Design and Analysis of Algorithms

Rahul Jain

Lecturer : RAHUL JAIN

Office : S15-04-01

Email: rahul@comp.nus.edu.sg

Phone: 65168826 (off)

Grading :

50 marks for final exam
35 marks for midterm exam
10 marks for two quizzes (5 marks each)
5 marks for tutorial participation

Tutors :

ERICK PURWANTO (erickp@comp.nus.edu.sg)

ZHANG JIANGWEI (jiangwei@nus.edu.sg)

Tutorials : Start next week. Information available at course home page

Prerequisites : (CS2010 or its equivalent) and (CS1231 or MA1100)

Acknowledgment : We thank Prof. Sanjay Jain for sharing with us his course material.

Book :

Title : Algorithms

Authors : R. Johnsonbaugh and M. Schaefer

Publication : Pearson Prentice Hall, 2004 (International Edition)

Other reference books mentioned in the course home page : http://www.comp.nus.edu.sg/~rahul/CS3230.html

Regarding CS3230R

According to my information current implementation of the R-modules is as follows:

- Discuss with the lecturer that you would like to do the R-module.
- The lecturer will decide whether it is appropriate for you after teaching you for a period (around middle of semester or end of semester).
- Start work on it after the lecturer has decided (middle of the current semester or at the next semester). The course will be registered only in the following semester.

What we cover in the course

Sorting/Searching/Selection

- A lower bound for the sorting problem
- Counting sort and Radix sort
- Topological sort of graphs

Divide and Conquer

- A Tiling problem
- Strassen's Matrix Product Algorithm
- Finding closest pair of points on the plane

Greedy Algorithms

Kruskal's algorithm for Minimum Spanning Tree
Prim's algorithm for Minimum Spanning Tree
Dijkstra's algorithm for finding shortest path between a pair of points in a graph
Huffman codes
The continuous Knapsack problem

Dynamic Programming

Computing Fibonacci numbers
Coin changing
The algorithm of Floyd and Warshall

What we cover in the course

- P v/s NP
  - Polynomial time, Non-deterministic algorithms, NP
  - Reducibility and NP-completeness, NP complete problems

- How to deal with NP hard problems
  - Brute force
  - Randomness
  - Approximation
  - Parameterization
  - Heuristics

Assume familiarity with :

- Basic data structures like Stacks, Queues, Linked lists, Arrays, Binary Trees, Binary Heaps,
- Basic sorting algorithms like Heap sort
- Basic search algorithms like Depth-First search, Breadth-First search
- Basic mathematical concepts like Sets, Mathematical Induction, Graphs, Trees, Logarithm

What is an Algorithm ?

Abū ʿAbdallāh Muḥammad ibn Mūsā al-Khwārizmī (c. 780 – c. 850)

A Persian mathematician, astronomer and geographer.

Properties :

- Input
- Output
- Precision
- Determinism

  (exceptions: randomization, quantum etc.)
- Finiteness
- Correctness
- Generality

Analysis :

- Correctness
- Termination
- Time analysis
- Space analysis

Pseudocode : Has precision, structure, universality. It is unambiguous, clear enough (not too rigorous, will not be concerned with semi-colons, uppercase, lowercase etc.)

Pseudocode - Example

Algorithm for finding maximum element in an array

Input Parameters : s

Output Parameters: None

$array - max(s)$

{

  $large = s[1]$

  $i = 2$

  $while(i \leq s.last)$

  {

    if $(s[i] > large)$ // larger value found

      $large = s[i]$

    $i = i + 1$

  }

  return $large$

}

---

if (condition)

        action 1

else

        action 2

---

do

{

        action

} while (condition)

---

for var = init to limit

        action

---

for var = init downto limit

        action

Algorithm analysis
Worst case, Average case

Worst case analysis :

t(n) = Maximum units of time taken by the algorithm to terminate for an input of size n.

Average case analysis :

t(n) = the average units of time taken by the algorithm to terminate for an input of size n.

Similar analysis can also be done for space required by the algorithm.

Input Parameters : s
Output Parameters: None

```
array − max(s)
{
  large = s[1]
  i = 2
  while(i ≤ s.last)
  {
    if (s[i] > large) // larger value found
      large = s[i]
    i = i + 1
  }
  return large
}
```

The worst case and average case times of the algorithm on an array of size n are each

constant * (n-1)

since the while loop is always executed (n-1) times, and every other operation takes constant time.

Big Oh, Omega and Theta

1. We say '$f(n) = O(g(n))$' OR '$f(n)$ is order at most $g(n)$' OR '$f(n)$ is big oh of $g(n)$', if there exists constants C_1 and N_1 such that, for all $n \geq N_1$

$$f(n) \leq C_1 g(n)$$

   g is an asymptotic upper bound for f, e.g. $1000n = O(n^2)$

2. We say '$f(n) = \Omega(g(n))$' OR '$f(n)$ is order at least $g(n)$' OR '$f(n)$ is omega of $g(n)$', if there exists constants C_2 and N_2 such that, for all $n \geq N_2$

$$f(n) \geq C_2 g(n)$$

   g is an asymptotic lower bound for f, e.g. $2^n = \Omega(50n \log n)$

3. We say '$f(n) = \Theta(g(n))$' OR '$f(n)$ is order $g(n)$' OR '$f(n)$ is theta of $g(n)$', if

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

   g is an asymptotic tight bound for f, e.g. $60n^2 - 100n + 50 = \Theta(30n^2 + 60n + 70)$

Recurrence Relations

Fibonacci sequence :

$$f_n = f_{n-1} + f_{n-2}, \quad n \geq 3$$

$$f_1 = f_2 = 1$$

Example : $c_n = c_{n-1} + n, \quad n \geq 1; \quad c_0 = 0$

$$\begin{aligned} c_n &= c_{n-1} + n \\ &= c_{n-2} + (n-1) + n \\ &= c_{n-3} + (n-2) + (n-1) + n \\ &\vdots \\ &= 0 + 1 + 2 + 3 + \ldots + (n-1) + n \\ &= \frac{n(n+1)}{2}. \end{aligned}$$

Examples

```
example(n){
if (n == 1)
   x = x + 1
   return
for i = 1 to n
   x = x + 1
example(n/2)
}
```

Let c_n be the number of times the statement $x = x + 1$ is executed. Then,

$c_n = n + c_{n/2}, \quad c_1 = 1$
$c_n = n + n/2 + \cdots + 1 = 2n - 1 = \Theta(n)$

```
j = n
while (j ≥ 1)
{
for i = 1 to j
   {x = x + 1}
j = j/2
}
```

If $n = 2^k$, the number of times the statement $x = x + 1$ is executed is

$n + n/2 + n/4 + \cdots + 1 = 2n - 1 = \Theta(n)$

Main Recurrence Theorem

Let $a \geq 1, b \geq 2, k \geq 0$.

Upper Bound: If $T(n) \leq aT(n/b) + f(n)$ and $f(n) = O(n^k)$, then $T(n) =$

1. $O(n^k)$ if $a < b^k$
2. $O(n^k \log n)$ if $a = b^k$
3. $O(n^{\log_b a})$ if $a > b^k$

Lower Bound: If $T(n) \geq aT(n/b) + f(n)$ and $f(n) = \Omega(n^k)$, then $T(n) =$

1. $\Omega(n^k)$ if $a < b^k$
2. $\Omega(n^k \log n)$ if $a = b^k$
3. $\Omega(n^{\log_b a})$ if $a > b^k$

Exact: If $T(n) = aT(n/b) + f(n)$ and $f(n) = \Theta(n^k)$, then $T(n) =$

1. $\Theta(n^k)$ if $a < b^k$
2. $\Theta(n^k \log n)$ if $a = b^k$
3. $\Theta(n^{\log_b a})$ if $a > b^k$

Proof Idea

Suppose (for simplicity) $T(n) = aT(n/b) + cn^k$ and $n = b^m$. Then,

$$\begin{aligned} T(n) = T(b^m) &= aT(b^{m-1}) + c(b^k)^m \\ &= a[aT(b^{m-2}) + c(b^k)^{m-1}] + c(b^k)^m \\ &= a^2T(b^{m-2}) + c[a(b^k)^{m-1}] + (b^k)^m] \\ &\vdots \\ &= a^mT(b^0) + c\sum_{i=1}^{m} a^{m-i}(b^k)^i \end{aligned}$$

If $a \neq b^k$,

$$\begin{aligned} T(n) &= a^mT(1) + c\left[\frac{(b^k)^{m+1} - a^{m+1}}{b^k - a} - a^m\right] \\ &= C_1 n^{\log_b a} + C_2 n^k \end{aligned}$$

where $C_1 = T(1) - \left[\frac{cb^k}{b^k - a}\right]$ and $C_2 = \frac{cb^k}{b^k - a}$.

Proof Idea contd.

$$\begin{aligned} \text{If } a = b^k, \quad T(n) &= a^m T(1) + c\sum_{i=1}^{m} a^m \\ &= a^m T(1) + cma^m \\ &= C_3 n^k + C_4 n^k \log_b n \end{aligned}$$

where $C_3 = T(1)$ and $C_4 = c$

$$\text{If } a \neq b^k, \quad T(n) = C_1 n^{\log_b a} + C_2 n^k$$

Therefore $T(n) =$

1. $\Theta(n^k)$ if $a < b^k$
2. $\Theta(n^k \log n)$ if $a = b^k$
3. $\Theta(n^{\log_b a})$ if $a > b^k$

Similar ideas work for other cases to get the full proof.

With today's lecture and the material covered in pre-requisite courses we have covered Chapters 1, 2, 3, 4 from the book.

Divide and Conquer

Divide and Conquer

1. If the problem is small solve it directly.

2. If the problem is big, divide it into subproblems. Solve the subproblems, again using divide and conquer.

3. Combine the solutions of the subproblem to get the solution of the original problem.

A Tiling Problem

Input Parameters: n, a power of 2 (the board size) and the location L of the missing square

Output Parameters: None

$tile(n, L)$ {
if $(n == 2)$ {
// the board is a right tromino T
tile with T
return
}
divide the board into four $n/2 \times n/2$ subboards
place one tromino in the centre depending on L
// each of the 1×1 squares in this tromino is considered as missing
let m_1, m_2, m_3, m_4 denote the locations of the missing squares
$tile(n/2, m_1)$
$tile(n/2, m_2)$
$tile(n/2, m_3)$
$tile(n/2, m_4)$
}

Idea of the algorithm

- Divide the problem of size n by n into four subproblems each of size n/2 by n/2.
- Place one tromino at the centre to create missing cells in each subproblem.
- Solve the subproblems using recursion.
- 2 by 2 problem is solved directly.

Time Analysis

Let $T(n)$ be the worst case running time of the algorithm on input of size n. Then

$$T(n) = 4 \cdot T(n/2) + \text{ constant }, \quad n > 2 \quad ; \quad T(2) = \text{ constant}$$

Using Main Recurrence Theorem $T(n) = \Theta(n^2)$.

- Chu and Johnsonbaugh (1986) showed that if $n \neq 5$, and $n^2 - 1$ is divisible by 3 then the n by n deficient board can be tiled using trominoes.
- Some 5 by 5 boards can be tiled and some cannot.

Mergesort

This algorithm sorts the array $a[i], \ldots, a[j]$ in nondecreasing order.

Input Parameters: a (an array), i, j
Output Parameters: a

```
mergesort(a, i, j)  {
// if only one element, just return
if (i == j)
   return
// divide a into two nearly equal parts
m = (i + j)/2
// sort each half
mergesort(a, i, m)
mergesort(a, m + 1, j)
// merge the two sorted halves
merge(a, i, m, j)
}
```

Subroutine Merge

In the input given below, assume that $a[i], \ldots, a[m]$ and $a[m+1], \ldots, a[j]$ are each sorted in nondecreasing order. These two subarrays are merged into a single nondecreasing array.

Input Parameters: a, i, m, j
Output Parameters: a

```
merge(a, i, m, j)  {
p = i // index in a[i], ..., a[m]
q = m + 1 // index in a[m + 1], ..., a[j]
r = i // index in a local array c
while (p ≤ m && q ≤ j)  {
// copy smaller value to c
if (a[p] ≤ a[q])  {
c[r] = a[p]
p = p + 1
}
else  {
c[r] = a[q]
q = q + 1
}
r = r+1
}
// copy remainder, if any, of the first subarray to c
while (p ≤ m)  {
c[r] = a[p]
p = p + 1
r = r + 1
}
// copy remainder, if any, of the second subarray to c
while (q ≤ j)  {
c[r] = a[q]
q = q + 1
r = r + 1
}
// copy c back to a
for r = i to j
   a[r] = c[r]
}
```

Idea of the algorithm

- Divide the array of size n into two arrays of size n/2.
- Sort the two subarrays using recursion.
- Merge the two sorted parts using subroutine merge.

Mergesort – Time Analysis

Let $T(n)$ be the running time of the algorithm. Then

$$T(n) = 2 \cdot T_{n/2} + \Theta(n)$$

Using Main Recurrence Theorem $T(n) = \Theta(n \log n)$.

Stable sort: A sorting algorithm is stable if the relative positions of items with duplicate values are unchanged by the algorithm.

Mergesort is stable.

Strassen's Matrix Product Algorithm

A is $m \times p$ matrix, B is $p \times n$ matrix.

$C_{i,j} = \sum_{k=1}^{p} A_{ik} B_{kj}, \quad 1 \le i \le m, \quad 1 \le j \le n$

Input Parameters: A, B $(n \times n \; matrices)$
Output Parameters: C

$matrix - product(A, B, C)$ {
$n = A.last$
for $i = 1$ to n {
for $j = 1$ to n {
$C[i][j] = 0$
for $k = 1$ to n
$C[i][j] = C[i][j] + A[i][k] * B[k][j]$
}
}

Running time is $\Theta(n^3)$.

$$A = \begin{pmatrix} \mathbf{a}_{11} & \mathbf{a}_{12} \\ \mathbf{a}_{21} & \mathbf{a}_{22} \end{pmatrix}$$

$$B = \begin{pmatrix} \mathbf{b}_{11} & \mathbf{b}_{12} \\ \mathbf{b}_{21} & \mathbf{b}_{22} \end{pmatrix}$$

$$AB = \begin{pmatrix} \mathbf{a}_{11}\mathbf{b}_{11} + \mathbf{a}_{12}\mathbf{b}_{21} & \mathbf{a}_{11}\mathbf{b}_{12} + \mathbf{a}_{12}\mathbf{b}_{22} \\ \mathbf{a}_{21}\mathbf{b}_{11} + \mathbf{a}_{22}\mathbf{b}_{21} & \mathbf{a}_{21}\mathbf{b}_{12} + \mathbf{a}_{22}\mathbf{b}_{22} \end{pmatrix}$$

$$c_n = 8c_{n/2} + \Theta(n^2)$$
$$\Rightarrow c_n = \Theta(n^3)$$

Strassen's Matrix Product Algorithm

$$
\begin{aligned}
\mathbf{q}_1 &= (\mathbf{a}_{11} + \mathbf{a}_{22}) * (\mathbf{b}_{11} + \mathbf{b}_{22}) \\
\mathbf{q}_2 &= (\mathbf{a}_{21} + \mathbf{a}_{22}) * \mathbf{b}_{11} \\
\mathbf{q}_3 &= \mathbf{a}_{11} * (\mathbf{b}_{12} - \mathbf{b}_{22}) \\
\mathbf{q}_4 &= \mathbf{a}_{22} * (\mathbf{b}_{21} - \mathbf{b}_{11}) \\
\mathbf{q}_5 &= (\mathbf{a}_{11} + \mathbf{a}_{12}) * \mathbf{b}_{22} \\
\mathbf{q}_6 &= (\mathbf{a}_{21} - \mathbf{a}_{11}) * (\mathbf{b}_{11} + \mathbf{b}_{12}) \\
\mathbf{q}_7 &= (\mathbf{a}_{12} - \mathbf{a}_{22}) * (\mathbf{b}_{21} + \mathbf{b}_{22})
\end{aligned}
$$

$$c_n = 7c_{n/2} + \Theta(n^2)$$
$$c_n = \Theta(n^{\log_2 7}) = \Theta(n^{2.807})$$

$$
\begin{aligned}
AB &= \begin{pmatrix} \mathbf{q}_1 + \mathbf{q}_4 - \mathbf{q}_5 + \mathbf{q}_7 & \mathbf{q}_3 + \mathbf{q}_5 \\ \mathbf{q}_2 + \mathbf{q}_4 & \mathbf{q}_1 + \mathbf{q}_3 - \mathbf{q}_2 + \mathbf{q}_6 \end{pmatrix} \\
&= \begin{pmatrix} \mathbf{a}_{11}\mathbf{b}_{11} + \mathbf{a}_{12}\mathbf{b}_{21} & \mathbf{a}_{11}\mathbf{b}_{12} + \mathbf{a}_{12}\mathbf{b}_{22} \\ \mathbf{a}_{21}\mathbf{b}_{11} + \mathbf{a}_{22}\mathbf{b}_{21} & \mathbf{a}_{21}\mathbf{b}_{12} + \mathbf{a}_{22}\mathbf{b}_{22} \end{pmatrix}
\end{aligned}
$$

Algorithm due to Coppersmith and Winograd runs in time $\Theta(n^{2.376})$.

Idea of the algorithm

- Divide the problem of multiplying two n by n matrices into the problem of multiplying 7, n/2 by n/2 matrices (you also do constant number of additions of n/2 by n/2 matrices).

- Use recursion to solve the subproblems.

Finding a closest pair of points on the plane

$p[1], \ldots, p[n]$ is an array; $p.x$ represents x-coordinate and $p.y$ represents y-coordinate of p. The function $dist(p, q)$ returns the Euclidian distance between points p and q.

Input Parameters: p
Output Parameters: None

$closest - pair(p)$ {
$n = p.last$
$mergesort(p, 1, n)$ // sort by x-coordinate
return $rec - cl - pair(p, 1, n)$
}

// $rec - cl - pair$ assumes that the input is sorted by x-coordinate
// At termination, the input is sorted by y-coordinate (also)

$rec - cl - pair(p, i, j)$ {
if $(j - i < 3)$ {
$mergesort(p, i, j)$ // sort by y-coordinate
// find the distance δ between a closest pair
$\delta = dist(p[i], p[i + 1])$
if $(j - i == 1)$ // two points
return δ
// three points
if $(dist(p[i + 1], p[i + 2]) < \delta)$
$\delta = dist(p[i + 1], p[i + 2])$
if $(dist(p[i], p[i + 2]) < \delta)$
$\delta = dist(p[i], p[i + 2])$
return δ
}

$k = (i + j)/2$
$l = p[k].x$
$\delta_L = rec - cl - pair(p, i, k)$
$\delta_R = rec - cl - pair(p, k + 1, j)$
$\delta = min(\delta_L, \delta_R)$
// $p[i], \ldots, p[k]$ is now sorted by y-coordinate, and
// $p[k + 1], \ldots, p[j]$ is now sorted by y-coordinate
$merge(p, i, k, j)$
// $p[i], \ldots, p[j]$ is now sorted by y-coordinate
// store points in the vertical strip in v
$t = 0$
for $k = i$ to j {
if $(p[k].x > l - \delta$ && $p[k].x < l + \delta)$ {
$t = t + 1$
$v[t] = p[k]$
}
// look for closer pairs in the strip by comparing
// each point in the stip to the next 7 points
for $k = 1$ to $t - 1$
for $s = k + 1$ to $min(t, k + 7)$
$\delta = min(\delta, dist(v[k], v[s]))$
return δ
}

Idea of the algorithm

- Divide the set of n points on the plane into two halfs (using x-coordinate).
- Find the shortest distance between pairs of points in the two halfs using recursion. Let delta be their minimum.
- Consider a strip of size 2 times delta in the middle of the two halfs.
- Sort the points in this strip using y-coordinate (this can be done only using merge since the subparts are already sorted according to y-coordinate).
- Start from bottom and compare each point with the next seven points to identify the closest pair (this works because in each box that we considered there can be at most 8 points).

Time Analysis

Let a_n be the worst case time taken by $rec - cl - pair$ on input of size n. Then

$$a_n = 2 \cdot a_{n/2} + \Theta(n)$$

which means $a_n = \Theta(n \log n)$. Sorting by the x-coordinate takes time $\Theta(n \log n)$; thus the worst case time of $closest - pair$ is $\Theta(n \log n)$.

What we did last time

Divide and conquer:

1. Tiling problem
2. Mergesort
3. Strassen's matrix product
4. Finding closest pair of points on a plane

Sorting and Selection

Insertion Sort

Input Parameter: a
Output Parameter: a

$insertion - sort(a)$ {
$n = a.last$
for $i = 2$ to n {
// save $a[i]$ so it can be inserted into the correct place
$val = a[i]$
$j = i - 1$
// if $val < a[j]$, move $a[j]$ right to make room for $a[i]$
while ($j \geq 1$ && $val < a[j]$) {
$a[j + 1] = a[j]$
$j = j - 1$
}
$a[j + 1] = val$ // insert val
}
}

Time: $1 + 2 + \ldots + (n - 1) = \frac{(n-1)n}{2} = \Theta(n^2)$

Idea of the algorithm

- Assume that the elements of the array arrive one by one.
- Keep inserting them at the right place in the current subarray which is already sorted.
- Nice property: It is an online algorithm.

Quicksort

The algorithm 'partition' inserts val at the index h such that values less than val are on the left of index h and values at least val are on the right of h.

Input Parameters: a, i, j
Output Parameters: a

$partition(a, i, j)$ {
$val = a[i]$
$h = i$
for $k = i + 1$ to j {
if $(a[k] < val)$ {
$h = h + 1$
$swap(a[h], a[k])$
}
}
$swap(a[i], a[h])$
return h
}

$swap(a, b)$ {
$c = a$
$a = b$
$b = c$
}

Quicksort Algorithm

Input Parameters: a, i, j
Output Parameters: a

$quicksort(a, i, j)$ {
if $(i < j)$ {
$p = partition(a, i, j)$
$quicksort(a, i, p - 1)$
$quicksort(a, p + 1, j)$
}

Quicksort : Time analysis

Lower Bound: If the array is already in increasing order (or decreasing order), time taken is :

$$(n-1)+(n-2)+(n-3)+\ldots+1=\frac{(n-1)n}{2}=\Omega(n^2)$$

Upper bound: Let c_n be the time taken on input of size n. We show $c_n \leq c_1 n^2$ using mathematical induction. Base case $n=1$ is true. We have

$$\begin{aligned} c_n &\leq n+\max_{1\leq p\leq n} c_{p-1}+c_{n-p} \\ &\leq n+\max_{1\leq p\leq n} c_1(p-1)^2+c_1(n-p)^2 \\ &\leq n+c_1(n-1)^2 \\ &\leq c_1 n^2 \end{aligned}$$

Randomized Quicksort

Below we assume that $rand(i, j)$ executes in constant time and returns a random integer between i and j, inclusive.

Input Parameters: a, i, j
Output Paramters: a

$random - partition(a, i, j)$ {
$k = rand(i, j)$
$swap(a[i], a[k])$
return $partition(i, j)$
}

Randomized Quicksort

Input Parameters: a, i, j
Output Parameters: a

$random - quicksort(a, i, j)$ {
if $(i < j)$ {
$p = random - partition(a, i, j)$
$random - quicksort(a, i, p - 1)$
$random - quicksort(a, p + 1, j)$
}

The *expected* run time of random quicksort is $\Theta(n \log n)$. Proof done in class. Idea: when the element is chosen randomly, it lies near the middle (in expectation). This makes the two subarrays of size roughly half.

Idea for partition algorithm

1. Let the current stage of array be like : F SSSS LLLL C RRRRR
2. F is first element, S < F, L ≥ F, C is the current element, R is rest of the elements.
3. If current element C ≥ F, then don't do anything, just let next element on right to be C.
4. If C < F, then swap C with first L, and let next element on right to be C.
5. At the end swap F with the last S element.

Idea of the algorithms

- Take an element of the array (of value val) and place it at the right index h. Elements to the left of h are less than val and to the right are greater than or equal to val (partition algorithm).
- Recursively sort a[i] … a[h-1] and a[h+1] … a[j]
- In normal quicksort, the element is taken to be the first element of the array.
- In randomized quicksort, the element is randomly chosen among all elements.

Quicksort is quite good in practice that is why ‘quicksort’.

No extra array is needed.

Quicksort is not stable.

A lower bound for comparison based algorithms

The worst case time of a comparison based sorting algorithm is $\Omega(n \log n)$.

Proof done in class. Idea: Any comparison based algorithm will have a decision tree, with n! leaves (for n bit inputs). Height of this tree is $\Omega(n \log n)$.

A decision tree for the comparison based algorithm is a binary tree with the nodes representing comparisons ($a < b$?), the left subtree represents the continuation of the algorithm on the result of comparison being yes and the right subtree represents the continuation of the algorithm on the result of comparison being no.

Counting Sort

- Not a comparison based algorithm, uses some information about the inputs i.e. the elements are numbers from 0 to m.
- Running time is linear, lower bound of Ω(n log n) does not apply.

```
Input Parameters: a, m
Output Parameters: a

counting − sort(a, m)  {
// set c[k] = the number of
// occurrences of value k in the array a
// begin by initializing c to zero
for k = 0 to m
   c[k] = 0
n = a.last
for i = 1 to n
   c[a[i]] = c[a[i]] + 1
// modify c so that c[k] = number of elements ≤ k
for k = 1 to m
   c[k] = c[k] + c[k − 1]
// sort a with the result in b
for i = n downto 1  {
b[c[a[i]]] = a[i]
c[a[i]] = c[a[i]] − 1
}
// copy b back to a
for i = 1 to n
   a[i] = b[i]
}
```

Time analysis

- Since each loop runs in time $\theta(m)$ or $\theta(n)$, the running time is $\theta(m + n)$.

Idea of algorithm

- First make sure that c[k] = number of occurrences of value k in input array a.
- Then make sure that c[k] = number of occurrences of value less than or equal to k in input array a.
- Produce sorted array b using c.
- Copy b back to a.

Counting sort is stable (done in tutorial).

Radix Sort

Again not a comparision based algorithm. Uses information about the input.
Uses $counting - sort$ is stable.

Below each integer in the input array a has at most k digits.

Input Parameters: a, k
Output Parameters: a

$radix - sort(a, k)$ {
for $i = 0$ to $k - 1$
  $counting - sort(a, 10)$ // key is digit in 10^i's place
}

Idea of algorithm: Starting from the least significant digit, sort using counting-sort.

Correctness of the algorithm shown in class. Running time is done in tutorial.

Radix sort was originally used to sort punch cards.

(Random) Selection

The algorithm below finds the k-th number, in the non-decreasing order, in $a[i], \ldots, a[j]$. It uses the $random-partition$ algorithm that we discussed before.

Input Parameters: a, i, j, k
Output Paramters: a

$random - select(a, i, j, k)$ {
if $(i < j)$ {
$p = random - partition(a, i, j)$
if $(k == p)$
return
if $(k < p)$
$random - select(a, i, p-1, k)$
else
$random - select(a, p+1, j, k)$
}
}

Finding median is a special case of this.

Idea of algorithm

- Using random-partition, get location p (note that everything to the left of p will be smaller than the value at p and everything to the right will be at least the value at p.)
- If k = p then stop.
- If k < p, act recursively on the left of p.
- Else act recursively on the right of p.

Running Time : The expected running time is $\theta(n)$. Proof done in class. Idea similar to the analysis of random-partition. When element is selected randomly, p is near middle (in expectation).

- The worst case time of random-select is $\theta(n^2)$ (done in tutorial).
- Any deterministic algorithm solving selection must take time at least n.
- Deterministic algorithm due to Blum, Floyd, Pratt, Rivest and Tarjan (1973) runs in time $\theta(n)$.

What we did last time

- Selection sort
- Quicksort
- Lower bound for comparison based algorithms
- Couting sort, Radix sort (not comparison based)
- Random Selection

Greedy Algorithms

Greed is Good !

Coin Changing

The algorithm makes change for input amount A using denominations

$$denom[1] > denom[2] > \ldots > denom[n] = 1$$

Input Parameters : $denom, A$
Output Parameters: None

$greedy - coin - change(denom, A)$ {
$i = 1$
while $(A > 0)$ {
$c = A/denom[i]$
println("use " + c +"coins of denomination " + $denom[i]$)
$A = A - c * denom[i]$
$i = i + 1$
}
}

- The algorithm is not optimal for denominations {1,6,10}.

Optimality for {1,5,10}

The algorithm is optimal for denominations {1,5,10} for every input amount A. Idea: use mathematical induction.

- Let greedy(A) represent the number of coins used by the greedy algorithm for input amount A.
- Let optimal(A) represent the optimal number of coins for amount A.
- Can verify by direct calculation for A = 1, 2, … , 9, that greedy(A) = optimal(A).
- Let A ≥ 10. Then by induction hypothesis: greedy(A-10) = optimal(A-10).
- Note that greedy(A) = 1+ greedy(A-10).
- Also note that optimal(A) = 1 + optimal(A-10) (since optimal solution must use at least one 10 dollar coin for A ≥ 10).
- Hence greedy(A) = optimal(A).

What goes wrong with {1,6,10} in this argument ?

Kruskal's Algorithm

Kruskal's algorithm finds a minimal spanning tree in a connected, weighted graph G with vertex set {1, …, n}.

A spanning tree is a tree T such that every pair of vertices are connected via edges in T. A minimal spanning tree is a spanning tree such that the sum of the weights of all its edges is the least among all spanning trees.

Note: A forest with (n-1) edges is a spanning tree.

Idea (Greedy approach):

1. Start with S having no edges.
2. Add an edge of minimum weight not contained in S, to S, such that S does not contain a cycle.
3. Keep doing this till S contains n-1 edges.
4. At the end since S is a forest with n-1 edges, it must be a spanning tree.

Kruskal's Algorithm

The input to the algorithm is edgelist, an array of edge, and n. The members of edge are:

1. v and w, the vertices on which the edge is incident.
2. weight, the weight of the edge.

The algorithm uses subroutines that manipulate sets of vertices:

1. makeset(v): makes a set containing the vertex v alone.
2. findset(v) : returns the name (e.g. the least element in it) of the set containing vertex v.
3. union(v,w): does the union of the sets containing v and w.

We can assume that these subroutines run in time O(log n).

Kruskal's Algorithm

Input Parameters: $edgelist, n$
Output Parameters: None

```
kruskal(edgelist, n)  {
// sort according to the weight of the edges
// in nondecreasing order
mergesort(edgelist)
for i = 1 to n
   makeset(i)
count = 0
i = 1
while (count < n − 1)  {
if (findset(edgelist[i].v) != findset(edgelist[i].w))  {
println(edgelist[i].v + " " + edgelist[i].w)
count = count + 1
union(edgelist[i].v, edgelist[i].w)
}
i = i + 1
}
}
```

Time Analysis:

- Since the graph is connected m ≥ n-1.
- There are O(m) makeset, findset and union operations, time taken is O(m log m).
- Sorting takes time θ(m log m).
- Hence total time is θ(m log m).

Correctness of Kruskal's Algorithm
Proof Idea

- Let S be the set of edges chosen so far by the algorithm.
- We show by induction that S is a part of a minimal spanning tree.
- Base case is true since at the beginning, S is empty.
- Let e be the new edge chosen by the algorithm. Let the statement be true for S and we show it is true for S U {e}.
- Let T be a minimal spanning tree containing S. If T contains S U {e} then we are done. Otherwise T U {e} must form a cycle C.
- Let e1 be an edge in T not is S which is part of C. Then weight(e1) ≥ weight(e) (since: we could have added either e1 or e but we added e).
- Consider T1 = T U {e} – {e1}. Then T1 is also a spanning tree and T1 contains S U {e}. Also weight (T) ≥ weight(T1). But T was a minimal spanning tree and hence T1 is also a minimal spanning tree. Hence S U {e} is part of a minimal spanning tree T1.

At the end of the algorithm S contains n-1 edges and is a part of minimal spanning tree and hence it is a minimal spanning tree.

Prim's Algorithm

Prim's algorithm finds a minimal spanning tree in a connected, weighted graph G with vertex set {1, …, n}.

Only difference with Kruskal's algorithm: Intermediate graph is a tree instead of a forest.

Idea (Greedy approach):

1. Start with S having no edges.
2. Add an edge e of minimum weight not contained in S, to S, such that S does not contain a cycle.
3. Ensure that one of the endpoints of e touches S and the other does not (this is the only difference with Kruskal's algorithm).
4. Keep doing this till S contains n-1 edges.
5. At the end since S is a tree with n-1 edges, it must be a spanning tree (proof of correctness done in class, similar to Kruskal Algorithm's proof of correctness).

Running time: Can be made $\theta(m \log n)$ by using binary heaps and can be made $\theta(m + \log n)$ by using Fibonacci heaps.

Quiz-1
(5 marks, 15 minutes, open book)

Write an algorithm (pseudocode) which has:

Input: An array a (an n element array, all elements distinct)

Output: A 2 dimensional array perm (which has n! rows and n columns; perm[i,j] represents the element in the ith row and the jth column). Each row of perm is a distinct permutation of the elements of a.

Quiz-1: One Solution

Input parameter : a
Output parameter : None

$creat - perm(a)$ {
$n = a.last$
if $(n == 1)$ {
$perm[1,1] = a[1]$
return $perm$ }
for $i = 1$ to n {
// create array b with all elements
// in a except $a[i]$
for $j = 1$ to $i - 1$
$b[j] = a[j]$
for $j = i + 1$ to n
$b[j - 1] = a[j]$
$p = create - perm(b)$
$startrow = (i - 1)(n - 1)!$
for $k = 1$ to $(n - 1)!$ {
$perm[startrow + k, 1] = a[i]$
for $j = 2$ to n
$perm[startrow + k, j] = p[k, j - 1]$
} }
return $perm$
}

Dijkstra's Algorithm

This algorithm takes as input a weighted graph and a start vertex. It outputs the shortest path from the start vertex to all other vertex in the in the graph.

Input Parameters: G (graph is input in the form of adjacency list which also contains the weights of the edges), start (start vertex)

Output Parameters: predecessor (array which tells for every vertex, the previous vertex in the shortest path from start), key (array which tells for every vertex the length of the shortest path from start)

$dijkstra(G, start, predecessor, key)$ {
$n =$ number of vertex in G
for $i = 1$ to n {
$key[i] = \infty$
$predecessor[i] = -1$ }
$key[start] = 0$
$S = \{\}$ // empty set
while S does not contain all vertex {
$v =$ vertex not in S with minimum key value
$S = S \cup \{v\}$
for every neighbour w of v not in S {
if $(key[v] + weight(v, w) < key[w])$ {
$key[w] = key[v] + weight(v, w)$
$predecessor[w] = v$ } } } }

Running time:

Can be made θ(m log n) by using binary heaps and can be made θ(m + log n) by using Fibonacci heaps.

Proof of correctness

Proof is by showing loop invariants. We maintain three loop invariants.

Let l[w] represent the length of a shortest path from start to w.

1. For every vertex w in G: key[w] ≥ l[w].
2. For every vertex w in S: key[w] = l[w].

At the end S contains every vertex in G and hence key[w]=l[w] for every vertex in G.

3. For every vertex w in S: predecessor[w]=u then l[w] = l[u]+ weight[u,w]

At the end S contains every vertex in G and hence u is the predecessor of w in a shortest path from start to w.

Base case is true.

Assume the three invariants are true at the beginning of some iteration of the loop. We show they are true at the end of that iteration as well. Let N(v) represent the neighbors of v for which key was decreased in this iteration.

1. For w in N(v) :

key[w] = key[v] + weight(v,w)
≥ l[v] + weight(v,w) (using 1. at the beginning of the iteration)
= length of some path from start to w
≥ l[w].

Hence invariant 1. is true at the end of the iteration as well.

Proof of correctness

2. Assume it is true for S. Need to show that invariant is true for S U {v}.

   Hence need to show key[v] = l[v]. We know key[v] ≥ l[v], because of invariant 1. at the beginning of the iteration. If key[v]=l[v] then we are done. Hence assume for contradiction that key[v] > l[v]. Let

   (start ... w' w ... v) be a shortest path from start to v such that (start … w') is in S and w is not in S. Note that (start … w' w) is also a shortest path from start to w. Hence

   key[v] > l[v] ≥ l[w] = l[w'] + weight[w',w]

   = key[w'] + weight[w',w] ≥ key[w]

   This is contradiction to the fact that key[v] was the smallest among vertices not in S.

3. Assume it is true for S. Need to show that invariant is true for S U {v}. Hence need to show l[v] = l[u]+ weight[u,v], where u = predecessor[v]. Note that predecessor[v] ≠ -1, since key[v] = l[v] ≠ ∞ (using invariant 2. for v just shown). Also note that u is in S.

   l[v] = key[v] = key[u]+ weight[u,v]

   = l[u] + weight[u,v] (using invariant 2. for S)

Huffman Codes

- A Huffman code for characters $a_1, a_2, \ldots, a_n$ is a prefix-free code (or just prefix code) of bits 0 and 1. That is each a_i is represented by a string of 0 and 1.

- A Huffman code can be represented by a binary tree (Huffman tree) with the leaves being the characters $a_1, a_2, \ldots, a_n$.

- Suppose we are given characters $a_1, a_2, \ldots, a_n$ with frequencies $f_1, f_2, \ldots, f_n$. Let there be a Huffman tree T with the path-length for character a_i (from the root) being p_i . The weighted path length of T is

  $$wpl(T) = p_1 f_1 + p_2 f_2 + \ldots + p_n f_n \ .$$

- Optimal Huffman tree T is a tree such that wpl(T) is minimum among all Huffman trees.

Huffman's Algorithm

Input Parameter : a : each element in the array is (character, frequency)
Output Parameter: None (the algorithm returns the root of an optimal Huffman tree)

```
huffman(a) {
n = a.last
copy a into h
for i = 1 to n − 1 {
let l be a character with least frequency among all characters in h
delete l and its frequency from h
let r be a character with least frequency among all characters in h
delete r and its frequency from h
create node v_i in the tree T
set l and r to be the left and right child of v_i
set frequency of v_i to be frequency of l plus the frequency of r
insert v_i and its frequency in h
}
return v_{n−1}
}
```

Time : Since each operation in the loop can be done in O(log n) total time is O(n log n).

Proof of correctness

Claim-1 : Let f_1 and f_2 be two least frequencies in $(f_1, f_2, \ldots, f_n)$. We show there exists an optimal Huffman tree in which f_1 and f_2 must be at the lowest level and also be siblings of each other.

Proof: Let T be an optimal Huffman tree. Let f_1 not be at the lowest level in T. Let f be at lowest level. Let p_1 be the path length of f_1 and let p be the path length of f in T. Then $p \geq p_1$ and $f \geq f_1$.

Let T_1 be a tree such that f and f_1 are swapped in T. Then

$$wpl(T_1) = wpl(T) - pf - p_1f_1 + pf_1 + p_1f$$

$$= wpl(T) - (p - p_1)(f - f_1) \leq wpl(T)$$

Hence T_1 is also an optimal Huffman tree.

Now if f_2 is not sibling of f_1 in T_1, swap the sibling of f_1 with f_2 to get tree T_2. By doing previous arguments we can say that T_2 is also an optimal Huffman tree.

Proof of Correctness

Claim-2 : Let T_A be the tree output by Huffman's algorithm. T_A is an optimal Huffman tree.

Proof: The proof is by induction on n.

Base case : n = 2 : Is easily seen since there is only one Huffman tree in this case.

Let n > 2. Let T_2 be an optimal Huffman tree as in Claim-1. Consider T_2' with f_1 and f_2 deleted in T_2 and the parent assigned frequency $f_1 + f_2$. Note that $wpl(T_2') = wpl(T_2) - f_1 - f_2$. Hence T_2' must be optimal for $(f_1+ f_2, f_3, \dots , f_n)$.

Let T_A' be with obtained from T_A with f_1 and f_2 deleted and the parent assigned frequency $f_1 + f_2$. Then we have $wpl(T_A') = wpl(T_A) - f_1 - f_2$. Note that T_A' is the output of the algorithm for frequencies $(f_1+ f_2, f_3, \dots , f_n)$.

By induction hypothesis for n-1, $wpl(T_A') = wpl(T_2')$. Hence

$$wpl(T_A) = wpl(T_A') + f_1 + f_2 = wpl(T_2') + f_1 + f_2 = wpl(T_2).$$

Hence T_A is an optimal Huffman tree.

The Continuous-Knapsack problem

$$\max \sum_{i=1}^{n} x_i p_i$$

subject to the constraints:

$$\sum_{i=1}^{n} x_i w_i \leq C, \quad x_i = \{0, 1\}$$

The 0/1 knapsack problem

$$\max \sum_{i=1}^{n} x_i p_i$$

subject to the constraints:

$$\sum_{i=1}^{n} x_i w_i \leq C, \quad 0 \leq x_i \leq 1$$

The continuous knapsack problem

Idea of the algorithm for continuous knapsack problem:
Greedy approach

1. Sort p_i/w_i in the non-increasing order.
2. Include objects in this order till your capacity C is exhausted.

The Continuous-Knapsack problem

Input Parameters: a, C
(each element $a[i]$ has two parts $a[i].p$ and $a[i].w$)
Ouput Parameters: x

$continuous - knapsack(a, C)$ {
$n = a.last$
for $i = 1$ to n {
  $ratio[i] = \frac{a[i].p}{a[i].w}$
  $x[i] = 0$ }
$mergesort(a, ratio)$
$W = 0$
$j = 1$
while $(j \leq n$ && $W < C)$ {
  if $(W + a[j].w \leq C)$ {
    $x[j] = 1$
    $W = W + a[j].w$ }
  else {
    $x[j] = \frac{C-W}{a[j].w}$
    $W = C$ }
  $j = j + 1$ }
}

$$\max \sum_{i=1}^{n} x_i p_i$$

subject to the constraints:

$$\sum_{i=1}^{n} x_i w_i \leq C, \quad 0 \leq x_i \leq 1$$

Time Analysis:

While loop runs in time θ(n) and sorting takes time θ(n log n), hence overall time is θ(n log n).

Correctness

Let $\{x_i\}$ be the solution produced by the algorithm for input $\{C, w_i, p_i\}$.
Let $\{y_i\}$ be any other solution which satisfies the contraints.
Let k be the smallest index with $x_k < 1$. We will show,

$$\forall i: \quad (x_i - y_i)\left(\frac{p_i}{w_i}\right) \geq (x_i - y_i)\left(\frac{p_k}{w_k}\right), \text{ and}$$

$$\sum_{i=1}^{n}(x_i - y_i)w_i \geq 0\,.$$

This will imply:

$$\begin{aligned}
\sum_{i=1}^{n}(x_i - y_i)p_i &= \sum_{i=1}^{n}(x_i - y_i)w_i\frac{p_i}{w_i} \\
&\geq \sum_{i=1}^{n}(x_i - y_i)w_i\frac{p_k}{w_k} \\
&= \frac{p_k}{w_k}\sum_{i=1}^{n}(x_i - y_i)w_i \\
&\geq 0.
\end{aligned}$$

$$\Rightarrow \textstyle\sum_{i=1}^{n} x_i p_i \geq \sum_{i=1}^{n} y_i p_i \quad .$$

Correctness

$\sum_{i=1}^{n} x_i w_i = C \geq \sum_{i=1}^{n} y_i w_i$

$\Rightarrow \sum_{i=1}^{n} (x_i - y_i) w_i \geq 0$.

$\max \sum_{i=1}^{n} x_i p_i$

subject to the constraints:

$\sum_{i=1}^{n} x_i w_i \leq C, \quad 0 \leq x_i \leq 1$

Recall that k is the smallest index with $x_k < 1$.

For $i < k$, since $x_i = 1$, we have $x_i \geq y_i$. Also $\frac{p_i}{w_i} \geq \frac{p_k}{w_k}$ (because the ratio of profit to weight is sorted in non-increasing order). Hence

$$(x_i - y_i)\left(\frac{p_i}{w_i}\right) \geq (x_i - y_i)\left(\frac{p_k}{w_k}\right).$$

For $i = k$ we have $(x_i - y_i)\left(\frac{p_i}{w_i}\right) = (x_i - y_i)\left(\frac{p_k}{w_k}\right)$.

For $i > k$ since $x_i = 0$, we have $x_i \leq y_i$. Also $\frac{p_i}{w_i} \leq \frac{p_k}{w_k}$ (because the ratio of profit to weight is sorted in non-increasing order). Hence

$$(x_i - y_i)\left(\frac{p_i}{w_i}\right) \geq (x_i - y_i)\left(\frac{p_k}{w_k}\right).$$

Applications in real life ?

- Dijkstra shortest path algorithm for google maps.
- Sorting for library, dictionary
- Searching for internet websites

Dynamic Programming

Basic of dynamic programming

- In order to solve the big problem solve sub-problems first.
- The difference with Divide-and-Conquer is that in this case it is not clear which sub-problems should be solved.
- Therefore in dynamic programming all sub-problems that might be needed are solved.
- First the simplest sub-problems are solved and then the more complex ones are solved, all the way up to the original problem.
- The results of the sub-problems are stored in a table and used later whenever needed.

Computing Fibonacci Numbers

$$f_n = f_{n-1} + f_{n-2} \quad ; \quad f_1 = f_2 = 1$$

Input Parameter: n
Output Parameter: None

```
fibonacci − recurs(n) {
if (n == 1)
   return 1
if (n == 2)
   return 1
return(fibonacci − recurs(n − 2)
   +fibonacci − recurs(n − 1) )
}
```

Very inefficient. f_1 , f_2, f_3 etc. are calculated exponentially many times.

Time : $\theta(2^n)$

Input Parameter: n
Output Parameter: None

```
fibonacci − iterative(n) {
// f is a local array
if (n == 1)
   f[1] = 1
if (n == 2)
   f[2] = 1
for i = 3 to n {
   f[i] = f[i − 1] + f[i − 2]
return f[n]
}
```

Efficient : Time is $\theta(n)$

This is an example of a dynamic program.

Memorized recursive algorithm

```
mem − fibonacci(n) {
for i = 1 to n
   results[i] = −1 // −1 means undefined
return mem − fibonacci − recurs(results, n)
}

mem − fibonacci − recurs(results, n) {
if (results[n] ! =  − 1)
   return results[n]
if (n == 1)
   val = 1
else if (n == 2)
   val = 1
else {
   val = mem − fibonacci − recurs(results, n − 2)
   val = val + mem − fibonacci − recurs(results, n − 1)
}
results[n] = val
return val
}
```

Time : θ(n)

Coin-changing revisited

$$denom[1] > denom[2] > \ldots > denom[n] = 1$$

Input Parameters: $denom, A$
Output Parameters: $C, used$

```
dynamic − coin − change(denom, A, C, used) {
n = denom.last
for j = 0 to A {
   C[n][j] = j
   used[n][j] = true
}
for i = n − 1 downto 1
   for j = 0 to A
      if (denom[i] > j || C[i + 1][j] < 1 + C[i][j − denom[i]]) {
         C[i][j] = C[i + 1][j]
         used[i][j] = false
      }
      else {
         C[i][j] = 1 + C[i][j − denom[i]]
         used[i][j] = true
      }
}
```

Idea : Let $c_{i,j}$ represent the minimum number of coins of denominations denom[i] to denom[n] needed to get amount j.

Start with $c_{n,j} = j$ for all j from 0 to A. Iteratively calculate by running i from n-1 downto 1 and j from 0 to A:

$c_{i,j} = \min\{c_{i+1,j}, 1 + c_{i,j-denom[i]}\}$

The idea is that for finding $c_{i,j}$, either

a) you use denom[i] in which case $c_{i,j} = 1 + c_{i,j-denom[i]}$
b) or you do not use denom[i] in which case $c_{i,j} = c_{i+1,j}$

Time : $\theta(n A)$

Optimal substructure property

- If S is an optimal solution to a problem, then the components of S are optimal solutions to subproblems.
- For a dynamic-programming algorithm to solve an optimization problem correctly, the optimal substructure property must hold.
- This property holds for coin-changing problem.
- This property does not hold for longest-simple-path problem : For a connected weighted graph, and vertices (v,w), find the longest simple path from v to w.

Midterm Answer 1

$$T(n) = T(n-1) + n^2 \quad ; \quad n \geq 1$$

$T(n) = n^2 + (n-1)^2 + (n-2)^2 + \ldots + 1^2 + T(0).$

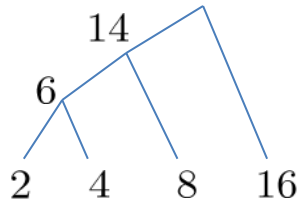
Therefore $T(n) \leq n \cdot n^2 + T(0) = O(n^3)$.

Also $T(n) \geq n^2 + (n-1)^2 + (n-2)^2 + \ldots + (\frac{n}{2})^2 \geq \frac{n}{2} \cdot (\frac{n}{2})^2 = \Omega(n^3)$.

Hence overall $T(n) = \Theta(n^3)$.

Midterm Answer 2

$f_1 = 2, f_2 = 4, f_3 = 8, f_4 = 16.$



weighted path length $= 2 * 3 + 4 * 3 + 8 * 2 + 16 * 1 = 50.$

For general n: weighted path length $=$

$$2^n \cdot 1 + 2^{n-1} \cdot 2 + 2^{n-2} \cdot 3 + \ldots + 4 \cdot (n-1) + 2 \cdot (n-1)$$

$$= \sum_{i=1}^{n} i \cdot 2^{n-i+1} - 2 \quad = \quad 2 \cdot \sum_{i=1}^{n} i \cdot 2^{n-i} - 2 \quad = \quad 2\left(\sum_{i=1}^{n} i \cdot 2^{n-i} - 1\right)$$

$$= 2\left(\sum_{i=1}^{n} (n - (n-i)) \cdot 2^{n-i} - 1\right)$$

$$= 2\left(\sum_{i=1}^{n} n \cdot 2^{n-i} - \sum_{i=1}^{n} (n-i) \cdot 2^{n-i} - 1\right)$$

$$= 2\left(n \sum_{i=0}^{n-1} 2^i - \sum_{i=1}^{n-1} i \cdot 2^i - 1\right)$$

$$= 2\left(n\left(2^n - 1\right) - \left((n-2)2^n + 2\right) - 1\right) \quad = \quad 2^{n+2} - 2n - 6$$

Midterm Answer 2

For general n: weighted path length $=$

$$\left(2^n \cdot 1 + 2^{n-1} \cdot 2 + 2^{n-2} \cdot 3 + \ldots + 4 \cdot (n-1) + 2 \cdot n\right) - 2$$

$$= 2^{n+2} - 2 \cdot n - 6$$

$$\begin{array}{lrl}
 & 2^n + 2^{n-1} + \ldots + 8 + 4 + 2 & = 2^{n+1} - 2 \\
+ & 2^{n-1} + \ldots + 8 + 4 + 2 & = 2^n - 2 \\
 & \vdots & \vdots \\
+ & 8 + 4 + 2 & \\
+ & 4 + 2 & = 2^3 - 2 \\
+ & 2 & = 2^2 - 2
\end{array}$$

$$= \left(2^{n+2} - 4\right) - 2 \cdot n$$

Midterm Answer 3

Idea:

1. Sort S in increasing order.
2. Set n to be the size of S.
3. Set $j = 0, last = 1$.
4. While $last \leq n$

   Choose k to be maximum index such that $S[k] - S[last] \leq D$.

   Set $j = j + 1, last = k + 1$.
5. Output j.

Input Parameters : a, D
Ouput Parameters: None

```
set − partition(a, D) {
n = a.last
// sort a in increasing order
mergesort(a)
j = 0
last = 1
while (last ≤ n) {
   k = last + 1
   while (k ≤ n && a[k] − a[last] ≤ D)
      { k = k + 1 }
   j = j + 1
   last = k
}
return j
}
```

Time : While loop will take time O(n) and mergesort will take time θ(n log n). Hence total time is θ(n log n).

Midterm Answer 3

The proof of correctness will be via induction on n which is the size of S.

Base case $n = 1$ is easily seen since then there is just one partition.

Let $n > 1$. Let $S_1, S_2, \ldots, S_r$ be the partition of S that is produced by the algorithm on input (S, D).

Let $T_1, T_2, \ldots, T_m$ be an optimal partition of S for input (S, D).

Let $a[j]$ be the minimum element in a. Then $a[j] \in S_1$. Let $a[j] \in T_i$, then $T_i \subseteq S_1$.

Now move the elements in $S_1 - T_i$ (if any) from other subsets in optimal partition into T_i.

Let the new sets obtained be $T'_1, T'_2, \ldots, T'_m$. This is also an optimal partition of S. Also now $T'_i = S_1$.

Let $S' = S - S_1$. Note that $|S'| < |S|$.

Note that $S_2, \ldots, S_r$ will be the partition of S' produced by the algorithm on input (S', D).

Also note that $T'_1, \ldots, T'_{i-1}, T'_{i+1}, \ldots, T'_m$ will be an optimal partition of S' on input (S', D).

Therefore from induction hypothesis $r - 1 = m - 1$. Hence $r = m$.

Midterm Answer 4

Median(a,b)

1. Set n to be the number of elements in a (also in b).

2. If n = 1
   1. If a[1] ≥ b[1], then return b[1]
   2. Otherwise return a[1].

1. If a[n/2] ≥ b[n/2]
   1. Then note that

   a[n/2] ≥ Median(a U b) ≥ b[n/2+1]
   2. The problem now reduces to size n/2 and then we can proceed similarly.

5. If a[n/2] < b[n/2]
   1. Then note that
      a[n/2+1] ≤ Median(a U b) ≤ b[n/2]
   2. The problem now reduces to size n/2 and then we can proceed similarly.

Input Parameters: a, b
Output Parameters: None

```
median(a, b) {
n = a.last
i = k = 1
j = l = n
while (j > i && l > k) {
   if (a[(i + j − 1)/2] ≥ b[(k + l − 1)/2]) {
      j = (i + j − 1)/2
      k = (k + l + 1)/2
   }
   else {
      i = (i + j + 1)/2
      l = (l + k − 1)/2
   }
}
if (a[i] ≥ b[l])
   return b[l]
else
   return a[i]
}
```

Time : With each iteration of the while loop, the difference between j and i (similarly the difference between l and k) reduces by factor of 2. Hence while loop can run log n times. Hence overall running time is O(log n).

Optimal Matrix Multiplication

$$M_{size[0],size[1]} \times M_{size[1],size[2]} \times \cdots \times M_{size[n-1],size[n]}$$

Need to multiply matrices by using least number of scalar multiplications.

Input Parameters : $size$
(First matrix is $size[0] \times size[1]$, second matrix is $size[1] \times size[2]$, nth matrix is $size[n-1] \times size[n]$)

Output Parameters: s

```
opt − matrix − mult(size, s) {
   n = size.last
for i = 1 to n
   s[i][i] = 0
for w = 1 to n − 1
   for i = 1 to n − w {
      j = w + i
      s[i][j] = ∞
      for k = i to j − 1 {
         q = s[i][k] + s[k + 1][j] + size[i − 1] · size[k] · size[j]
         if (q < s[i][j])
            s[i][j] = q
      }
   }
}
```

Idea : Let s[i,j] represent the minimum number of scalar multiplications needed to multiply ith matrix through jth matrix.

The idea is that to find s[i,j] you check for groupings (i,k) and (k,j) for all k between i and j; one of them must be optimal, so take their minimum.

Start with s[i,i]= 0 for all i. Iteratively calculate by increasing w from 0 to n-1 and increasing i from 1 to n-w :

s[i,i+w] =
$\min_k$\{s[i,k] + s[k+1,i+w] + size[i-1]size[k]size[i+w]\}

Time : $\theta(n^3)$

The longest common subsequence problem

Given two sequences a[1], …, a[m] and b[1], ….b[n]

find a subsequence $a[i_1], \ldots, a[i_k]$ of a[1], …, a[m]

and find a subsequence $b[j_1], \ldots, b[j_k]$ of b[1], …, b[n]

such that $a[i_1] = b[j_1] , \ldots, a[i_k] = b[j_k]$.

The task is to maximize k.

Example : For

a , b , c, d, e, f and a, e, c, f, d, g, p

A longest common subsequence is (a, c, d). Another is (a,c,f).

The longest common subsequence problem

The task is to find the longest common subsequence of $a[1], \ldots, a[m]$ and $b[1], \ldots, b[n]$.

Input Parameters: a, b
Output Parameters: c

$LCS(a, b, c)$ {
$m = a.last$
$n = b.last$
for $i = 0$ to m
  $c[i][0] = 0$
for $j = 0$ to n
  $c[0][j] = 0$
for $i = 1$ to m
  for $j = 1$ to n
    if $(a[i] != b[j])$
      $c[i][j] = \max(c[i-1][j], c[i][j-1])$
    else
      $c[i][j] = 1 + c[i-1][j-1]$
}

Time : θ(mn)

Idea : Let c[i][j] be the length of a longest common subsequence of
a[1], …, a[i] and b[1], …, b[j]
for i = 0, …, m and j=0, …, n.

The idea is that if a[i]=b[j] then a[i] appears in a longest common subsequence, otherwise a longest common subsequence needs to be found in
a[1] … a[i-1] and b[1] … b[j]
Or
a[1] … a[i] and b[1] … b[j-1]

Start with c[i][0] = 0 for all i and c[0][j]=0 for all j.
Iteratively calculate by increasing i from 1 to m and increasing j from 1 to n:

If a[i] != b[j] then
c[i][j] = max{c[i-1][j] , c[i][j-1]}

else
c[i][j] = 1+ c[i-1][j-1] .

Floyd's algorithm for All Pairs Shortest Paths

We want to find shortest paths between all pairs of vertices in a simple, undirected, weighted graph G. We are given matrix A (as a two dimensional array), such that A[i][j] is the weight of the edge (i,j), if there is edge (i,j). Otherwise A[i][j] is ∞ .

Idea: Let A^k represent the matrix such that $A^k[i][j]$ is the length of a shortest path from i to j, where the intermediate vertices allowed is {1,2, …, k}.

Initially A^0 = A, when no intermediate vertex is allowed.

Assume we have computed $A^{(k-1)}$. Then to compute $A^k[i][j]$:

a) If k appears in a shortest path between i and j with intermediate allowed vertices being {1, …, k}, then

$$A^k[i][j] = A^{(k-1)}[i][k] + A^{(k-1)}[k][j]$$

b) If k does not appear in a shortest path between i and j with intermediate allowed vertices being {1, …, k}, then

$$A^k[i][j] = A^{(k-1)}[i][j]$$

Floyd's algorithm for All Pairs Shortest Paths

In the algorithm below, at the end A[i][j] is the length of a shortest path between i and j. Also next[i][j] is the vertex after i, in a shortest path from i to j.

```
Input Parameter : A
Output Parameters: A, next

all − paths(A, next) {
n = A.last
// initialize next: if no intemediate
// vertices are allowed next[i][j] = j
for i = 1 to n
   for j = 1 to n
      next[i][j] = j
for k = 1 to n
   for i = 1 to n
      for j = 1 to n
         if (A[i][k] + A[k][j] < A[i][j]) {
            A[i][j] = A[i][k] + A[k][j]
            next[i][j] = next[i][k]
         }
}
```

Time : $\theta(n^3)$

Warshall's Algorithm

We are given a simple, undirected, weighted graph G. We want to find for all pairs of vertices, if they are connected or not? We are given adjacency matrix A (as a two dimensional array), such that A[i][j] is 1 if there is an edge (i,j); otherwise A[i][j] is 0 .

Idea: Let A^k represent the matrix such that $A^k[i][j]$ =1 if there is a path from i to j, where the intermediate vertices allowed is {1,2, …, k} (otherwise $A^k[i][j]$ =0).

Initially A^0 = A, when no intermediate vertex is allowed.

Assume we have computed $A^{(k-1)}$. Then to compute $A^k[i][j]$:

a) If k appears in a path between i and j with intermediate allowed vertices being {1, …, k}, then

$A^k[i][j] = A^{(k-1)}[i][k]$ AND $A^{(k-1)}[k][j]$

b) If k does not appear in a path between i and j with intermediate allowed vertices being {1, …, k}, then

$A^k[i][j] = A^{(k-1)}[i][j]$

Warshall's Algorithm

In the algorithm below, at the end A[i][j] =1 if there is a path between i and j.

Input Parameter : A
Output Parameters: A

```
warshall(A) {
n = A.last
for k = 1 to n
   for i = 1 to n
      for j = 1 to n
         if (A[i][k] + A[k][j] = 2) {
            A[i][j] = 1
         }
}
```

Time : $\theta(n^3)$

Let R be a relation on a set X (that is $R \subseteq X \times X$). The transitive closure of R is the set

$$R' = \{(x_1, x_k) \text{ such that } (x_1, x_2) \in R, (x_2, x_3) \in R, \ldots, (x_{k-1}, x_k) \in R\}.$$

Warshall's algorithm can also be used to find the transitive closure of R. R is input in the form of a matrix A such that $A(i, j) = 1$ if $(i, j) \in R$, otherwise $A(i, j) = 0$. Finding transitive closure is exactly like finding if (i, j) is connected in the graph with its adjacency matrix being A.

P and NP

(warning: NP is NOT ‘not P’)

Decision problems and Function problems

A decision problem L is a set of binary strings.

An algorithm A is said to accept an input string x, if A(x) = 1 .
(A(x) represents the output of the algorithm A on input x).

An algorithm A is said to reject an input string x, if A(x) = 0.

Algorithm A is said to decide L if A(x) =1 for all x in L and A(x) = 0 for all x not in L.

e.g. Graph connectivity problem : L is the binary encodings of graphs that are connected.

Function problems: Algorithm needs to compute a function f(x), given input x.

e.g. Factoring problem: Given a natural number, output smallest divisor greater than 1.

P (for polynomial time)

We say that algorithm A is a polynomial time algorithm if there exists constants k, c, such that on all inputs x, A outputs A(x) within time $c\,|x|^k$ (and uses at most $c\,|x|^k$ binary cells of memory).

We say a decision problem L can be decided in polynomial time if there is a polynomial time algorithm A that decides L. We define:

P = {L | L can be decided in polynomial time}.

Theorem: Let A and B be polynomial time algorithms. Let C be an algorithm that is composition of A and B, that is C(x) = A(B(x)); C first run B on input x and then runs A on input B(x). Then C is also polynomial time algorithm.

Proof:

When B runs on input x, it takes time at most $d_1\,|x|^{k1}$ (where d_1 , k1 are some constants).

When A runs on input y, it takes time at most $d_2\,|y|^{k2}$ (where d_2 , k2 are some constants).

Therefore when C runs on input x, it takes time at most $d_1\,|x|^{k1}$ to produce B(x) and then takes time at most $d_2\,(d_1\,|x|^{k1})^{k2}$ (since $|B(x)| \le d_1\,|x|^{k1}$) to produce A(B(x)). Hence total time taken by C is at most $d_3|x|^{k3}$ (where d_3 , k3 are some constants).

P

The issue related to encodings:

Let L_1 = {1^p (unary encoding): p is a prime}; L_1 can be decided in polynomial time. Just check for all number n, between 1 and p, if n divides p.

Let L_2 = {p (binary encoding): p is a prime}; we do not know if L_2 can be decided in polynomial time. The algorithm above takes exponential time.

Theorem: Let L, a set of binary strings, be decided in polynomial time.

Let L_t = {t-ary encoding of x | x in L}. Then L_t can be decided in polynomial time, for any t > 2.

Proof: Let A be a polynomial time algorithm for deciding L. Let B be a polynomial time algorithm which converts any t-ary string to equivalent binary string.

Then define C(x) = A(B(x)). By a previous theorem C is also polynomial time. Also C decides L_t .

Nondeterministic Algorithms

A nondeterministic polynomial time algorithm M.

M(x) {
\\ guess a witness string w_x of length polynomial in |x|
    guess w_x
\\ V is a polynomial (in |x|) time algorithm
    Output $V(x,w_x)$
}

We say that a problem L is decided by M if the following two conditions hold:

1. If x is in L, then there exists a w_x such that $V(x,w_x)$ = true .
2. If x is not in L, then for every string w_x, $V(x,w_x)$ = false.

We say that L can be decided in nondeterministic polynomial time if L can be decided by a nondeterministic polynomial time algorithm M. We define:

NP = { L | L can be decided in nondeterministic polynomial time } .

Since every deterministic polynomial time algorithm is also a nondeterministic polynomial time algorithm (where there are no guesses), it implies P is subset of NP.

Some problems in NP

Graph-Corolaribity = { <G,k> | G can be colored using k colors such that no two adjacent vertices get the same color}

```
Graph-Coloroability (x) {
if x is not a valid encoding of a graph and a number
        return false
else
        let x = <G=(V,E),k>
// guess a color for each vertex
for each v in V
        guess the color c[v]
// check that the coloring is valid
for each v in V
        if (c[v] not in {1,2,…k})
                return false
for each v in V
// N(v) is the neighbors of v
        for each w in N(v)
                if (c[w] == c[v])
                        return false
return true
}
```

Time : O(|V| + |E|), hence nondeterminitic polynomial time.

Some problems in NP

Hamiltonian-Cycle = { < G > | G has a cycle which touches all vertices exactly once}

Hamiltonian-Cycle (<G>) {
// G = (V,E), V= {1,2,…n}
n= |V|
// guess the sequences of vertices that will appear in a Hamiltonian Cycle
guess v_1, v_2, …, v_n
// check only edges in G are used
// check that all vertices are visited
for i = 1 to n
    visited[i] = false
for i = 1 to n
    visited[v_i] = true
    if (v_i, v_{i+1}) is a not an edge in G
        return false
for i = 1 to n
    if (visited[i] = false)
        return false
return true
}

Time : O(|V|), hence nondeterminitic polynomial time.

Some problems in NP

Traveling Salesperson (TSP) = { < G,w > | G is a weighted graph with Hamiltonian cycle of weight at most w}

```
TSP(<G,w>) {
// G=(V,E), V = {1,2,…n}
n= |V|
// guess the sequences of vertices that will appear in a Hamiltonian Cycle
guess v1, v2, …, vn
if v1, v2, …, vn is a not Hamiltonian cycle
      return false
// check that the total weight of the cycle is at most w
t = 0
for i = 1 to n-1
      t = t + weight(vi , vi+1)
if (t > w)
      return false
return true
}
```

Time : O(|V|), hence nondeterminitic polynomial time.

Exponential time

We say that algorithm A is an exponential time algorithm if there exists constants k, c, such that on all inputs x, A outputs A(x) within time $c\,2^{|x|^k}$.

We say a decision problem L can be decided in exponential time if there is an exponential time algorithm A that decides L. We define:

EXP = {L | L can be decided in exponential time}

Theorem : NP is a subset of EXP .

Proof : Any L be in NP. Consider a nondeterministic polynomial time algorithm for L. It is of the form:

```
NPalgoforL(x) {
guess w_x of length c_1|x|^k1
// V is a polynomial time algorithm running in time c_2|x|^k2
return V(x,w_x)
}

EXPalgoforL(x) {
for all w_x of length at most c_1|x|^k1
      if (V(x,w_x) = true)
            return true
return false
}
```

EXPalgoforL(x) runs in time $O(c_2|x|^{k2}\ 2^{c_1|x|^{k1}}) = c_3\ 2^{|x|^{k3}}$. Hence L is in EXP.

Reducibility

We say that a problem A reduces, in polynomial time, to problem B, and we write $A \leq_p B$, if there is a function f that can be computed by an algorithm in polynomial time such that for all strings x,

$$x \in A \Leftrightarrow f(x) \in B.$$

We also say that f is a polynomial time reduction from A to B. We also say that A reduces to B in polynomial time via function f.

Theorem : If $A \leq_p B$ and $B \in P$, then $A \in P$.

Proof: Let f be a polynomial time reduction from A to B computed by algorithm R. Let M be a polynomial time algorithm for B. Consider following polynomial time algorithm for A.

$N(x)$ {
$y = R(x)$
retrun $M(y)$
}

Hence $A \in P$.

NP Completeness

Theorem : If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.

Proof: Let $A \leq_p B$ via polynomial time reduction f and $B \leq_p C$ via polynomial time reduction g. Then

$$x \in A \Leftrightarrow f(x) \in B \Leftrightarrow g(f(x)) \in C.$$

Define $h(x) = g(f(x))$. Then from a previous theorem, h can be computed in polynomial time. Hence $A \leq_p C$ via h.

Definition: We say that a problem B in NP is NP-complete if for all NP problems A, we have $A \leq_p B$.

Theorem : If A and B are problem in NP, A is NP-complete and $A \leq_p B$, then B is NP-complete.

Proof: Let C be any problem in NP. Then $C \leq_p A$ (since A is NP-complete) and we are given $A \leq_p B$. Hence from previous theorem $C \leq_p B$. Therefore B is NP-complete.

Theorem : If any NP complete problem is in P, then $NP = P$.

Proof: Let B be an NP-complete problem that is in P. Let A be any problem in NP. Then $A \leq_p B$. By a previous Theorem, $A \in P$. Hence $NP \subseteq P$. We know $P \subseteq NP$. Hence $P = NP$.

Boolean formula

A boolean formula is an expression involving boolean variables $x_1, x_2, \ldots$ and boolean operators: conjunction (AND) ($\wedge$), disjunction (OR) ($\vee$) and negation (NOT) ($^-$).

A formula that is true for some truth assignment (true/false) to its variables is called 'satisfiable'. A formula that is false for all assignments of truth values to its variable is called 'unsatisfiable'.

Example : $(x_1 \vee \bar{x}_1)$ is satisfiable.

Example : $(x_1 \wedge \bar{x}_1)$ is unsatisfiable.

A boolen formula in CNF (conjuntive normal form) is 'AND of ORs'.

Example : $(x_1 \vee \bar{x}_5) \wedge (x_4 \vee \bar{x}_8 \vee \bar{x}_1)$

$(x_1 \vee \bar{x}_5)$ and $(x_4 \vee \bar{x}_8 \vee \bar{x}_1)$ are called clauses of this CNF formula.

$x_1, \bar{x}_5, \bar{x}_8, x_4$ are called literals. x_1, x_4 are positive literals and $\bar{x}_5, \bar{x}_8$ are negative literals.

We say a formula is k-CNF if every clause contains exactly k literals.

A boolen formula in DNF (disjunctive normal form) is 'OR of ANDs'.

Example : $(x_3 \wedge \bar{x}_6 \wedge x_8) \vee (x_4 \wedge \bar{x}_2 \wedge x_3)$

Cook-Levin Theorem – First NP-complete problem

Cook-Levin Theorem:
SAT $= \{< \phi > \mid < \phi >$ is satisfiable CNF formula $\}$
is NP-complete.

We will not get into the proof of this (beyond the scope of this course) but use this as a starting point to show several other problems as NP-complete.

3-SAT is NP-complete

Theorem : 3SAT $= \{< \phi > \mid \phi$ is a satisfiable 3-CNF formula$\}$ is NP-complete.

Proof: Reduce SAT to 3-SAT. We are given CNF formula ϕ. We will convert ϕ to 3-CNF formula ϕ' such that : ϕ is satisfiable if and only if ϕ' is satisfiable. In other words

$$< \phi > \in \text{SAT} \Leftrightarrow < \phi' > \in \text{3SAT}.$$

Reduce SAT to 3-SAT

Replace clause with single literal (l) by

$$(l \vee y_1 \vee y_2) \wedge (l \vee \bar{y}_1 \vee y_2) \wedge (l \vee y_1 \vee \bar{y}_2) \wedge (l \vee \bar{y}_1 \vee \bar{y}_2)$$

Replace clause with two literals $(l_1 \vee l_2)$ by

$$(l_1 \vee l_2 \vee y_1) \wedge (l_1 \vee l_2 \vee \bar{y}_1)$$

Replace clause with k literals $(l_1 \vee l_2 \vee \ldots \vee l_k)$ by

$$\begin{aligned}
&(l_1 \vee l_2 \vee \bar{y}_1) \wedge \\
&(y_1 \vee l_3 \vee \bar{y}_2) \wedge \\
&\quad \vdots \\
&(y_{i-2} \vee l_i \vee \bar{y}_{i-1}) \wedge \\
&\quad \vdots \\
&(y_{k-4} \vee l_{k-2} \vee \bar{y}_{k-3}) \wedge \\
&(y_{k-3} \vee l_{k-1} \vee l_k)
\end{aligned}$$

Arguments for correctness of reduction done in class.
This reduction can be done in polynomial time (in the length of the original formula ϕ) to get new 3-CNF formula ϕ'.

Independent set is NP-complete

An independent set in a graph is a set of vertices such that there is no edge between any two of them.

Theorem : Independent-Set $= \{< G, k > \mid G$ contains an independent set of size $k\}$ is NP-complete.

Proof: Reduce 3-SAT to Independent-Set. Given a 3-CNF formula ϕ with m clauses, we will produce a graph G with $3m$ vertices. G will have independent set of size m if and only if ϕ is satisfiable. In other words

$$< \phi > \in \text{3SAT} \Leftrightarrow < G, m > \in \text{Independent-Set}.$$

Proof done in class.

Clique is NP-complete

A clique in a graph is a set of vertices such that there is an edge between every two of them.

Theorem : Clique $= \{< G, k >$ | G contains a clique of size $k\}$ is NP-complete.

Proof: Reduce Independent-Set to Clique. Easy to see that ($\bar{G}$ is complement of G):

$$< G, k > \in \text{Independent-Set} \Leftrightarrow < \bar{G}, k > \in \text{Clique}.$$

Graph 3-colorability is NP-complete

Theorem :
Graph-3-colorability $= \{< G > \mid G$ can be colored using 3-colors $\}$
is NP-complete.

Proof: Reduce 3-SAT to Graph-3-colorability. Proof done in class.

A few other NP-complete problems

$$\text{Partition} = \{< (s_1, s_2, \ldots, s_n) > \mid \text{There exists } I \subseteq \{1, 2, \ldots, n\} \text{ such that } \sum_{i \in I} s_i = \sum_{j \notin I} s_j \}.$$

$$\text{Subset-sum} = \{< (s_1, s_2, \ldots, s_n, k) > \mid \text{There exists } I \subseteq \{1, 2, \ldots, n\} \text{ such that } \sum_{i \in I} s_i = k \}.$$

Hamiltonian-Cycle = { < G > | G has a cycle which touches all vertices exactly once}

Traveling Salesperson (TSP) = { < G,w > | G is a weighted graph with Hamiltonian cycle of weight at most w}

Many other real world problems arising from Networks, Packing, Scheduling, Graphs, Cryptography, Games, Computational Biology etc. are NP-complete. Very little hope for finding polynomial time algorithms for these!

Algorithms for NP-complete problems

Approximation Algorithms

Bin Packing

Input : Sizes s[1], s[2], …, s[n] ϵ (0,1].
Output: Minimum number of bins (each of capacity 1) in which these objects can be packed.

```
next-fit(s) {
n = s.last
k = 1 // current bin
c[k] = 0 // capacity filled till now in current bin
for i = 1 to n
      if (c[k] + s[i] ≤ 1 ) {
            b[i] = k
            c[k] = c[k]+ s[i]
      }
      else {
      k = k + 1
      b[i] = k
      c[k] = s[i]
      }
return k
}
```

Theorem: $next - fit(s) \leq 2opt(s)$

$$
\begin{aligned}
opt(s) &\geq \sum_{i=1}^{n} s[i] \\
&\geq \sum_{k=1}^{next-fit(s)} c[k] \\
&\geq \frac{next - fit(s)}{2}
\end{aligned}
$$

Wigderson Coloring

Theorem : A 3-colorable graph can be colored in polynomial time using at most O(√n) colors.

Input : A 3-colrable graph G = (V,E)
Ouput : A valid coloring using O(√n) colors.

```
Wigderson-coloring(G) {
n = |V|
c= 0  // c is color count
while (V contains a vertex of degree at least √n) {
      pick v of degree at least √n
      G' = (N(v), E)
      two-color(G', c) // this colors G' using two colors c +1 and c +2
      c= c +2 // move to the next set of colors
      G = G – N(v)
      }
S = {c, c+1, …. , n}
for each v in V
      color v with smallest color in S not used by any vertex in N(v)
}
```

Brute Force

Largest Independent Set

Input : Graph G=(V,E)
Output: None

```
Largest-independent-set(G) {
if (E is empty)
    return |V|
else {
    pick first v in V such that N(v) is not empty
    G1 = G – {v}
    G2 = G – {v} – N(v)
    k1 = largest-independent-set(G1) // assume v is not in independent set
    k2 = largest-independent-set(G2) // assume v is in independent set
    return max(k1, k2 +1)
    }
}
```

Let a_n be the running time. Then

$a_n = a_{n-1} + a_{n-2} + cn^2$

Solving this recursion gives
$a_n = O(1.62^n)$

Brute Force

3-SAT

Input Parameter : 3-CNF formula ϕ

Ouput : None

```
3-satisfiability(ϕ) {
    if (ϕ does not contain any clauses)
        return ϕ // ϕ has to be logical constant either true of false
    if (ϕ contains a clause with one literal a) {
        ϕ1 = ϕ[a->true] // a has to be true
        return 3-satisfaibility(ϕ1)
    }
    if (ϕ contains a clause with two literals a and b) {
        ϕ1 = ϕ[a->true]
        ϕ2 = ϕ[a->false][b->true]
        return 3-satisfaibility(ϕ1) ) || 3-satisfaibility(ϕ2)
    }
    if (ϕ contains a clause with three literals a, b and c){
        ϕ1 = ϕ[a->true]
        ϕ2 = ϕ[a->false][b->true]
        ϕ3 = ϕ[a->false][b->false][c->true]
        return 3-satisfaibility(ϕ1)
        || 3-satisfaibility(ϕ2)
        || 3-satisfaibility(ϕ3)
        }
}
```

Let a_n be the number of recursive calls made by the algorithm on input formula with n variables. Then

$a_n = a_{n-1} + a_{n-2} + a_{n-3}$

Solving this recursion gives $a_n = O(1.84^n)$. In each recursion the time taken is $O(|\phi|)$ and hence total time taken is $O(|\phi|\ 1.84^n)$.

The best time known till now is $O(|\phi|\ 1.77^n)$ due to Dantsin, Goerdt, Hirsch and Schoning.