: $O(mn)$.
2. For random texts, it is in $O(n)$.
3. On average, Horspool's algorithm is faster than the brute-force algorithm.

Example:

(From Problem Session 4, Question 3)

Consider the problem of searching for genes in DNA sequences using Horspool's algorithm. A DNA sequence is represented by a text on the alphabet $\{A, C, G, T\}$, and the gene or a gene segment is a pattern.

1. Construct the shift table for the following gene segment.
   TCCTATTCTT
2. Apply Horspool' s algorithm to locate the pattern in the following DNA sequence.
   TTATAGATCTGGTATTCTTTTATAGATCTCCTATTCTT

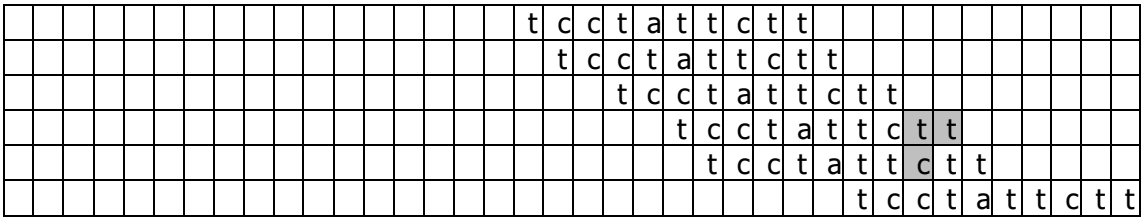
Solution:

1.

A	C	G	T
5	2	10	1

2.

t	t	a	t	a	g	a	t	c	t	g	g	t	a	t	t	c	t	t	t	t	a	t	a	g	a	t	c	t	c	c	t	a	t	t	c	t	t
t	c	c	t	a	t	t	c	t	t																													
	t	c	c	t	a	t	t	c	t	t																												
											t	c	c	t	a	t	t	c	t	t																	
												t	c	c	t	a	t	t	c	t	t																	



Single-source shortest-paths

Definition:

(See Week5, Single-source shortest-paths, Definition)
All-pairs shortest paths:

1. Nonnegative edge weights
   a) Dijkstra's algorithm $|V|$ times: $O(VE + V^2 \lg V)$
2. General
   a) Bellman-Ford $|V|$ times: $O(V^2E)$
   b) Dynamic Programming (one of the today's topics)

Bellman-Ford algorithm

Definition:

Finds all shortest-path lengths from a source $s \in V$ to all $v \in V$ or determines that a negative-weight cycle exists.

Pseudocode:

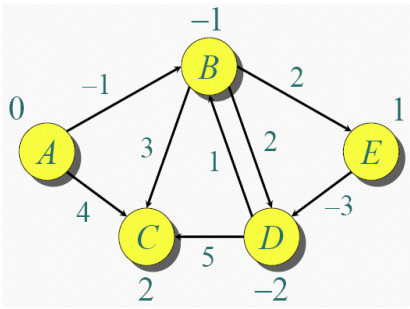
```
Algorithm Bellman-Ford(G=(V, E), s)
//input: a graph G=(V,E) with a source vertex s
//output: a array d[0..|V|-1], indexed with V, d[v] is the
//length of shortest path from s to v
   d[s] ← 0
   for each v ∈ V - {s}
      do d[v] ← ∞

   for i ← 1 to |V| - 1
      do for each edge (u, v) ∈ E
         do if d[v] > d[u] + w(u, v)
            then d[v] ← d[u] + w(u, v)

   for each edge (u, v) ∈ E
      do if d[v] > d[u] + w(u, v)
         then report that a negative-weight cycle exists
```

Time complexity is $O(VE)$.

Example:



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	1	∞
0	−1	2	−2	∞
0	−1	2	−2	1

Leave a question open.

All-pairs shortest paths

Definition:

Problem: A weighted (di)graph $G = (V, E)$, find a $|V| \times |V|$ matrix, it's entry d_{ij} is the shortest-path length between vertices i, j, where $i, j \in V$.

Solution:

Naïve solution:

1. Run Bellman-Ford algorithm once for each vertex.
2. Time $= O(V^2E)$.
3. Dense graph $\Longrightarrow O(V^4)$ time, e.g., for a complete graph, $|E| = |V||(V-1)|/2$

Floyd's Algorithm

Definition:

1. Problem: In a weighted (di)graph, find shortest paths between every pair of vertices.
2. Weight Matrix: the entry w_{ij} is the weight on edge (v_i, v_j).
3. Distance Matrix: the entry d_{ij} is the distance between v_i and v_j.
4. Idea: construct solution through series of matrices $D^{(0)}, \ldots, D^{(n)}$ using increasing subsets of the vertices allowed as intermediate.

Algorithm:

On the k-th iteration, the algorithm determines shortest paths between every pair of vertices i, j that use only vertices among $1, \ldots, k$ as intermediate.

$D^{(k)}[i,j] = min\{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$

Pseudocode:

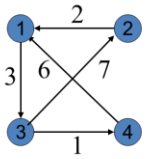
```
Algorithm Floyd(W[1..n, 1..n])
    //Implements Floyd's algorithm for the all-pairs shortest-
    //paths problem
    //Input: The weight matrix W of a graph with no negative-
```

```
//length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i,j] ← min{D[i,j], D[i,k]+ D[k,j]}
return D
```

Complexity:

Time efficiency: $O(n^3)$.
Space efficiency: Matrices can be written over their predecessors.
Note: Shortest paths themselves can be found, too.

Example:



$D^{(0)} =$

0	∞	3	∞
2	0	∞	∞
∞	7	0	1
6	∞	∞	0

$D^{(1)} =$

0	∞	3	∞
2	0	**5**	∞
∞	7	0	1
6	∞	**9**	0

$D^{(2)} =$

0	∞	3	∞
2	0	5	∞
9	7	0	1
6	∞	9	0

$D^{(3)} =$

0	**10**	3	**4**
2	0	5	**6**
9	7	0	1
6	**16**	9	0

$D^{(4)} =$

0	10	3	4
2	0	5	6
7	7	0	1
6	16	9	0

Warshall's Algorithm

Definition:

1. Computes the transitive closure of a relation
2. Alternatively: existence of all nontrivial paths in a digraph

Algorithm:

1. Constructs transitive closure T as the last matrix in the sequence of n-by-n matrices $R^{(0)}, \dots, R^{(k)}, \dots, R^{(n)}$ where $R^{(k)}[i,j] = 1$ iff there is nontrivial path from i to j with only first k vertices allowed as intermediate.
   Note that $R^{(0)} = A$ (adjacency matrix), $R^{(n)} = T$(transitive closure).
2. On the k-th iteration, the algorithm determines for every pair of vertices i, j if a path exists from i and j with just vertices $1, \dots, k$ allowed as intermediate.
   $$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] \\ R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j] \end{cases}$$
3. It implies the following rules for generating $R^{(k)}$ from $R^{(k-1)}$:
   a) If an element in row i and column j is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
   b) If an element in row i and column j is 0 in $R^{(k-1)}$, it has to be changed to 1 in

$R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$.

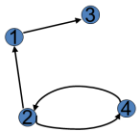
Pseudocode:

```
Algorithm Warshall(A[1..n, 1..n])
    //Implements Warshall's algorithm for computing the transitive
closure
    //Input: The adjacency matrix A of a digraph with n vertices
    //Output: The transitive closure of the digraph
    R(0) ← A
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← i to n do
                R(k)[i,j] ← R(k-2)[i,k] or (R(k-1)[i,k] and R(k-1)[k,j]
    return R(n)
```

Time efficiency: $O(n^3)$.

Space efficiency: Matrices can be written over their predecessors.

Example:



$R^{(0)} =$

0	0	1	0
1	0	0	1
0	0	0	0
0	1	0	0

$R^{(1)} =$

0	0	1	0
1	0	**1**	1
0	0	0	0
0	1	0	0

$R^{(2)} =$

0	0	1	0
1	0	1	1
0	0	0	0
1	**1**	**1**	**1**

$R^{(3)} =$

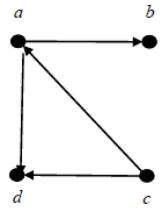
0	0	1	0
1	0	1	1
0	0	0	0
1	1	1	1

$R^{(4)} =$

0	0	1	0
1	**1**	1	1
0	0	0	0
1	1	1	1

(From Problem Session 4, Question 1)

Apply Warshall's algorithm to find the transitive closure of the following digraph.



Solution:

$$R0 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$R1 = R0 \vee \begin{pmatrix}0\\0\\1\\0\end{pmatrix}(0\ 1\ 0\ 1) = \begin{pmatrix}0\ 1\ 0\ 1\\0\ 0\ 0\ 0\\1\ 1\ 0\ 1\\0\ 0\ 0\ 0\end{pmatrix}$$

$$R2 = R1 \vee \begin{pmatrix}1\\0\\1\\0\end{pmatrix}(0\ 0\ 0\ 0) = \begin{pmatrix}0\ 1\ 0\ 1\\0\ 0\ 0\ 0\\1\ 1\ 0\ 1\\0\ 0\ 0\ 0\end{pmatrix} = R1$$

$$R3 = R2 \vee \begin{pmatrix}0\\0\\0\\0\end{pmatrix}(1\ 1\ 0\ 1) = \begin{pmatrix}0\ 1\ 0\ 1\\0\ 0\ 0\ 0\\1\ 1\ 0\ 1\\0\ 0\ 0\ 0\end{pmatrix} = R2$$

$$R4 = R3 \vee \begin{pmatrix}1\\0\\1\\0\end{pmatrix}(0\ 0\ 0\ 0) = \begin{pmatrix}0\ 1\ 0\ 1\\0\ 0\ 0\ 0\\1\ 1\ 0\ 1\\0\ 0\ 0\ 0\end{pmatrix} = R3$$

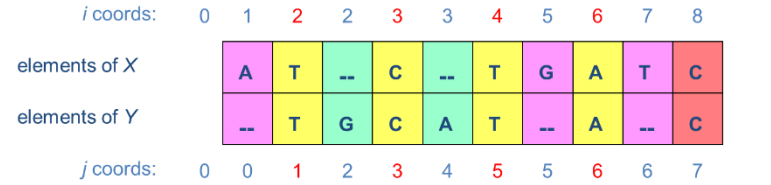
Week9

Longest Common Subsequence (LCS)

Terminology:

1. DNA analysis: Two DNA string comparison.
2. DNA string: A sequence of symbols A, C, G, T.
   $S = ACCGGTCGAGCTTCGAAT$
3. Subsequence (of X): X with some symbols left out.
   $Z = CGTC$ is a subsequence of $X = ACGCTAC$.
4. Common subsequence Z (of X and Y): a subsequence of X and also a subsequence of Y.
   $Z = CGA$ is a common subsequence of both $X = ACGCTAC$ and $Y = CTGACA$.
5. Longest Common Subsequence (LCS): the longest one of common subsequences.
   $Z' = CGCA$ is the LCS of the above X and Y.
6. LCS problem: Given $X =< x_1, x_2, \dots, x_m >$ and $Y =< y_1, y_2, \dots, y_n >$, find their LCS.

Example:



(0,0)→(1,0)→(2,1)→(2,2)→(3,3)→(3,4)→(4,5)→(5,5)→(6,6)→(7,6)→(8,7)

Matches shown in red

positions in X: 2 < 3 < 4 < 6 < 8

positions in Y: 1 < 3 < 5 < 6 < 7

Dynamic Programming (LCS)

Algorithm:

Step1: Optimal Substructure

Let $X =< x_1, x_2, \dots, x_m > (= X_m)$ and $Y =< y_1, y_2, \dots, y_n > (= Y_n)$
and $Z =< z_1, z_2, \dots, z_k > (= Z_k)$ be any LCS of X and Y

1. if $x_m = y_n$, then $z_k = x_m = y_n$, and Z_{k-1} is the LCS of X_{m-1} and Y_{n-1}.
2. if $x_m \neq y_n$, then $z_k \neq x_m$ implies Z is the LCS of X_{m-1} and Y_n.
3. if $x_m \neq y_n$, then $z_k \neq y_n$ implies Z is the LCS of X_m and Y_{n-1}.

Step 2: Recursive Solution

1. If $x_m = y_n$, find LCS of X_{m-1} and Y_{n-1}, then append x_m.
2. If $x_m \neq y_n$, find LCS of X_{m-1} and Y_n and LCS of X_m and Y_{n-1}, take which one is longer.
3. Both LCS of X_{m-1} and Y_n and LCS of X_m and Y_{n-1} will need to solve LCS of X_{m-1} and Y_{n-1}.

4. $c[i,j]$ is the length of LCS of X_i and Y_j.

$$c[i,j] = \begin{cases} 0 & \text{if i} = 0, \text{or j} = 0 \\ \text{c[i} - 1, \text{j} - 1] + 1 & \text{if i, j} > 0 \text{ and } \text{x}_\text{i} = \text{y}_\text{j} \\ \max\{\text{c[i} - 1, \text{j}], \text{c[i, j} - 1]\} & \text{if i, j} > 0 \text{ and } \text{x}_\text{i} \neq \text{y}_\text{j} \end{cases}$$

Step 3: Computing the Length of LCS

1. Matrix $c[0..m, 0..n]$, where $c[i,j]$ is defined as above. $c[m,n]$ is the answer (length of LCS).
2. $b[1..m, 1..n]$, where $b[i,j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i,j]$. From $b[m,n]$ backward to find the LCS.

Pseudocode:

```
LCS-LENGTH(X, Y)
    m ← length[X]
    n ← length[y]
    for i ← 1 to m
        do c[i,0] ← 0
    for j ← 0   to n
        do c[0,j] ← 0
    for i ← 1 to m do
        for j ← l to n
            do if xi = yj
                then c[i,j] ← c[i-1,j-1]+1
                    b[i,j] ← "↖"
                else if c[i-1,j] ⩾ c[i,j-1]
                    then c[i,j] ← c[i-1,j]
                        b[i,j] ← "↑"
                    else c[i,j] ← c[i,j-1]
                        b[i,j] ← "←"
    return c and b

PRINT-LCS(b, X, i, j)
    if i = 0 or j = 0
        then return
    if b[i,j] = "↖"
        then PRINT-LCS(b, X, i-1, j-1)
            print xi
    elseif b[i,j] = "↑"
        then PRINT-LCS(b, X, i-1, j)
    else PRINT-LCS(b, X, i, j-1)
```

Example:

$S1 = \text{“}aab\text{”}$

$S2 = \text{“}azb\text{”}$

	" "	a	a	b
" "	0	0	0	0
a	0	1 ↖	1 ↑	1 ←
z	0	1 ↑	1 ↑	1 ↑
b	0	1 ↑	1 ↑	2 ↖

The Length of LCS is 2.

Pairwise Sequence Alignment (PSA)

Definition:

1. Given a pair of sequences (DNA or protein) and a method for scoring a candidate alignment.
2. Find an alignment for which the score is maximized.

Dynamic Programming (Global Alignment)

Complexity:

1. $O(n^2)$ or $O(n \times m)$ algorithm (sequences of length n, m).
2. Feasible for moderate sized sequences, not entire genomes.

Algorithm:

$$F(i,j) = \max \begin{cases} F(i-1,j-1) + s(x_i, y_j) & c1 \\ F(i-1,j) + d & c2 \\ F(i,j-1) + d & c3 \end{cases}$$

x_i: i[th] letter of string x

y_j: j[th] letter of string y

$x_{1..i}$: prefix of x from letters 1 through i

F: matrix of optimal scores (DP Matrix)

$F(i,j)$ represents optimal score lining up $x_{1..i}$ with $y_{1..j}$

d: gap penalty

s: scoring matrix

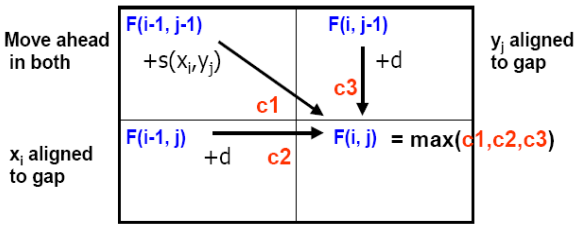
$c1$: match top & bottom

$c2$: insert bottom gap

$c3$: insert top gap

Algorithm:

1. Initialize: $F(0,0) = 0$, $F(i,0) = i \times d$, $F(0,j) = j \times d$ (gap penalties).
2. Fill from top left to bottom right using recursive formula.



While building the table, keep track of where optimal score came from, then reverse arrows

Example:

1. Seq1: HEAGAWGHEE
2. Seq2: PAWHEAE
3. Gap Penalty: -8 (Constant)
4. Scoring Matrix (Blosum 50)

	H	E	A	G	A	W	G	H	E	E
P	-2	-1	-1	-2	-1	-4	-2	-2	-1	-1
A	-2	-1	5	0	5	-3	0	-2	-1	-1
W	-3	-3	-3	-3	-3	15	-3	-3	-3	-3
H	10	0	-2	-2	-2	-3	-2	10	0	0
E	0	6	-1	-3	-1	-3	-3	0	6	6
A	-2	-1	5	0	5	-3	0	-2	-1	-1
E	0	6	-1	-3	-1	-3	-3	0	6	6

		H	E	A	G	A	W	G	H	E	E
	0	-8	-16	-24	-32	-40	-48	-56	-64	-72	-80
P	-8	-2	-9	-17	-25	-33	-42	-49	-57	-65	-73
A	-16	-10	-3	-4	-12	-20	-28	-36	-44	-52	-60
W	-24	-18	-11	-6	-7	-15	-5	-13	-21	-29	-37
H	-32	-14	-18	-13	-8	-9	-13	-7	-3	-11	-19
E	-40	-22	-8	-16	-16	-9	-12	-15	-7	3	-5
A	-48	-30	-16	-3	-11	-11	-12	-12	-15	-5	2
E	-56	-38	-24	-11	-6	-12	-14	-15	-12	-9	1

Trace arrows from bottom right to top left

1. Diagonal: Both match
2. Up: Left sequence matches a gap or insert a gap to top sequence
3. Left: Top sequence matches a gap or insert a gap to left sequence

Optimal global alignment:

H E A G A W A H E _ E

_ _ P _ A W _ H E A E

(From Problem Session 4, Question 4.1)

Using a gap penalty of $d = -5$ and scoring matrix as below.

	A	C	G	T
A	2	-7	-5	-7
C	-7	2	-7	-5
G	-5	-7	2	-7
T	-7	-5	-7	2

And applying dynamic programming to find the optimal global alignment of AATG and AGC.

Solution:

a) Complete the following table using the formula:

$$F(i,j) = \max \begin{cases} F(i-1,j-1) + s(x_i, y_j) & c1 \\ F(i-1,j) + d & c2 \\ F(i,j-1) + d & c3 \end{cases}$$

and set $F(0,0) = 0$, $F(i,0) = i \times d$, $F(0,j) = j \times d$ (gap penalties)

		A	A	T	G
	0	-5	-10	-15	-20
A	-5	2	-3	-8	-13
G	-10	-3	-3	-8	-6
C	-15	-8	-8	-8	-11

b) Trace arrows from bottom right to top left

1. Diagonal: Both match
2. Up: Left sequence matches a gap or insert a gap to top sequence
3. Left: Top sequence matches a gap or insert a gap to left sequence

There are two best global alignments of AATG and AGC.

A A T G _

_ A _ G C

and

A A T G _

A _ _ G C

Dynamic Programming (Local Alignment)

Global alignment vs. Local alignment

Global alignment:

1. The entire sequence of each protein or DNA sequence is contained in the alignment.
2. Global methods are useful when you want to force two sequences to align over their entire length.

Local alignment:

1. Only regions of greatest similarity between two sequences are aligned.
2. Local alignment is almost always used for database searches such as BLAST. It is useful to find domains (or limited regions of homology) within sequences.

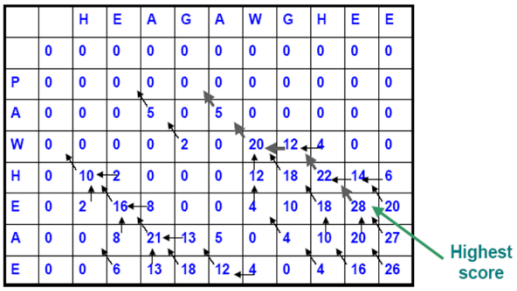
Algorithm:

1. Make 0 minimal score (i.e., start new alignment)
2. Alignment can start / end anywhere (Start at highest score(s) and End when 0 reached)

3. $F(i,j) = \max \begin{cases} F(i-1,j-1) + s(x_i, y_j) & c1 \\ F(i-1,j) + d & c2 \\ F(i,j-1) + d & c3 \\ 0 \end{cases}$

Example:

(See Week9, Dynamic Programming (Global Alignment), Example)



Traceback: Start at highest score and trace arrows back to first 0.
Optimal local alignment:
A W G H E
A W _ H E

(From Problem Session 4, Question 4.1)
(See Week9, Dynamic Programming (Global Alignment), Example)
Solution:

a) Complete the following table using the formula:

$$F(i,j) = \max \begin{cases} F(i-1,j-1) + s(x_i, y_j) & c1 \\ F(i-1,j) + d & c2 \\ F(i,j-1) + d & c3 \\ 0 \end{cases}$$

and set $F(0,0) = 0$.

		A	A	T	G
	0	0	0	0	0
A	0	2	2	0	0
G	0	0	0	0	2
C	0	0	0	0	0

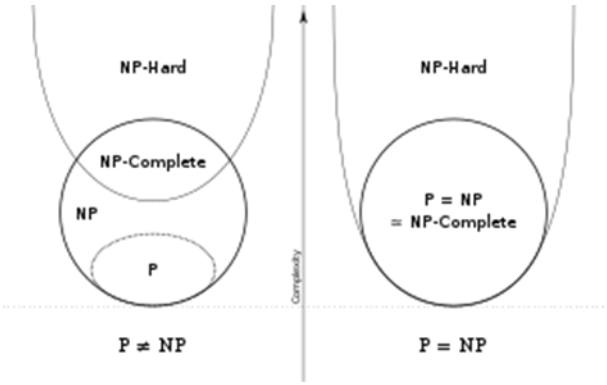
b) Traceback: Start at highest score and trace arrows back to first 0.
Three best local alignments are found:
A A G
A A G

Week10

Hard Computational Problems

Definition:

1. An algorithm is efficient if its running time is bounded by a polynomial of its input size. Some computational problems seem hard to solve.
2. Despite numerous attempts we do not know any efficient algorithms for these problems. We are also far away from proving these problems are indeed hard to solve
3. In more formal language, **we don't know whether $NP = P$ or $NP \neq P$**. This is an important and fundamental question in theoretical computer science!



Example:

Hamiltonian Circuit:

Input:

A connected graph G

Question:

Does G have a Hamiltonian circuit? i.e., does G have a circuit that passes through every vertex exactly once, except for the starting and ending vertex?

Knapsack Problem:

Input:

Given n items with integer weights $w_1, w_2, \dots, w_n$ and integer values $v_1, v_2, \dots, v_n$, and a knapsack with capacity W.

Problem (optimization version):

Find a subset of items whose total weight does not exceed W and that maximises the total value. (taking fractional parts of items is NOT allowed) Known as 0/1 Knapsack Problem.

Boolean Circuit:

A Boolean Circuit is a directed graph where each vertex, called a logic gate corresponds to a simple Boolean function, one of AND, OR, or NOT.

Incoming edges:

inputs for its Boolean function

Outgoing edges:

outputs

Circuit-SAT:

Input:

A Boolean Circuit with a single output vertex

Question:

Is there an assignment of values to the inputs so that the output value is 1?

Decision/Optimisation problems

Definition:

1. A decision problem is a computational problem for which the output is either yes or no.
2. In an optimisation problem, we try to maximise or minimise some value.
3. An optimisation problem can be turned into a decision problem if we add a parameter k; and then ask whether the optimal value in the optimisation problem is at most or at least k.
4. Note that if a decision problem is **hard**, then its related optimisation version must also be **hard**.

Example:

MST:

Optimisation problem:

Given a graph G with integer weights on its edges. What is the weight of a minimum spanning tree (MST) in G?

Decision problem:

Given a graph G with integer weights on its edges, and an integer k. Does G have a MST of weight at most k?

Knapsack problem:

Input:

Given n items with integer weights $w_1, w_2, \dots, w_n$ and integer values $v_1, v_2, \dots, v_n$, a knapsack with capacity W and a value k.

Optimisation problem:

Find a subset of items whose total weight does not exceed W and that maximises the total value.

Decision problem:

For any integer k, is there a subset of items whose total weight does not exceed W and whose total value is at least k?

Decidable:

1. Not all decision problems can be solved by algorithms.
2. The problems cannot be solved by algorithms is called undecidable problems.

Solving/Verifying:

1. Solving: We are given an input, and then we have to FIND the solution.

2. Verifying: In addition to the input, we are given a "certificate" and we verify whether the certificate is indeed a solution
3. We may not know how to solve a problem efficiently, but we may know how to verify whether a candidate is actually a solution.

Example:

Hamiltonian circuit problem:

1. Suppose through some (unspecified) means (like good guessing), we find a candidate for a Hamiltonian circuit, i.e., a list of vertices and edges that might be a Hamiltonian circuit in the input graph G.
2. It is easy to check if this is indeed a Hamiltonian circuit. Check
   a) that all the proposed edges exist in G.
   b) that we indeed have a cycle.
   c) that we hit every vertex in G once.
3. If the candidate solution is indeed a Hamiltonian circuit, then it is a certificate verifying that the answer to the decision problem is "Yes".

0/1 Knapsack Problem:

1. Consider an instance of the 0/1 Knapsack problem (decision version).
2. Suppose someone proposes a subset of items, it is easy to check.
   a) if those items have total weight at most W.
   b) if the total value is at least k.
3. If both conditions are true, then the subset of items is a certificate for the decision problem. (It verifies that the answer to the 0/1 knapsack decision problem is "Yes".)

Circuit-SAT:

1. Consider a Boolean Circuit.
2. Suppose someone proposes an assignment of truth values to the input, it is easy to check.
   a) if the input values lead to a final value of 1 in the output.
   b) this is done by checking every logic gate.
3. If the input truth values give a final value of 1, these values form a certificate for the decision problem.
4.

P and NP

Definition:

1. The complexity class P is the set of all decision problems that can be solved in worst-case polynomial time.
2. The complexity class NP is the set of all problems that can be verified in polynomial time.
3. P stands for polynomial, and NP stands for non-deterministic polynomial.

Example:

1. MST problem is in P
2. Single-source-shortest-paths problem is in P

3. Hamiltonian circuit problem is in NP
4. 0/1 Knapsack problem is in NP
5. Circuit-SAT is in NP

P = NP?:

1. Note that $P \subseteq NP$.
2. The (million dollar) question is that mathematicians and computer scientists do not know whether $P = NP$ or $P \neq NP$.
3. However, there is a common belief that P is different from NP
4.

Polynomial-time reduction

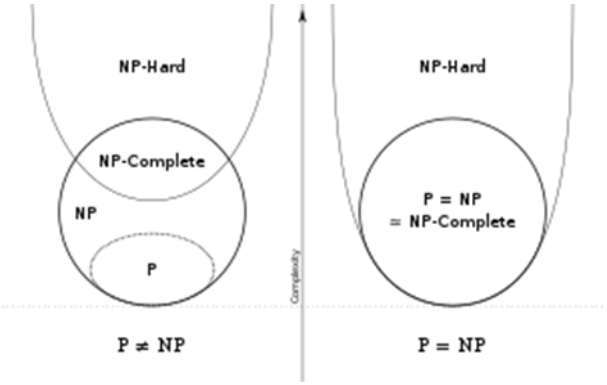
Definition:

1. Given any two decision problems A and B, we say that
   a) A is polynomial time reducible to B, or
   b) there is a polynomial time reduction from A to B
2. if given any input α of A, we can construct in polynomial time an input β of B such that α is yes if and only if β is yes.
3. We use the notation $A \leq_P B$.

NP-hardness / NP-completeness

Definition:

1. A problem M is said to be NP-hard if every other problem in NP is polynomial time reducible to M (Intuitively, this means that M is at least as difficult as all problems in NP).
2. M is further said to be NP-complete if M is in NP, and M is NP-hard.



NP-Complete Problem

Definition:

1. The Cook-Levin Theorem states that Circuit-SAT is NP-complete (a "first" NP-

complete problem)

2. Using polynomial time reducibility we can show existence of other NP-complete problems
3. A useful result to prove NP-completeness: If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.

Conjunctive normal form (CNF)

Definition:

a Boolean formula is in CNF if it is formed as a collection of clauses combined using the operator AND (•) and each clause is formed by literals (variables or their negations) combined using the operator OR (+).

CNF-SAT and 3-SAT

Definition:

CNF-SAT:

Input:

a Boolean formula in CNF

Question:

Is there an assignment of Boolean values to its variables so that the formula evaluates to true? (i.e., the formula is satisfiable)

3-SAT:

Input:

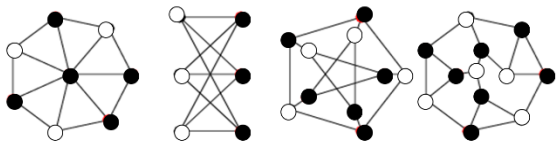
a Boolean formula in CNF in which each clause has exactly 3 literals

CNF-SAT and 3-SAT are NP-complete.

Vertex Cover

Definition:

Given a graph $G = (V, E)$. A vertex cover is a subset $C \subseteq V$ such that for every edge (v, w) in E, $v \in C$ or $w \in C$.



The optimisation problem is to find as **small** a vertex cover as possible.

Vertex Cover is the **decision** problem that takes a graph G and an integer k and asks whether there is a vertex cover for G containing at most k vertices.

Vertex Cover is NP-complete.

Week11

Exact Solution Strategies

List:

1. exhaustive search (brute force)
   a) useful only for small instances
2. backtracking
   a) eliminates some unnecessary cases from consideration
   b) yields solutions in reasonable time for many instances but worst case is still exponential
3. branch-and-bound
   a) further refines the backtracking idea for optimization problems
4. dynamic programming
   a) applicable to some problems (e.g., the knapsack problem)

Algorithm Design Techniques

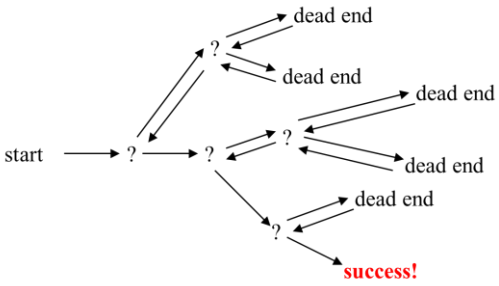
List:

1. Greedy
   a) Shortest path, minimum spanning tree, …
2. Divide and Conquer
   a) Divide the problem into smaller subproblems, solve them, and combine into the overall solution
   b) Often done recursively
   c) Quick sort, merge sort are great examples
3. Dynamic Programming
   a) Brute force through all possible solutions, storing solutions to subproblems to avoid repeat computation
4. Backtracking
   a) A clever form of exhaustive search

Backtracking

Definition:

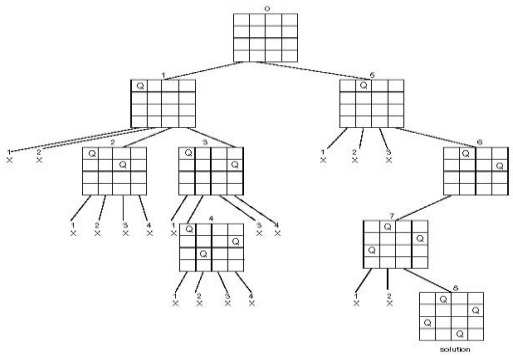
1. Construct the **state-space** tree
   a) nodes: partial solutions
   b) edges: choices in extending partial solutions
2. Explore the state space tree using depth-first search
3. "Prune" (**修剪**) nonpromising nodes
   a) DFS stops exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node's parent to continue the search.



Example:

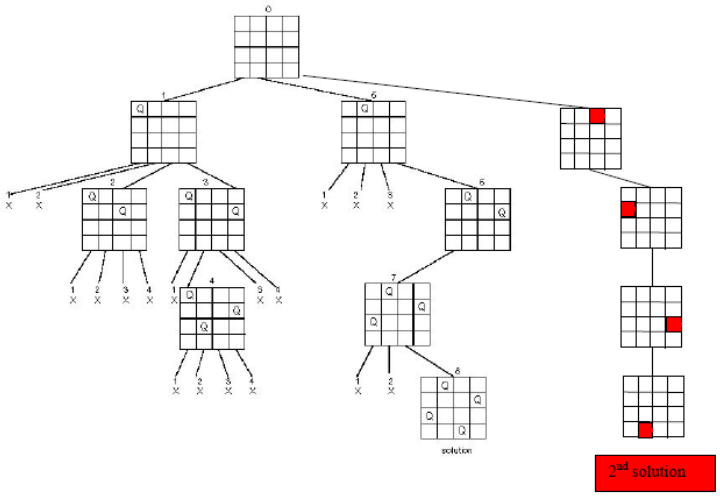
n-Queens Problem:

Place n queens on an n-by-n chess board so that no two of them are in the same row, column, or diagonal.



(From Problem Session 5, Question 1)

Continue the backtracking search for a solution to the four-queens problem, which was given in this week's lecture, to find the second solution to the problem. Explain how the board's symmetry can be used to find the second solution to the four-queens problem.

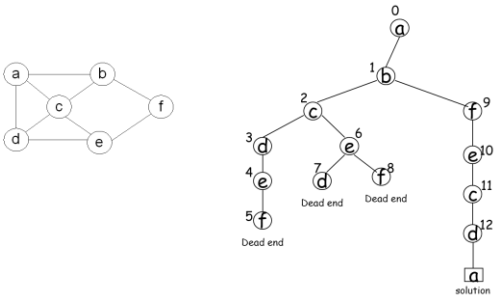


Solution:

The second solution can be obtained by following way:

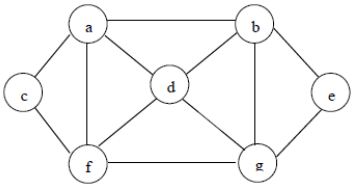
If there is a queen at position (i, j), using board's symmetry then in the second solution, it will have a queen at position $(i, 4 - j + 1)$ $(i = 1,2,3,4;\ j = 1,2,3,4)$.

Hamiltonian Circuit Problem:

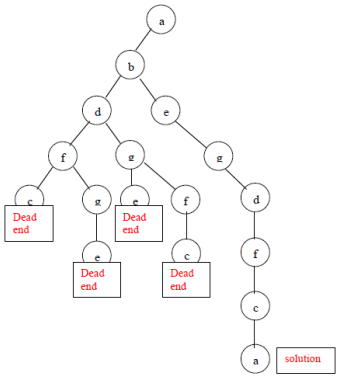


(From Problem Session 5, Question 2)

Apply backtracking to the problem of finding a Hamiltonian circuit in the following graph (starting from vertex a)



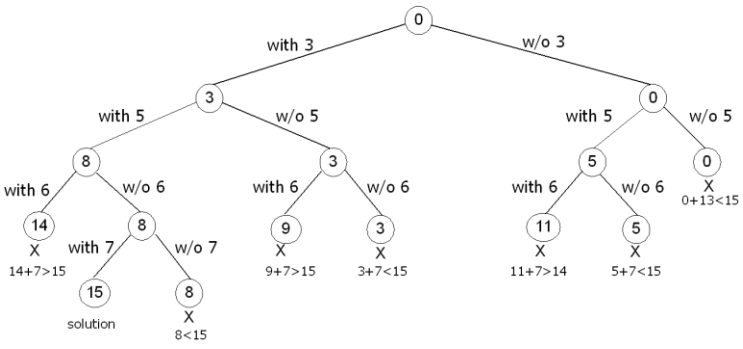
Solution:



Subset-Sum Problem:

Find a subset of a given set $S = \{s_1, \dots, s_n\}$ of n positive integers whose sum is equal to a give positive integer d.

For example: $S = \{3, 5, 6, 7\}$, $d = 15$



Comment:

1. Typically, backtrack is applied to difficult combinatorial problems for which no efficient algorithms for finding exact solutions possibly exist.
2. Unlike exhaustive search approach, which is doomed to be extremely slow for all

instances of a problem, backtrack at least holds a hope for solving some instances of nontrivial size in an acceptable amount of time.

3. Even if backtracking does not eliminate any elements of a problem's state space and ends up generating all its elements, it provides a specific technique for doing so, which can be of value in its own right.

Branch-and-Bound

Definition:

1. An enhancement of backtracking
2. Applicable to optimization problems
3. For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution)
4. Uses the bound for:
   a) ruling out certain nodes as "nonpromising" to prune the tree – if a node's bound is not better than the best solution seen so far
   b) guiding the search through state-space

Example:

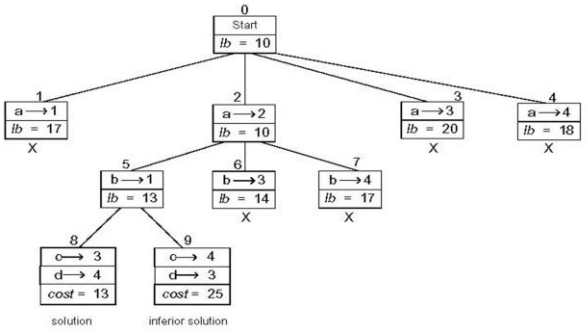
Assignment Problem:

Select one element in each row of the cost matrix C so that:

1. no two selected elements are in the same column
2. the sum is minimized

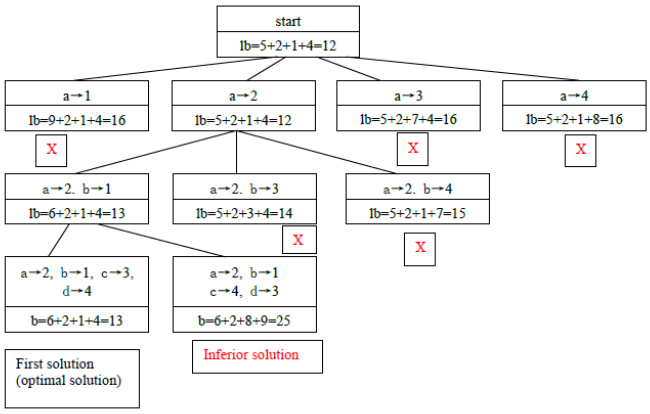
	Job 1	Job 2	Job 3	Job 4
Person *a*	9	2	7	8
Person *b*	6	4	3	7
Person *c*	5	8	1	8
Person *d*	7	6	9	4

Lower bound: Any solution to this problem will have total cost at least: $2 + 3 + 1 + 4$ (taking smallest value from row) (or $5 + 2 + 1 + 4$, taking smallest value from column)



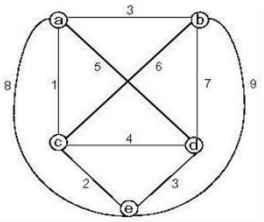
(From Problem Session 5, Question 3)

Solution:



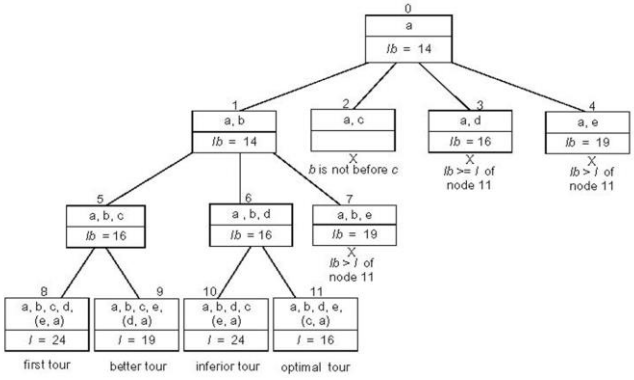
Traveling Salesman Problem:

1. Given a graph, the TSP Optimization problem is to find a tour, starting from any vertex, visiting every other vertex and returning to the starting vertex, with minimal cost.
2. It is NP-complete
3. We try to avoid n! exhaustive search by the branch-and-bound technique.



	a	b	c	d	e
a	0	3	1	5	8
b	3	0	6	7	9
c	1	6	0	4	2
d	5	7	4	0	3
e	8	9	2	3	0

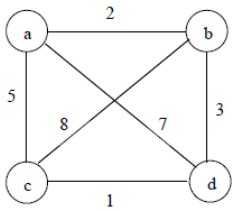
A reasonable lower bound is $lb = \Sigma(distances\ of\ two\ nearest\ cities\ of\ v\ in\ G)/2$.



By requiring b before C, we can reduce space by half.

(From Problem Session 5, Question 4)

Apply the branch-and-bound algorithm to solve the travelling salesman problem for the following graph.



Solution:

To reduce the complexity, we can assume that, b is before c in the tour. The lower bound for each node can be computed by

Node 0: $lb = \lceil [(2+5)+(2+3)+(1+5)+(1+3)]/2 \rceil = 11$

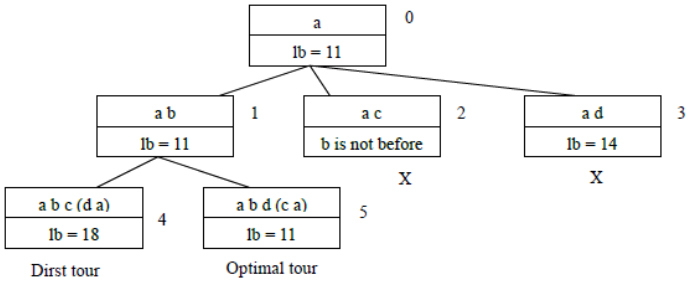
Node 1: $lb = \lceil [(2+5)+(2+3)+(1+5)+(1+3)]/2 \rceil = 11$

Node 2: ignored as b is not before c.

Node 3: $lb = \lceil [(2+7)+(2+3)+(1+5)+(1+7)]/2 \rceil = 14$

Node 4: leads to a tour with $lb = 18$

Node 5: leads to an optimal tour with $lb = 11$.



Solution: $a\ b\ d\ c\ a$.

Week12

Dynamic Programming

List:

1. 1-dimensional DP
2. 2-dimensional DP
3. Interval DP
4. Tree DP
5. Subset DP

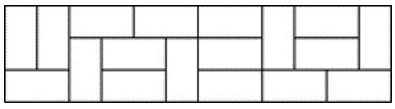
Steps:

1. Defining subproblems
2. Finding recurrences
3. Solving the base cases
4.

POJ 2663: Tri Tiling (1-dimensional DP)

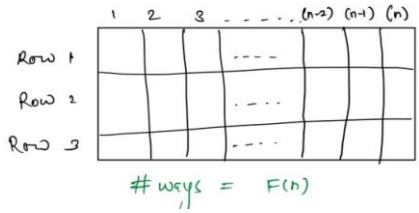
Definition:

Given n, find the number of ways to fill a $3 \times n$ board with dominoes.
Here is one possible solution for $n = 12$.

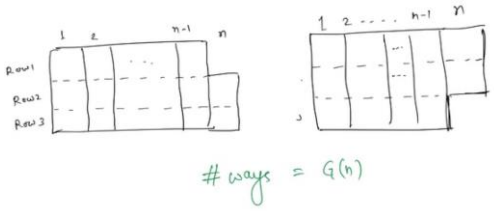


Algorithm:

1. $F(n)$ denotes the number of ways you can fill the $3 \times n$ rectangle using $2 \times n$ dominoes.

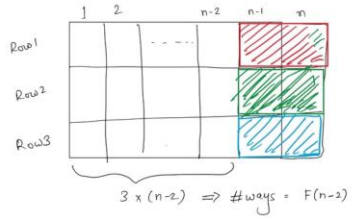


2. $G(n)$ denotes the number of ways you can fill $3 \times n$ rectangle with one of corner removed.

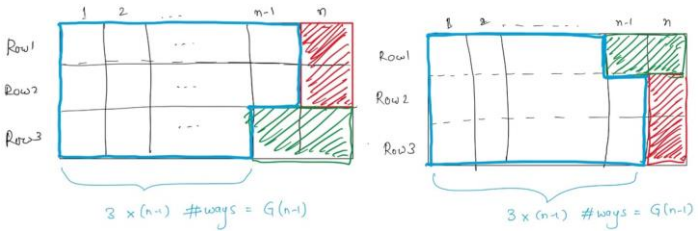


3. Total number of ways of filling up $3 \times (n-2)$ rectangle with 2×1 dominoes would

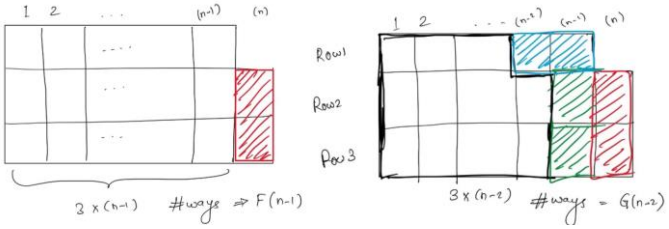
be $F(n-2)$.



4. So now just add all the ways in each case to get $F(n)$ as $F(n) = F(n-2) + 2 \times G(n-1)$.



5. So now just add all the ways in each case to get $G(n)$ as $G(n) = F(n-1) + G(n-2)$.



6. $\begin{cases} G(n) = F(n-1) + G(n-2) \\ F(n) = F(n-2) + 2 \times G(n-1) \end{cases}$

LCS Problem (2-dimensional DP)

Definition:

(See Week9, Dynamic Programming (LCS), ALL)

Palindrome (Interval DP)

Definition:

Problem:

given a string $x = x_{1\ldots n}$, find the minimum number of characters that need to be inserted to make it a palindrome.

Example:

1. x: Ab3bd
2. – Can get "dAb3bAd" or "Adb3bdA" by inserting 2 characters (one 'd', one 'A')

Algorithm:

1. Define subproblems
   a) Let Di j be the minimum number of characters that need to be inserted to make

$x_{i\ldots j}$ into a palindrome.

2. Find the recurrence
   a) Consider a shortest palindrome $y_{1\ldots k}$ containing $x_{i\ldots j}$
   b) Either $y_1 = x_i$ or $y_k = x_j$ (why?)
   c) $y_{2\ldots k-1}$ is then an optimal solution for $x_{i+1\ldots j}$ or $x_{i\ldots j-1}$ or $x_{i+1\ldots j-1}$
   d) Last case possible only if $y_1 = y_k = x_i = x_j$
3. $D_{ij} = \begin{cases} 1 + \min\{D_{i+1,j}, D_{i,j-1}\} & x_i \neq x_j \\ D_{i+1,j-1} & x_i = x_j \end{cases}$
4. Find and solve the base cases: $D_{ii} = D_{i,i-1} = 0$ for all i.

Color node (Tree DP)

Definition:

Problem:

given a tree, color nodes black as many as possible without coloring two adjacent nodes.

Algorithm:

1. Subproblems:
   a) First, we arbitrarily decide the root node r
   b) B_v: the optimal solution for a subtree having v as the root, where we color v black.
   c) W_v: the optimal solution for a subtree having v as the root, where we don't color v
   d) Answer is $max\{B_r, W_r\}$
2. Find the recurrence
   a) Crucial observation: once v's color is determined, subtrees can be solved independently.
   b) If v is colored, its children must not be colored. $B_v = 1 + \sum W_u \ \ u \in children(v)$
   c) If v is not colored, its children can have any color. $W_v = \sum max\{B_u, W_u\} \ u \in children(v)$
3. Base cases: leaf nodes.

Knapsack 0-1 Problem

Definition:

1. The goal is to maximize the value of a knapsack that can hold at most W units (i.e., lbs or kg) worth of goods from a list of items $I_0, I_1, \ldots I_{n-1}$.
2. Each item has 2 attributes:
   a) Value – let this be v_i for item I_i
   b) Weight – let this be w_i for item I_i
3. The difference between this problem and the fractional knapsack one is that you CANNOT take a fraction of an item.

Algorithm:

1. The best subset of S_{k-1} that has total weight w
2. Or, the best subset of S_{k-1} that has total weight $w - w_k$ plus the item k.
3. $B[k,w] = \begin{cases} B[k-1,w] & if\ w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k] + b_k\} & else \end{cases}$

Pseudocode:

```
for w = 0 to W { // Initialize 1st row to 0's
    B[0,w] = 0
}
for i = 1 to n { // Initialize 1st column to 0's
    B[i,0] = 0
}
for i = 1 to n {
    for w = 0 to W {
        if wi <= w { //item i can be in the solution
            if vi + B[i-1,w-wi] > B[i-1,w]
                B[i,w] = vi + B[i-1,w-wi]
            else
                B[i,w] = B[i-1,w]
        }
        else B[i,w] = B[i-1,w] // wi > w
    }
}
```

Time complexity is $O(n * W)$.

Finding the Items:

1. This algorithm only finds the max possible value that can be carried in the knapsack (The value in $B[n,W]$)
2. To know the items that make this maximum value, we need to trace back through the table.

```
Let i = n and k = W
if B[i, k] ≠ B[i-1, k] then
    mark the ith item as in the knapsack
    i = i-1, k = k-wi
else
    i = i-1 // Assume the ith item is not in the knapsack
            // Could it be in the optimally packed knapsack?
```

Example:

1. $n = 4$ (# of elements)
2. $W = 5$ (max weight)
3. Elements (weight, value): $(2,3), (3,4), (4,5), (5,6)$

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

The max possible value that can be carried in this knapsack is $7.

The optimal knapsack should contain: Item 1 and Item 2.

(From Problem Session 6, Question 1)

Given the following instance of the 0/1 Knapsack problem:

item	weight	value
1	2	$12
2	1	$10
3	3	$20

The Knapsack Capacity $W = 3$.

Let $V[i,j]$ be the value of the most valuable subset of the first i items that fit into the Knapsack of capacity j. Then $V[i,j]$ can be recursively defined as following:

$$V[i,j] = \begin{cases} 0 & if\ i = 0\ or\ j = 0 \\ \max\{B[k-1,w], B[k-1,w-w_k] + b_k\} & if\ j - w_i \geq 0 \\ V[i-1,j] & if\ j - w_i < 0 \end{cases}$$

1. Using dynamic programming, complete the following table.

capacity j

Item i	0	1	2	3
0				
w_1=2, v_1=12 1				
w_2=1, v_2=10 2				
w_3=3, v_3=20 3				

2. What is the value of the most valuable subset?
3. Give an optimal subset of the instance based on the table.
4. Based on the recurrence relation given above, write a pseudocode of the bottom-up dynamic programming algorithm for the knapsack problem.
5. What is the value of the most valuable subset if the capacity of the knapsack is 2?

Solution:

1.

Item i	0	1	2	3
0	0	0	0	0
w_1=2, v_1=12 1	0	0	12	12
w_2=1, v_2=10 2	0	10	12	22
w_3=3, v_3=20 3	0	10	12	22

2. 22
3. {Item1, item2}
4.

```
Pseudocode for 0/1 Knapsack algorithm
```

```
For j = 0 to W
   V[0, j] = 0
For i = 1 to n
   V[i, 0] = 0
For i = 1 to n
   For j = 0 to W
      if wi <= j // item i can be part of the solution
         if bi + V[i-1, j-wi] > V[i-1, j]
            V[i, j] = bi + V[i-1, j- wi]
         else
            V[i, j] = V[i-1, j]
      else // wi > j
         V[i, j] = V[i-1,j]
```

5. 12

Week13

Coping With NP-Hardness

List:

1. Brute-force algorithms.
   a) Develop clever enumeration strategies.
   b) Guaranteed to find optimal solution.
   c) No guarantees on running time.
2. Heuristics.
   a) Develop intuitive algorithms.
   b) Guaranteed to run in polynomial time.
   c) No guarantees on quality of solution.
3. Approximation algorithms.
   a) Guaranteed to run in polynomial time.
   b) Guaranteed to find "high quality" solution, say within 1% of optimum.
   c) Obstacle: need to prove a solution's value is close to optimum, without even knowing what optimum value is!

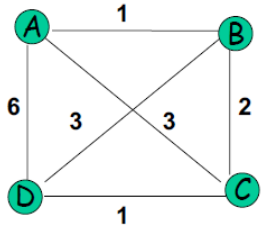
Heuristics

Definition:

1. A heuristic is a common-sense rule drawn from experience
   a) not a mathematically proven assertion
   b) a "rule-of-thumb"
2. Examples:
   a) TSP: go to next nearest city
   b) Knapsack: start with highest value/weight ratio

Nearest-Neighbor Algorithm for TSP:

Starting at some city, always go to the nearest unvisited city, and, after visiting all the cities, return to the starting one.



s_a: A – B – C – D – A of length 10
s^*: A – B – D – C – A of length 8
Note: Nearest-neighbor tour may depend on the starting city
Accuracy: $R_A = \infty$ (unbounded above) – make the length of AD arbitrarily large in the above example.

Approximation Algorithms

Definition:

Find a "good" solution fast

1. sufficient for many applications
2. we often have inaccurate data to start with, so approximation may be as good as optimal solution

Accuracy Ratio:

1. minimization problems: $r(s_a) = f(s_a)/f(s^*)$
   a) $f(s_a)$ = value of objective function for solution given by approximation algorithm
   b) $f(s^*)$ = value of objective function for optimal solution
2. maximization problems: $r(s_a) = f(s^*)/f(s_a)$
3. in either case $r(s_a) \geq 1$.

Performance Ratio:

1. If there exists $c \geq 1$, such that $r(s_a) \leq c$ for all instances of a problem, the given algorithm is called a **c-approximation algorithm.**
2. The smallest value of c that holds for all instances is called the performance ratio, R_A, of the algorithm,

Drawback:

1. c-approximation algorithms are good if you can find one.
   a) if $c = 1.1$, your approximation is never more than 10% worse than optimal.
2. but in some cases, no bound for c can be found.
   a) approx. alg. may be great for 99% of instances, but there are a few really terrible cases
3. For instance, THEOREM: if $P \neq NP$, no c-approximation algorithm for TSP exists.
4. it's unlikely that we can find a poly-time approx. algorithm for TSP such that $f(s_a) \leq cf(s^*)$ for all instances.

Vertex cover

Definition:

Instance:

an undirected graph $G = (V, E)$.

Problem:

find a set $C \subseteq V$ of minimal size s.t. for any $(u, v) \in E$, either $u \in C$ or $v \in C$.

Observation:

Let $G = (V, E)$ be an undirected graph. The complement V\C of a vertex-cover C is an independent-set of G.

Proof:

Two vertices outside a vertex-cover cannot be connected by an edge.

Pseudocode:

```
C ← ∅
E′ ← E
while E′ ≠ ∅
    do let (u,v) be an arbitrary edge of E′
        C ← C ∪ {u,v}
        remove from E′ every edge
    incident to either u or v
return C
```

Time complexity is $O(n^2)$.

Correctness:

$OPT \leq Our\ solution \leq 2 * CV$

Worst-case is $2 * OPT$, so it called 2-Approximation algorithm.

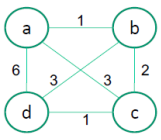
TSP

Euclidean Instances:

1. Euclidean instances of the TSP problem obey the natural geometry of a 2D map.
   a) triangle inequality: $d[i,j] \leq d[i,k] + d[k,j]$
   b) symmetry: $d[i,j] = d[j]$
2. For Euclidean instances, the nearest neighbor algorithm satisfies: $r(s_a) \leq \frac{1}{2}(\log_2 n + 1), where\ n = \#cities$ (still not a c-approximation algorithm)

Multifragment-heuristic:

1. sort edges in increasing order
2. Repeat until tour of length n: Add next smallest edge, if it doesn't create a vertex of degree 3 and doesn't create a cycle of length < n
3. More expensive than nearest-neighbor, same accuracy ratio



edge list: ***(a,b),*** (c,d), (b,c), (a,c), (b,d), (a,d)
tour:

edge list: ***(c,d)***, (b,c), (a,c), (b,d), (a,d)
tour: (a,b)

edge list: ***(b,c)***, (a,c), (b,d), (a,d)
tour: (a,b), (c,d)

edge list: ~~(a,c), (b,d),~~ ***(a,d)***
tour: (a,b), (c,d), (b,c)

tour: (a,b), (c,d), (b,c), (a,d) length 10

Twice Around the Tree:

Stage 1:

Construct a minimum spanning tree of the graph (e.g., by Prim's or Kruskal's algorithm)

Stage 2:

Starting at an arbitrary vertex, create a path that goes twice around the tree and

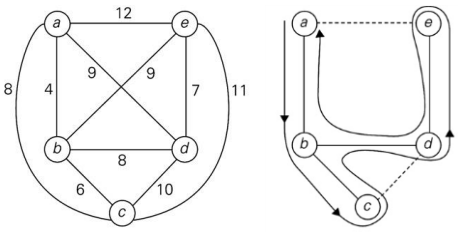
returns to the same vertex

Stage 3:

Create a tour from the circuit constructed in Stage 2 by making shortcuts to avoid visiting intermediate vertices more than once

Note:

$R_A = \infty$ for general instances, but this algorithm tends to produce better tours than the nearest-neighbor algorithm



DFS walk: abcbdedba

eliminate repeated nodes: abcdea

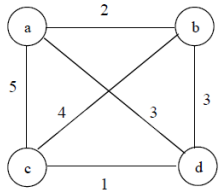
$OPT \leq MST\ Tour \leq 2 * MST$

Worst-case is $2 * OPT$

Example:

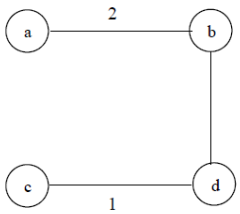
(From Problem Session 6, Question 3)

1. Is the following graph a Euclidean graph?
2. Apply Twice-Around-the-Tree algorithm to solve the travelling salesman problem for the following graph.
3. What is the optimization solution?
4. What is the accuracy ratio?



Solution:

1. No, as $d(a,d) + d(d,c) < d(a,c)$.
2. A minimum spanning tree of the graph is depicted below. Starting from a, by DFS, we have the following travel: $a\ b\ d\ c\ d\ b\ a$. By removing the repeat nodes except a, we have the following approximation solution: $S_A = a\ b\ d\ c\ a$, with length 11.



3. For this instance, the optimization solution is $S^* = a\ b\ c\ d\ a$ with length 10.
4. The $r(s_a) = f(s_a)/f(s^*) = 11/10 = 1.1$.