

Design and Analysis of Algorithms

4th Semester, CST Department
Theory Paper

Complexity – asymptotic bounds

- An approximate measure of the time taken for the program to run
- Stated as a function $f(n)$ of the problem size
- Problem size for sorting – size of the array
- Factorization of one integer – its magnitude
- Inequality in terms of known function $g(n)$

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n > n_0$$

Complexity – asymptotic bounds

- Tight bound Big theta $f(n) = \Theta(g(n))$
$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$
- Tight upper bound Big-O $f(n) = O(g(n))$
$$f(n) \leq c_2 g(n)$$
- Tight lower bound Big Omega $f(n) = \Omega(g(n))$
$$c_1 g(n) \leq f(n)$$
- Upper bound small-o $f(n) = o(g(n))$
$$f(n) < c_2 g(n)$$
- Lower bound small-omega $f(n) = \omega(g(n))$
$$c_1 g(n) < f(n)$$

Outline Syllabus – different paradigms

- Divide and conquer approach
 - Sorting algorithms
- Dynamic programming approach
 - Matrix multiplication
- Greedy strategy
 - Minimum spanning tree in graph
- Backtracking strategy
 - Depth first and breadth first search in graph

Outline syllabus - some data structures and operations

- Operation on Disjoint sets
 - Shortest path in graph
- Operation on dynamic sets
 - Hashing
- Operation on Polynomials
 - FFT algorithm
- Operations in number theory
 - RSA algorithm

Outline syllabus – advanced topics

- Complexity theory
- Randomized algorithms
- Parallel algorithms
- NP completeness
- Approximation algorithms

Divide and conquer strategy

- DIVIDE into subproblems
- CONQUER the subproblems solving recursively
- COMBINE the solutions of subproblems

Consider a subproblems, each of size $(1/b)$

Time to divide = $D(n)$, Time to combine = $C(n)$

$$T(n) = a T(n/b) + D(n) + C(n)$$

Analogy in Merge Sort

- DIVIDE: n -element sequence into $n/2$ elements in two subsequences
- CONQUER: sort the two subsequences recursively
- COMBINE: MERGE the two subsequences

$$D(n) = \Theta(1), C(n) = \Theta(n), a=2 \text{ and } b=2$$

$$T(n) = 2 T(n/2) + \Theta(n) \text{ giving } T(n) = \Theta(n \lg n)$$

MERGE SORT algorithm

```
MERGE_SORT (A,p,r)
  if (p<r)
    q = floor((p+r)/2)
    MERGE_SORT(A,p,q)
    MERGE_SORT(A,q+1,r)
    MERGE(A,p,q,r)
```

Initial sequence: (5);(7);(6);(4);(1);(3);(2);(8)

Step 1 of conquer: (5,7) ; (4,6); (1,3); (2,8)

Step 2 of conquer: (4,5,6,7) ; (1,2,3,8)

Step 3 of conquer: (1,2,3,4,5,6,7,8)

Quick Sort Algorithm

```
QUICK_SORT(A,p,r)
    if (p<r)
        q=PARTITION(A,p,r)
        QUICK_SORT(A,p,q)
        QUICK_SORT(A,q+1,r)
PARTITION(A,p,r)
    x=A[p]
    i=p-1, j=r+1
    while TRUE
        repeat j=j-1 until A[j] ≤ x
        repeat i=i+1 until A[i] ≥ x
    If i<j
        exchange A[i], A[j]
    Else
        return (j)
```

Solving recurrence

- Substitution method
 - Guess a bound and then use mathematical induction to prove the guess correct
- Iteration method
 - Convert the recurrence into a summation and then bound the summation
- Master method
 - Provides bound for all recurrences of the form $T(n) = a T(n/b) + f(n)$

Substitution method

- $T(n) = 2 T(n/2) + n \rightarrow$ Guess $T(n) = O(n \lg n)$
- Start by assuming this bound holds for $T(n/2)$
- Hence $T(n/2) \leq c (n/2) \lg (n/2)$
- Boundary condition can be shown to hold for $n \geq n_0$
- *Difficult for $n=1$ but holds for $n=2$ onwards with $c \geq 2$*
- Substituting into the recurrence yields
 - $$\begin{aligned} T(n) &\leq 2(c (n/2) \lg (n/2)) + n \\ &\leq c n \lg (n/2) + n \\ &= cn \lg n - cn \lg 2 + n = cn \lg n - cn + n \\ &\leq c n \lg n \text{ (proved by induction)} \end{aligned}$$

Substitution method

- $T(n) = 2 T(\sqrt{n}) + \lg n$
- Renaming $m = \lg n$, $T(2^m) = 2 T(2^{m/2}) + m$
- Next rename $T(2^m) = S(m)$ to get
 - $S(m) = 2 S(m/2) + m$
- $S(m) = O(m \lg m)$ *alike the earlier example.*
- Hence $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$

Iteration method

$$\begin{aligned}T(n) &= 3 T(n/4) + n \text{ ----- now iterate steps of this} \\&= n + 3 ((n/4) + 3 T(n/16)) \\&= n + 3 ((n/4) + 3 (n/16) + 3T(n/64)) \\&= n + 3(n/4) + 9(n/16) + 27 T(n/64)\end{aligned}$$

So i-th term is $3^i (n/4^i)$

Last term is $3^{(\log_4 n)} \Theta(1) = \Theta(n^{(\log_4 3)})$

Iteration hits $n=1$ when $(n/4^i) = 1$ or $i > \log_4 n$

Hence $T(n)$ is a geometric series of $(3/4)^i$

This yields $T(n) \leq 4n + o(n) = O(n)$

Recursion tree

Convenient way to visualize the recursions

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

At the topmost level, excess work is n^2

The next level will need $(n/2)^2 + (n/2)^2 = n^2/2$

The next level will need $4 (n/4)^2 = n^2/4$

This continues to $\lg n$ levels, total work $O(n^2)$

Recursion tree

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

Here at every level, work amount is n .

So it is important to find the height of this tree

Longest path from root to leaf is $(2/3)^k n = 1$

So the height of the tree is $k = \log_{3/2} n$

Solution of the recurrence is at most $n \log_{3/2} n$

This can be considered as $O(n \lg n)$

Master theorem

Theorem *Let a be an integer greater than or equal to 1 and b be a real number greater than 1. Let c be a positive real number and d a nonnegative real number. Given a recurrence of the form*

$$T(n) = \begin{cases} aT(n/b) + n^c & \text{if } n > 1 \\ d & \text{if } n = 1 \end{cases}$$

then for n a power of b ,

- 1. if $\log_b a < c$, $T(n) = \Theta(n^c)$,*
- 2. if $\log_b a = c$, $T(n) = \Theta(n^c \log n)$,*
- 3. if $\log_b a > c$, $T(n) = \Theta(n^{\log_b a})$.*

Outline of proof

Proof: In this proof, we will set $d = 1$, so that the bottom level of the tree is equally well computed by the recursive step as by the base case. It is straightforward to extend the proof for the case when $d \neq 1$.

Let's think about the recursion tree for this recurrence. There will be $\log_b n$ levels. At each level, the number of subproblems will be multiplied by a , and so the number of subproblems at level i will be a^i . Each subproblem at level i is a problem of size (n/b^i) . A subproblem of size n/b^i requires $(n/b^i)^c$ additional work and since there are a^i problems on level i , the total number of units of work on level i is

$$a^i(n/b^i)^c = n^c \left(\frac{a^i}{b^{ci}} \right) = n^c \left(\frac{a}{b^c} \right)^i.$$

Terminating condition

$$\left(\frac{a}{b^c}\right) = 1$$

$$\Leftrightarrow a = b^c$$

$$\Leftrightarrow \log_b a = c \log_b b$$

$$\Leftrightarrow \log_b a = c$$

In general, we have that the total work done is

$$\sum_{i=0}^{\log_b n} n^c \left(\frac{a}{b^c}\right)^i = n^c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i$$

Proof of first two cases

In case 1,
$$n^c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c} \right)^i = \Theta(n^c).$$

In Case 2 we have that $\frac{a}{b^c} = 1$ and so

$$n^c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c} \right)^i = n^c \sum_{i=0}^{\log_b n} 1^i = n^c (1 + \log_b n) = \Theta(n^c \log n) .$$

Proof of third case

In Case 3, we have that $\frac{a}{b^c} > 1$. So in the series

$$\sum_{i=0}^{\log_b n} n^c \left(\frac{a}{b^c} \right)^i = n^c \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c} \right)^i,$$

the largest term is the last one,

the sum is $\Theta \left(n^c \left(\frac{a}{b^c} \right)^{\log_b n} \right)$

$$\begin{aligned} n^c \left(\frac{a}{b^c} \right)^{\log_b n} &= n^c \frac{a^{\log_b n}}{(b^c)^{\log_b n}} \\ &= n^c \frac{n^{\log_b a}}{n^{\log_b b^c}} \\ &= n^c \frac{n^{\log_b a}}{n^c} \\ &= n^{\log_b a}. \end{aligned}$$

Thus the solution is $\Theta(n^{\log_b a})$.

Generalization

Theorem *Let a and b be positive real numbers with $a \geq 1$ and $b \geq 2$. Let $T(n)$ be defined by*

$$T(n) = \begin{cases} aT(\lceil n/b \rceil) + f(n) & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

Then

1. *if $f(n) = \Theta(n^c)$ where $\log_b a < c$, then $T(n) = \Theta(n^c) = \Theta(f(n))$.*
2. *if $f(n) = \Theta(n^c)$, where $\log_b a = c$, then $T(n) = \Theta(n^{\log_b a} \log_b n)$*
3. *if $f(n) = \Theta(n^c)$, where $\log_b a > c$, then $T(n) = \Theta(n^{\log_b a})$*

The same results apply with ceilings replaced by floors.

Complexity and randomization

- Running time of PARTITION on subarray of size n will be $\Theta(n)$
- Worst case: $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$
- Best case: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$

To find average case, we need to randomize.

```
RANDOMIZED_PARTITION(A,p,r)
```

```
    i=RANDOM(p,r)
```

```
    exchange A[p] with A[i]
```

```
    return PARTITION(A,p,r)
```

Average case complexity of Randomized Quick Sort

$$T(n) = (1/n) (T(1)+T(n-1) + \sum_{q=1}^{n-1} T(q) + T(n-q)) + \Theta(n)$$

$$\text{Now, } (1/n)(T(1)+T(n-1)) = (1/n)(O(1)+O(n^2)) = \Theta(n)$$

Hence, due to symmetry wrt q ,

$$T(n) = (2/n) \sum_{k=1}^{n-1} T(k) + \Theta(n)$$

Assume inductively that $T(n) \leq a n \lg n + b$

Then $\sum k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$ which is bounded

This establishes the $\Omega(n \lg n)$ bound.

Lower bound for sorting algorithms that employ comparisons

Result: Any decision tree that sorts n elements
has a height of $h = \Omega(n \lg n)$

Proof:

There can be $n!$ permutations of n elements.

Hence there are $n!$ leaves of the decision tree

For binary tree, $n! \leq 2^h$ or, $h \geq \lg(n!)$

Use Stirling approximation, $n! > (n/e)^n$

To get $h > n \lg n$ which leads to the result.

Counting Sort and Radix Sort

COUNTING-SORT (A,B,k)

for i=1 to k

 C[i]=0

for j=1 to length[A]

 C[A[j]]=C[A[j]]+1

for i=2 to k

 C[i]=C[i]+C[i-1]

for j=length[A] downto 1

 B[C[A[j]]]=A[j]

 C[A[j]]=C[A[j]]-1

When $k=O(n)$, $T(n)=O(n)$ – linear

Radix-Sort(A,d)

for i=1 to d

 use stable counting sort
 on digits from right to left, keeping
 the other digits intact

- A = [3¹ 6 4¹ 1¹ 3² 4² 1² 4³] (array to be sorted)
- C = [2 0 2 3 0 1] (count)
- C = [2 2 4 7 7 8] (cumulative)
- B = [- - - - -] (initial)
- B = [- - - - - 4³ -] C = [2 2 4 6 7 8]
- B = [- 1² - - - - 4³ -] C = [1 2 4 6 7 8]
- B = [- 1² - - - 4² 4³ -] C = [1 2 4 5 7 8]
- B = [- 1² - 3² - 4² 4³ -] C = [1 2 3 5 7 8]
- B = [1¹ 1² - 3² - 4² 4³ -] C = [0 2 3 5 7 8]
- B = [1¹ 1² - 3² 4¹ 4² 4³ -] C = [0 2 3 4 7 8]
- B = [1¹ 1² - 3² 4¹ 4² 4³ 6] C = [0 2 3 4 7 7]
- B = [1¹ 1² 3¹ 3² 4¹ 4² 4³ 6] C = [0 2 2 4 7 7]

B holds the sorted output

Linear time sorting – Bucket sort

BUCKET-SORT (A)

$n = \text{length}(A)$

 for $i = 1$ to n

 insert $A[i]$ into list $B[\text{floor}(n A[i])]$

 for $i = 0$ to $n-1$

 sort list $B[i]$ using insertion sort

Concatenate lists $B[0], B[1], \dots, B[n-1]$ together

Bucket sort – time complexity

- Here n_i is a random variable denoting $|B[i]|$, the size of the i^{th} bucket.
- Insertion sort runs in quadratic time, so that expected time to sort is $\sum O(E(n_i^2))$ summed over all n buckets i.e. $i = 0$ to $n-1$.
- Now probability that $n_i = k$ follows binomial distribution with $p = 1/n$ since there are n elements and n buckets.
- Hence $E[n_i] = np = 1$ and $\text{Var}[n_i] = np(1-p) = 1 - 1/n$
- So, $E[n_i^2] = 2 - 1/n = \Theta(1)$.
- Hence total expected time for bucket sort is $O(n)$

Medians and order statistics

```
RANDOMIZED-SELECT(A,p,r,i)
  if p==r return(A[p])
  q=RANDOMIZED-PARTITION(A,p,r)
  k=q-p+1
  if  $i \leq k$ 
    return RANDOMIZED-SELECT(A,p,q,i)
  else
    return RANDOMIZED-SELECT(A,q+1,r,i-k)
```

```
RANDOMIZED_PARTITION(A,p,r)
  i=RANDOM(p,r)
  exchange A[p] with A[i]
  return PARTITION(A,p,r)
```

Time Complexity of Selection

To find an upper bound $T(n)$ on expected time

$$\begin{aligned} T(n) &\leq 1/n (T(\max(1, n-1)) + \sum T(\max(k, n-k))) + O(n) \\ &\leq 1/n (T(n-1) + 2 \sum T(k)) + O(n) \end{aligned}$$

[$\max(k, n-k) = k$ or $n-k$ split at $k=n/2$]

In worst case, $T(n-1) = O(n^2)$ so that $T(n-1)/n = O(n)$

To solve the recurrence- Substituting $T(n) \leq cn$, and noting that the sum over $T(k)$ runs from $n/2$ to n ;

$$T(n) \leq 2c/n \left(\frac{1}{2} n(n-1) - \frac{1}{2} (n/2) ((n/2)-1) \right) + O(n)$$

$$T(n) \leq c(3/4 n - 1/2) + O(n) \leq cn \text{ picking } c \text{ large enough}$$

Worst case linear time selection

1. Divide n elements into $n/5$ groups of 5 elements each, $n \bmod 5$ in last group
 - $O(n)$
2. Find median of each group using insertion sort and taking middlemost element. For even size in last group, take larger of the two medians.
 - $O(n)$ calls of $O(1)$ size set insertion sorts
3. Use SELECT recursively to find median x of these $n/5$ medians
 - $T(n/5)$
4. Partition input array around median of medians x as the pivotal element, no of elements on low side being k
 - $O(n)$
5. Use SELECT recursively to find i -th smallest element on low side if $i \leq k$ or $(i-k)$ th element on high side if $i > k$
 - $T(7n/10 + 6)$ see next page...

Time complexity – median of medians

- At least half of medians found in Step-2 are $\geq x$
- At least half of the groups contribute 3 elements $> x$, except its own group and the last one i.e. at least $3(\frac{1}{2}(n/5)-2)$
- Hence no of elements $> x$ or $< x$ is at least $= 3n/10 - 6$
- Hence SELECT is called recursively in STEP 5 on at most $7n/10 + 6$ elements
- $T(n) = O(1)$ if $n \leq 80$ and $n > 20$
- $T(n) \leq T(n/5) + T(7n/10 + 6) + O(n)$ if $n > 80$
- Assume $T(n) \leq cn$ to show $T(n) \leq 9cn/10 + 7c + O(n) \leq cn$

One example case for SELECT

One iteration on a randomized set of 100 elements from 0 to 99

	12	15	11	2	9	5	0	7	3	21	44	40	1	18	20	32	19	35	37	39
	13	16	14	8	10	26	6	33	4	27	49	46	52	25	51	34	43	56	72	79
Medians	17	23	24	28	29	30	31	36	42	47	50	55	58	60	63	65	66	67	81	83
	22	45	38	53	61	41	62	82	54	48	59	57	71	78	64	80	70	76	85	87
	96	95	94	86	89	69	68	97	73	92	74	88	99	84	75	90	77	93	98	91

Dynamic Programming

- Characterize structure of optimal solution
- Recursively define value of optimal solution
- Compute value of optimal solution bottom-up
- Construct an optimal solution from this

Problems that need such solution:

- Matrix chain multiplication
- Optimal polygon triangulation
- Longest common subsequence

Key ingredients to look for:

- Optimal substructures
- Overlapping subproblems

Matrix multiplication problem

- Multiply $A_1 \times A_2 \times A_3$
- Dimensions: $(p \times q) \times (q \times r) \times (r \times s)$
- Count of Operations:
 - $(A_1 \times A_2) \times A_3 \rightarrow \text{option-1} = pqr + prs$
 - $A_1 \times (A_2 \times A_3) \rightarrow \text{option-2} = qrs + pqs$

The order of the matrices in a matrix chain is such that the multiplication is compatible.

Polygon triangulation Problem

- **Cost of triangulation** is considered to be the sum of side lengths of the triangles
- **Substructures** - The polygon can be split into two subpolygons by joining two vertices
- **Overlapping** - A pentagon can be split into triangle and quadrilateral and the resulting quadrilateral splits further into two triangles, which two can as well be the starting point

Matrix Chain Multiplication Problem

- Let no of alternative parenthesizations be denoted by $P(n)$
- A sequence of n matrices can be split between k -th and $(k+1)$ -th matrices for any $k=1,2,..,n-1$ and then parenthesize the two resulting subsequences independently.
- $P(n) = \sum P(k) P(n-k)$ if $n \geq 2$ with $P(1)=1$
- Solution is $C(n) = (1/n+1) {}^{2n}C_n = (4^n/n^{3/2})$,
- This is called Catalan number and is exponential.
- Hence no of solutions is exponential and exhaustive search is a poor strategy

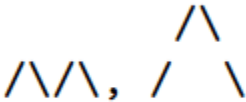
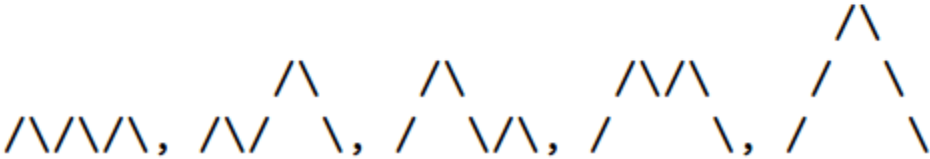
Catalan number

- Balanced parentheses
- Mountain ranges
- Diagonal avoiding paths
- Polygon triangulation
- Hands across a table
- Rooted Binary trees
- Planar trees
- Skewed polyominoes
- Multiplication ordering

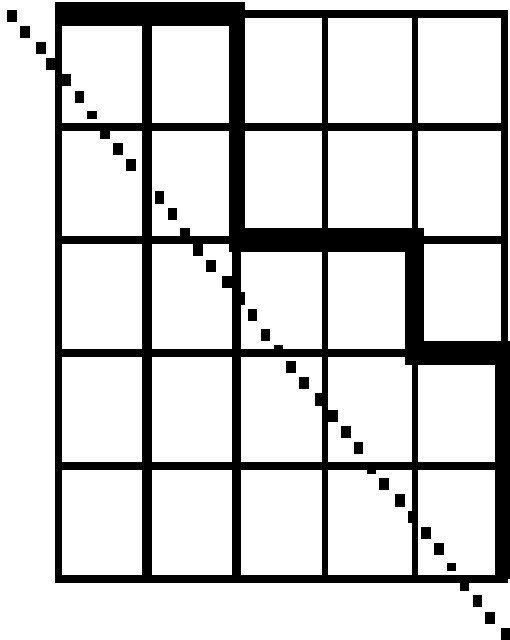
Balanced parentheses

$n = 0$:	*	1 way
$n = 1$:	()	1 way
$n = 2$:	()(), (())	2 ways
$n = 3$:	()()(), ()(()), (())(), (())(), ((()))	5 ways
$n = 4$:	()()()(), ()()(()), ()(())(), ()(())(), ()(()), (())()(), (())(), (())()(), ((())()), (())(), (())(), ((())()), ((())()), ((()))	14 ways
$n = 5$:	()()()()(), ()()()(), ()()()(), ()()()(), ()()(()), ()()()(), ()()()(), ()()()(), ()(()), ()()()(), ()()(), ()(()), ()(()), ()(()), ((())), (())()(), (())()(), (())()(), (())(), (())()(), (())()(), ((())(), ((())(), (())()(), (())()(), ((())(), ((())(), ((())(), (())()(), (())(), (())()(), (())()(), (())(()), ((())(), ((())(), ((())(), ((())(), ((())(), ((())(), ((())(), ((())(), ((())(), ((()))	42 ways

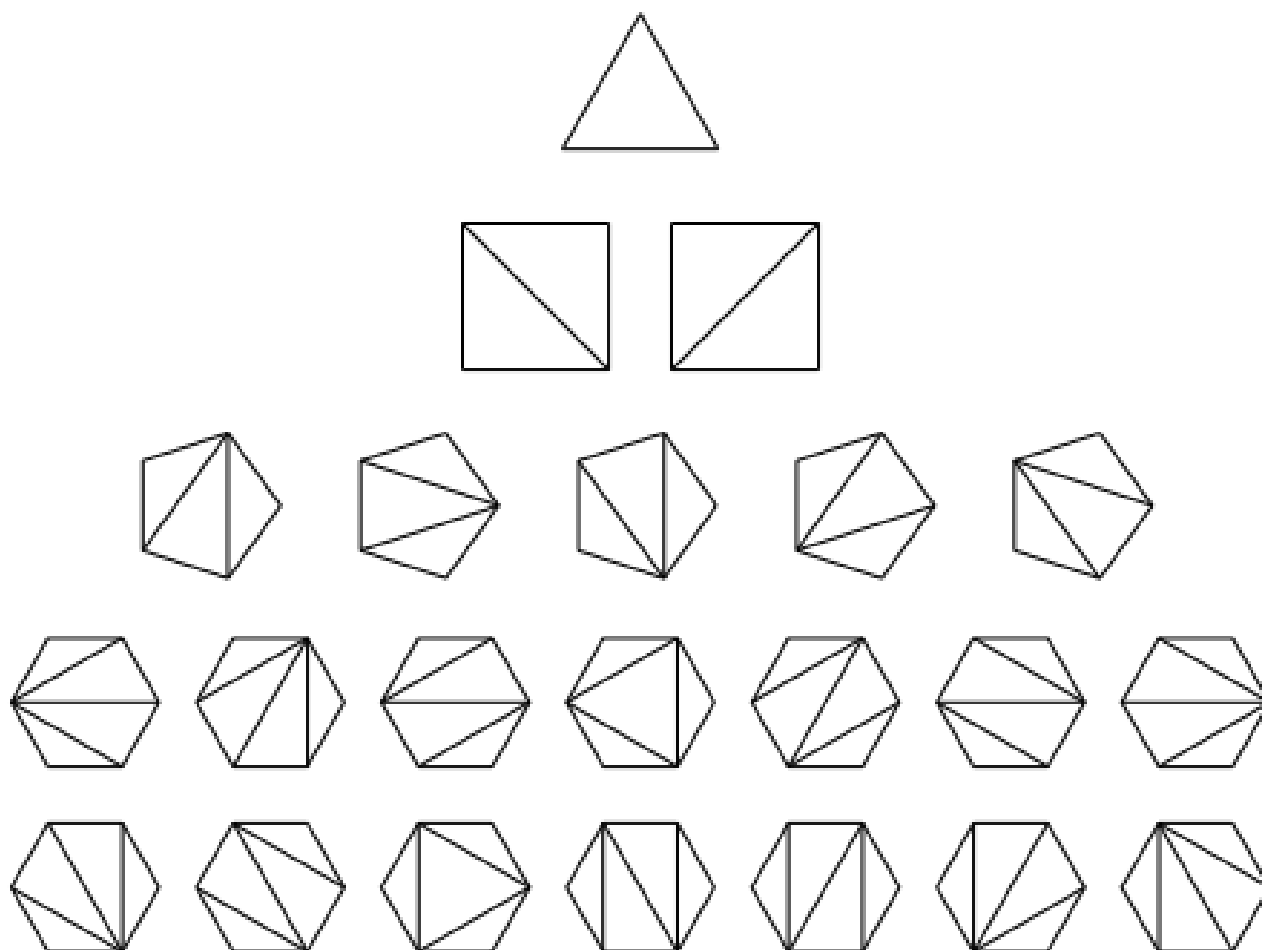
Mountain ranges

$n = 0$:	*	1 way
$n = 1$:	/\	1 way
$n = 2$:		2 ways
$n = 3$:		5 ways

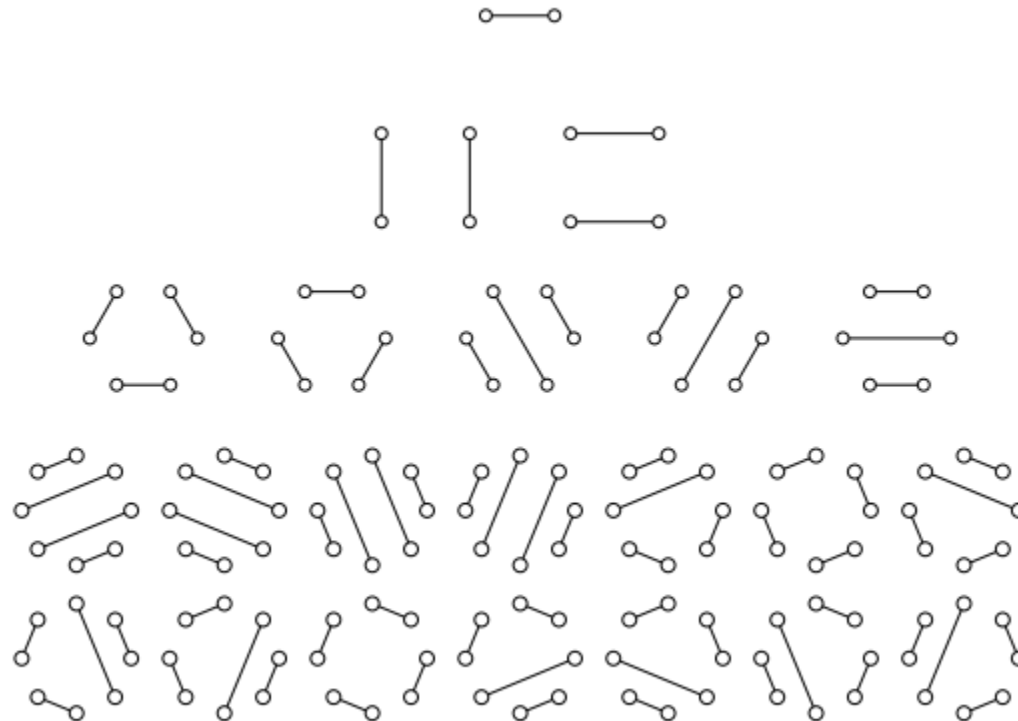
Diagonal avoiding paths



Polygon triangulation

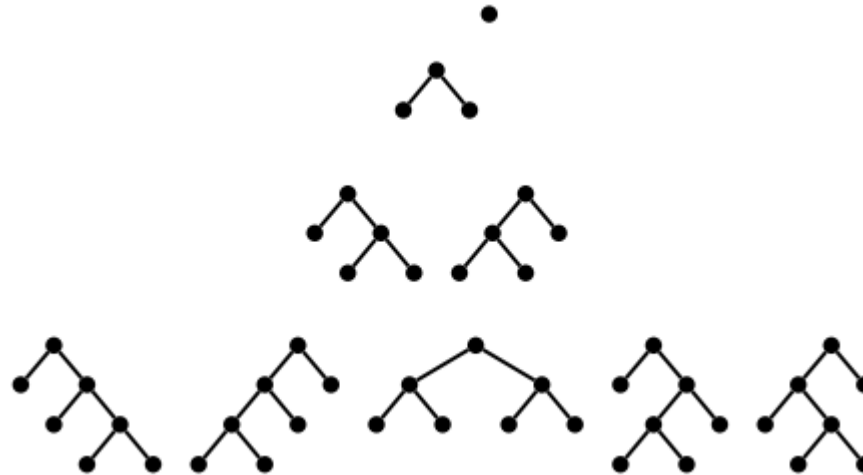


Hands across a table





Rooted binary trees


(internal nodes – those which connect to two nodes below)



Plane rooted trees



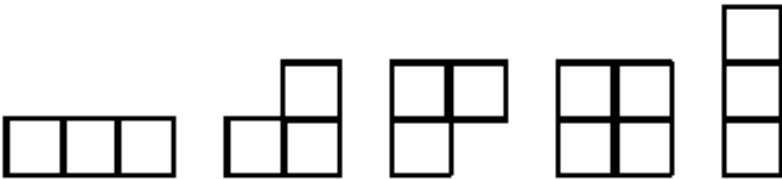
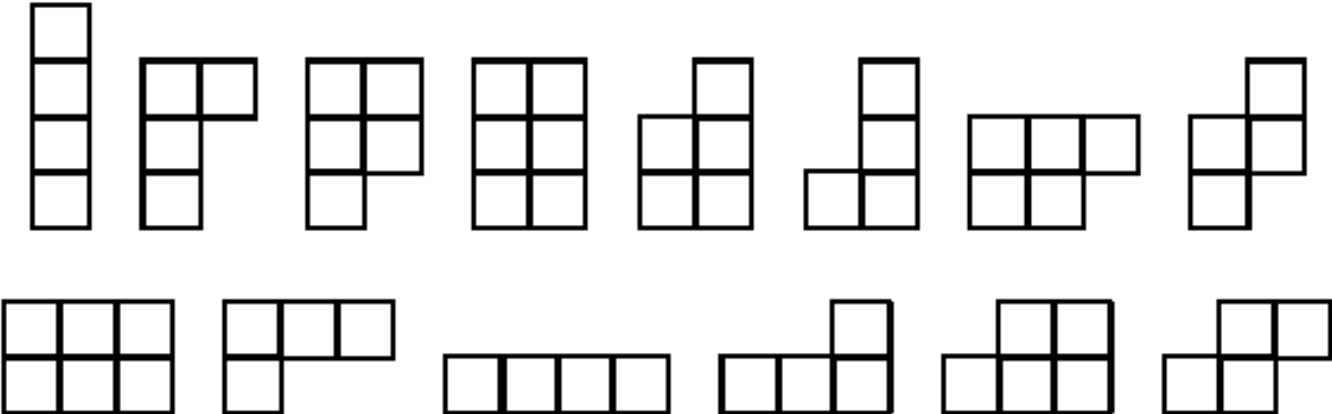
0 Edges: 

1 Edge: 

2 Edges: 

3 Edges: 

Skew polyominos (with perimeter $2n+2$)

$n = 1$	
$n = 2$	
$n = 3$	
$n = 4$	

Matrix chain multiplication

$n = 0$	(a)	1 way
$n = 1$	$(a \cdot b)$	1 way
$n = 2$	$((a \cdot b) \cdot c), (a \cdot (b \cdot c))$	2 ways
$n = 3$	$((((a \cdot b) \cdot c) \cdot d), ((a \cdot b) \cdot (c \cdot d)), ((a \cdot (b \cdot c)) \cdot d),$ $(a \cdot ((b \cdot c) \cdot d)), (a \cdot (b \cdot (c \cdot d))))$	5 ways
$n = 4$	$(((((a \cdot b) \cdot c) \cdot d) \cdot e), (((a \cdot b) \cdot c) \cdot (d \cdot e)), (((a \cdot b) \cdot (c \cdot d)) \cdot e),$ $((a \cdot b) \cdot ((c \cdot d) \cdot e)), ((a \cdot b) \cdot (c \cdot (d \cdot e))), (((a \cdot (b \cdot c)) \cdot d) \cdot e),$ $((a \cdot (b \cdot c)) \cdot (d \cdot e)), ((a \cdot ((b \cdot c) \cdot d)) \cdot e), ((a \cdot (b \cdot (c \cdot d)))) \cdot e),$ $(a \cdot (((b \cdot c) \cdot d) \cdot e)), (a \cdot ((b \cdot c) \cdot (d \cdot e))), (a \cdot ((b \cdot (c \cdot d)) \cdot e)),$ $(a \cdot (b \cdot ((c \cdot d) \cdot e))), (a \cdot (b \cdot (c \cdot (d \cdot e))))$	14 ways

Generating function - Fibonacci

- Consider the generating function to be power series with Fibonacci numbers as coefficients
- $F(z) = \sum F_i z^i$ with $F_i = F_{i-1} + F_{i-2}$
- $F(z) = \sum (F_{i-1} + F_{i-2}) z^i$
- $= z + z \sum F_{i-1} z^{i-1} + z^2 \sum F_{i-2} z^{i-2}$
- $= z + z F(z) + z^2 F(z)$
- $= z / (1 - z - z^2)$

Generating function – Catalan number

- $C(n) = \sum C(k)C(n-k)$ summed over all $k=1$ to $n-1$
- $C(n) = C_{n-1}C_0 + C_{n-2}C_1 + \dots + C_1C_{n-2} + C_0C_{n-1}$
- Using generating function, $f(z) = \sum C(n) z^n$
- Next, $[f(z)]^2 = C_0C_0 + (C_1C_0 + C_0C_1)z + \dots = C_1 + C_2z + C_3z^2 + \dots$
- Multiplying by z on both sides, $f(z) = C_0 + z[f(z)]^2$
- This quadratic in $f(z)$ solves to $(1 - \sqrt{1-4z})/2z$

Solution for Catalan number

$$(1 - 4z)^{1/2} = 1 - \frac{\left(\frac{1}{2}\right)}{1}4z + \frac{\left(\frac{1}{2}\right)\left(-\frac{1}{2}\right)}{2 \cdot 1}(4z)^2 - \frac{\left(\frac{1}{2}\right)\left(-\frac{1}{2}\right)\left(-\frac{3}{2}\right)}{3 \cdot 2 \cdot 1}(4z)^3 + \\ \frac{\left(\frac{1}{2}\right)\left(-\frac{1}{2}\right)\left(-\frac{3}{2}\right)\left(-\frac{5}{2}\right)}{4 \cdot 3 \cdot 2 \cdot 1}(4z)^4 - \frac{\left(\frac{1}{2}\right)\left(-\frac{1}{2}\right)\left(-\frac{3}{2}\right)\left(-\frac{5}{2}\right)\left(-\frac{7}{2}\right)}{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}(4z)^5 + \dots$$

We can get rid of many powers of 2 and combine things to obtain:

$$(1 - 4z)^{1/2} = 1 - \frac{1}{1!}2z - \frac{1}{2!}4z^2 - \frac{3 \cdot 1}{3!}8z^3 - \frac{5 \cdot 3 \cdot 1}{4!}16z^4 - \frac{7 \cdot 5 \cdot 3 \cdot 1}{5!}32z^5 - \dots$$

$$f(z) = 1 + \frac{1}{2!}2z + \frac{3 \cdot 1}{3!}4z^2 + \frac{5 \cdot 3 \cdot 1}{4!}8z^3 + \frac{7 \cdot 5 \cdot 3 \cdot 1}{5!}16z^4 + \dots$$

Solution for Catalan number

The terms that look like $7 \cdot 5 \cdot 3 \cdot 1$ are a bit troublesome. They are like factorials, except they are missing the even numbers. But notice that $2^2 \cdot 2! = 4 \cdot 2$, that $2^3 \cdot 3! = 6 \cdot 4 \cdot 2$, that $2^4 \cdot 4! = 8 \cdot 6 \cdot 4 \cdot 2$, et cetera. Thus $(7 \cdot 5 \cdot 3 \cdot 1) \cdot 2^4 4! = 8!$. If we apply this idea to Equation we can obtain:

$$f(z) = 1 + \frac{1}{2} \left(\frac{2!}{1!1!} \right) z + \frac{1}{3} \left(\frac{4!}{2!2!} \right) z^2 + \frac{1}{4} \left(\frac{6!}{3!3!} \right) z^3 + \frac{1}{5} \left(\frac{8!}{4!4!} \right) z^4 + \dots = \sum_{i=0}^{\infty} \frac{1}{i+1} \binom{2i}{i} z^i.$$

From this we can conclude that the i^{th} Catalan number is given by the formula

$$C_i = \frac{1}{i+1} \binom{2i}{i}.$$

Solution for Catalan number –use Stirling approximation on factorials

- Numerator = $(2n/e)^{2n} \sqrt{2\pi 2n}$
- Denominator = $(n) [(n/e)^n \sqrt{2\pi n}]^2$
- Overall expression will become
 - $(1/\sqrt{\pi}) 4^n / n^{3/2}$
 - **This implies that Catalan number grows exponentially**

Matrix Chain Order

MATRIX-CHAIN-ORDER (p)

n=length[p]-1

for i=1 to n

 m[i,i]=0

for l=2 to n

 for i=1 to n-l+1

 j=i+l-1

 m[i,j]=INF

 for k=i to j-1

 q=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j

 if q< m[i,j]

 m[i,j]=q

 s[i,j]=k

Return m and s

Multiplication of the chain

Matrix-Chain-Multiply (A,s,i,j)

if $j > i$

$X = \text{M-C-M}(A, s, i, s[i, j])$

$Y = \text{M-C-M}(A, s, s[i, j] + 1, j)$

return MATRIX-MULTIPLY(X, Y)

else

return A_i

Memoized Matrix Chain

MMC(p)

 n=length[p]-1

 for i= 1 to n

 for j= i to n

 m[i,j]=INF

 return LOOK-UP-CHAIN(p,1,n)

LookUpChain(p,i,j)

 if m[i,j]<INF return m[i,j]

 if i==j

 m[i,j]=0

 else

 for k=i to j-1

 q=LUC(p,i,k)+ LUC(p,k+1,j)+ $p_{i-1}p_kp_j$

 if q < m[i,j]

 m[i,j]=q

Return m[i,j]

Example of Matrix Chain

$m[i,j] = \min\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}$ for all $i \leq k < j$

Suppose, the best parentheses of 2-5 is needed.

$$m[2,5] = m[2,2] + m[3,5] + p_1p_2p_5$$

$$m[2,5] = m[2,3] + m[4,5] + p_1p_3p_5$$

$$m[2,5] = m[2,4] + m[5,5] + p_1p_4p_5$$

Here, all smaller chain matrix results are stored.

The minimum among the three is stored in $m[][]$

The corresponding index k is stored in $s[][]$

Recursive Matrix Chain

RMC(p,i,j)

if $i=j$ then return zero

$m[i,j]=\text{INF}$

for $k=i$ to $j-1$

$q=\text{RMC}(p,i,k)+\text{RMC}(p,k+1,j)+p_{i-1}p_kp_j$

if $q < m[i,j]$

$m[i,j]=q$

Return $m[i,j]$

$$T(n) \geq 1 + \sum (T(k) + T(n-k) + 1) \geq 2 \sum T(i) + n \geq 2^{n-1}$$

Greedy Algorithm for Activity Selection

- Locally optimum choice leading to globally optimum solution
- Greedy choice property & Optimal substructure
- If A is optimal then $A' = A - \{1\}$ is also optimal solution for the set S' with $s_i \geq f_1$
- Induction at every step - top down approach iteratively solves a smaller subproblem

Greedy Activity Selector

- To accommodate maximum number of activities with set of start times and finish times
- Sort the activities first on finish times in order to maximize the amount of unscheduled time remaining

```
n=length[S]
```

```
A={1}
```

```
j=1
```

```
for i=2 to n
```

```
    if  $s_i \geq f_j$  //Compatibility check
```

```
        A=A U {i}
```

```
        j=i // most recent addition
```

```
return A
```

Knapsack problem

- Thief robs a store having n items with value v_i and weight w_i with total carrying capacity of W .
- Optimal substructure – if a portion or whole of the most valuable is taken out, the remaining load is to be selected from remaining items
- In Fractional knapsack, use top-down greedy strategy on unit price of items $u_i = v_i / w_i$
- In 0/1 knapsack, the strategy fails as there can be empty space left out – use dynamic programming to solve the resulting overlapping subproblems bottom-up

Data compression - Huffman coding

- Variable length encoding based on frequency of occurrence of the symbols
- Collapse two least occurring symbols into compound symbol
- Continue the process until two symbols are left
- Heap based construction yields the coding tree
- Minimum overhead on average code word length ensured by collapsing of least probable symbols
- No other code that uses any other strategy is capable of better compression

Greedy Algorithm on Matroids

- Matroid is an ordered pair $M = [S, I]$
- To find Maximal weight independent subset I of a set S having elements of weight w

Graphic Matroid theory

Generic Matroid

S is a finite non-empty set

Independent: \mathcal{I} is non-empty family of subsets of S

Hereditary: If $B \in \mathcal{I}$ and A is contained in B then $A \in \mathcal{I}$

Exchange: If $A, B \in \mathcal{I}$ and $|A| < |B|$ then some $x \in B - A$ exists such that $A \cup \{x\} \in \mathcal{I}$

Graphic Matroid

Set of edges E of a graph (V, E)

A is \mathcal{I} iff A is acyclic, Set of edges are independent iff it is forest

Deletion of an edge retains the independent property – subset of forest is a forest

Extension to $A \cup \{x\}$ possible till formation of cycle

Maximal Independent Subset

- When no more extensions are possible
- All maximal subsets are of same size
- Add the edge weights to get $w(A) = \sum w(x)$
- Find maximum weight independent subset A of weighted matroid

GREEDY(M, w)

$A = \text{NULL}$

 sort $S[M]$ into non-decreasing order by weight

 for each $x \in S[M]$ taken in order of w

 if $A \cup \{x\} \in I[M]$

$A = A \cup \{x\}$

Return A

Exchange property of graphic matroid

- Suppose A and B are forests of G and that $|B| > |A|$, i.e. A, B are acyclic sets of edges and B contains more edges.
- Now, forest with k edges contains exactly $|V| - k$ trees. Begin with $|V|$ trees and no edges. As and when an edge is introduced, no of trees reduce by 1. So forest A contains $|V| - |A|$ trees and forest B has $|V| - |B|$ trees.
- Since forest B has fewer trees, it must be having some tree T whose vertices are in two different trees in forest A .

Exchange property of graphic matroid

- Now, T being connected, it must have an edge (u,v) connecting vertices in two different trees in forest A . Thus edge (u,v) can be added to A without creating a cycle. This satisfies exchange property.
- An edge e is an extension of A iff e is not in A and addition of e does not create a cycle. When no more extension is possible, A is maximal. For this A should not be contained in any larger independent subset of M .

Size of independent subsets

- **All maximal independent subsets in a matroid have the same size.**
- Suppose on the contrary, B is a maximal independent subset that is larger than another one A . In that case, exchange property implies that A is extendable to $A \cup \{x\}$ for some $x \in B - A$. which contradicts the assumption that A is maximal.

Correctness of greedy algorithm

- **Lemma 1:** Let x be the first element of S (sorted on weight) such that $\{x\}$ is independent. If such x exists, then there exists an optimal subset A of S that contains x . (greedy choice property)
- **Lemma 2:** If x is not an extension of NULL , it is not an extension of any independent subset A of S .
- **Lemma 3:** Let x be first element chosen. The remaining problem becomes $M'=(S',I')$ where $S'=\{y \in S: \{x,y\} \in I\}$ and $I'=\{B \text{ is subset of } S-\{x\}: B \cup \{x\} \in I\}$ i.e. M' is contraction of M by x .

Proof for Lemma 1

- If no x exists, we have empty set – independent.
- Otherwise, B is any non-empty optimal subset. Assume that x does not belong to B . otherwise $A=B$ and we are done. No element of B has weight $> w(x)$. Observe that $y \in B$ implies that $\{y\}$ is independent. Since $B \in I$ and I is hereditary.
- Our choice of x therefore ensures $w(x) \geq w(y)$ for any $y \in B$. Now construct set A as follows. Begin with $A=\{x\}$ which by choice of x makes A independent.
- Using exchange property repeatedly, find a new element of B that can be added to A until $|A|=|B|$ while preserving independence of A .
- Then $A= B-\{y\} \cup \{x\}$ for some $y \in B$ and so we have $w(A)=w(B) - w(y) + w(x) \geq w(B)$. Because B is optimal, A must also be optimal and since $x \in A$, lemma is proven.

Proof for Lemma 2 and 3

- **Proof:** Assume that x is an extension of A but not of NULL . Since x is extension of A , $A \cup \{x\} \in I$. Since I is hereditary, $\{x\} \in I$ which contradicts the assumption.
- **Proof:** If A is any maxwt independent subset of M containing x , then $A' = A - \{x\}$ is an independent subset of M' . Conversely, any independent subset A' of M' yields an independent subset $A = A' \cup \{x\}$ of M . Since in both cases we have $w(A) = w(A') + w(x)$, a maxwt solution in M containing x yields maxwt solution in M' and vice versa.

Proof for matroid greedy theorem

- By Lemma 2, any element that is not an initial extension of NULL, can be forgotten.
- Once first element x is selected, Lemma 1 implies that the greedy algorithm does not err by adding x to A , since there exists an optimal subset containing x .
- Finally Lemma 3 implies that the problem gets reduced to one of finding an optimal subset in M' that is a contraction of M by x .
- After setting $A = \{x\}$, the remaining steps act on $M'=(S',I')$ because B is independent in M' iff $B \cup \{x\}$ is independent in M , for all sets $B \in I'$.
- Thus subsequent operations of greedy algorithm finds maxwt independent subset for M' and overall operation finds a maxwt independent subset for M .

Task scheduling problem

- $S=\{1,2,\dots,n\}$ of n unit-time tasks with deadlines $\{d_1,d_2,\dots,d_n\}$ and penalty (non -ve) $\{w_1,w_2,\dots,w_n\}$ for missing the deadlines. To find a schedule for S that minimizes the total penalty.
- Early-deadline-first schedule is possible by swapping tasks while constructing a schedule starting from NULL set. Then we are left with a subset A of early tasks where tasks meet deadlines sorted in order of non-decreasing deadlines and another subset $\{S-A\}$ of late tasks where tasks miss deadlines and these may appear in any order.

Early and late task sets

- Set A of tasks is independent if there exists a schedule where none is late, the set of early tasks for a schedule forms an independent set of tasks.
- Let I denote the set of all independent sets. For $t=1,2,\dots,n$; let $N_t(A)$ denote no of tasks in A whose $d_i \leq t$ i.e. deadline is t or earlier.
- Clearly, if $N_t(A) > t$ for some t , then there is no way to make a schedule with no late tasks for set A , because there are more than t tasks to finish before time t .
- Hence set A is independent implies that $N_t(A) \leq t$ for $t=1,2,\dots,n$.

Minimizing penalty on early tasks

- Then there is no late task if those in A are scheduled in order of non-decreasing d_i . The i -th largest deadline is at most i . Given these properties it is easy to compute whether a given set of tasks is independent.
- Minimizing sum of penalties on late tasks is equivalent to maximizing the penalty on early tasks. So we can use greedy strategy to find an independent set A of tasks that maximizes the total penalty. Then this system must be shown to be a matroid.

Independent task sets

- Every subset of an independent set of tasks is certainly independent.
- Suppose B, A are independent with $|B| > |A|$.
- Let k be the largest t so that $N_t(B) \leq N_t(A)$.
- Since $N_n(B) = |B|$ and $N_n(A) = |A|$, but $|B| > |A|$,
- we must have $k < n$ and $N_j(B) > N_j(A)$ for all j in the range $k+1 \leq j \leq n$.
- Therefore B contains more tasks with deadline $k+1$ than A does.

Exchange property of tasks

- Let x be a task in $B-A$ with deadline $k+1$. Let $A' = A \cup \{x\}$.
- To show exchange property, we need to show that A' must be independent, using above property.
- For the range $1 \leq t \leq k$, $N_t(A') = N_t(A) \leq t$ since A is independent.
- For the range $k < t \leq n$ we have $N_t(A') \leq N_t(B) \leq t$ since B is independent. Thus A' is independent.

Task scheduling - example

Task	1	2	3	4	5	6	7
Deadline	4	2	4	3	1	4	6
Penalty	70	60	50	40	30	20	10

Augment A	Deadline	$N_1(A)$ ≤ 1	$N_2(A)$ ≤ 2	$N_3(A)$ ≤ 3	$N_4(A)$ ≤ 4	$N_5(A)$ ≤ 5	$N_6(A)$ ≤ 6	Remarks
{1}	4	0	0	0	1	-	-	Independent
{1,2}	4,2	0	1	1	2	-	-	-do-
{1,2,3}	4,2,4	0	1	1	3	-	-	-do-
{1,2,3,4}	4,2,4,3	0	1	2	4	-	-	-do-
{1,2,3,4,5}	4,2,4,3,1	1	2	3	5	-	-	$N_4(A) > 4$
{1,2,3,4,6}	4,2,4,3,4	0	1	2	5	-	-	$N_4(A) > 4$
{1,2,3,4,7}	4,2,4,3,6	0	1	2	4	4	5	Independent

Result of scheduling example

- The set $S=\{1,2,3,4,5,6,7\}$ of tasks is sorted on the penalty.
- Next, we sort A on deadlines so that schedule is early tasks $\langle 2,4,1,3,7 \rangle$ followed by late tasks $\langle 5,6 \rangle$
- Final schedule $\langle 2,4,1,3,7,5,6 \rangle$ with penalty=50

Operations on Dynamic sets

- Hash functions map items from a very large set to a smaller set (storage location)
- Each item has a key (an integer)
- Randomness is expected in the hash function
- Collision occurs when the mapping for a given key returns an occupied slot
- Collision resolution is done through chaining
- Rehashing can be done to avoid collision till finding an unoccupied location
- Symbol tables in assemblers and compilers are implemented as hash tables

Operations on hash table

- Insert, delete and search
- Problem size depends on
 - Number of slots in the hash table, m
 - Number of items stored in the hash table, n
 - Load factor = $\alpha = n/m$
- Chaining needs traversal along the chain if slot is occupied $\Rightarrow \alpha > 1$ is possible
- Open addressing needs to free slot for insertion upon deletion of an item $\Rightarrow \alpha \leq 1$

Search time complexity - Chaining

- Search time can be bounded for both success and failure
- In case of failure:
 - Time to hash the key = $O(1)$
 - Time to find the key = $O(\alpha)$
 - Total time = $O(\alpha+1)$
- In case of success:
 - Let the item be i -th one to be inserted
 - Load factor at that time was $(i-1)/m$
 - Chain traversal needed is of length $(i-1)$
 - Expected no of elements searched out of n elements
 - $= (1/n) \sum (1 + (i-1)/m)$ averaged over no of currently stored
 - $= 1 + (1/nm) (1/2) (n(n-1)) = O(1+\alpha)$

Types of hash functions

- Division method $h(k) = k \bmod m$, m is prime
- Multiplication method $h(k) = \text{floor}(m(kA \bmod 1))$ – A being $\{0,1\}$ $A = (\sqrt{5} - 1)/2 = 0.618$
- Collision probability depends on randomness of h
- Linear probing: $h(k,i) = (h'(k) + i) \bmod m$
 - may cause primary clustering
- Quadratic probing: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
 - may cause secondary clustering
- Double hashing: $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$
- Uniform hashing: unoccupied slots are equally likely to be selected $\Rightarrow h(k,i) = (h_1(k) + i h_2(k) + i^2 h_3(k) + \dots) \bmod m$

Unsuccessful Search-Open addressing

- $P_i = \text{prob}\{\text{exactly } i \text{ probes access occupied slots}\}$ for $i = 0, 1, 2, \dots, n$. $P_i = 0$ for $i > n$
- $Q_i = \text{prob}\{\text{at least } i \text{ probes access occupied slots}\}$ assuming uniform distribution, will be

$$= (n/m)(n-1/m-1)\dots(n-i+1/m-i+1) \leq (n/m)^i \leq \alpha^i$$
- Now, $\sum_{i=0}^{\infty} iP(x=i) = \sum_{i=0}^{\infty} i (\Pr(x \geq i) - \Pr(x \geq i+1))$

$$= \sum_{i=1}^{\infty} (\Pr(x \geq i)) = \sum_{i=1}^{\infty} Q_i$$
- Expected no of probes for unsuccessful search =

$$1 + \sum_{i=1}^{\infty} iP_i \leq 1 + \sum_{i=1}^{\infty} Q_i \leq \sum_{i=0}^{\infty} \alpha^i \leq 1/(1-\alpha)$$
- Expected no of probes = 2 for $\alpha=0.5$ and =10 for $\alpha=0.9$

Successful Search-Open addressing

- When element was inserted as $(i+1)$ -th element, load factor was $\alpha = i/m$
- It took $1/(1-i/m)$ no of unsuccessful probes
- Expected no of probes for successful search is sum over all such i when n slots are occupied

$$\frac{1}{n} \sum \frac{1}{(1-i/m)} = (m/n) \sum \frac{1}{(m-i)} = \frac{1}{\alpha} (H_m - H_{m-n})$$

- $H_i = \sum_{j=1}^i (1/j)$ bounded by $\ln(i) \leq H_i \leq 1 + \ln(i)$
- no of probes $\leq \frac{1}{\alpha} (1 + \ln(m) - \ln(m-n)) = \frac{1}{\alpha} + \frac{1}{\alpha} (\ln \frac{1}{1-\alpha})$
- Expected no of probes ≤ 3.387 , $\alpha=0.5$ and ≤ 3.67 , $\alpha=0.9$
- No of probes not increasing rapidly as table fills up.

Data structures for disjoint sets

- $S = \{S_1, S_2, \dots, S_k\}$ disjoint dynamic sets having representative elements
- `MAKE_SET(x)` creates new set with only one member pointed to by x and x cannot belong to any other set – disjoint.
- `UNION (x,y)` unites two dynamic sets S_x and S_y containing x and y into a new set U , with new representative.
- `FIND_SET (x)` returns pointer to the representative of the set containing x .

Connected Components

```
for each vertex  $v \in V[G]$ 
  do MAKE_SET( $v$ )
for each edge  $(u,v) \in E[G]$ 
  if FIND_SET( $u$ )  $\neq$  FIND_SET( $v$ )
    UNION( $v,u$ )
```

End

```
SAME_COMPONENT( $u,v$ )
  if FIND_SET( $u$ ) == FIND_SET( $v$ )
    then return TRUE
  else return FALSE
```

End

Parameters for analyzing running times of operations

- The no of MAKE_SET operations is n
- total no of operations is m
- Each UNION operation reduces no of sets by one, so that after $n-1$ such operations, only one set remains.
- Therefore, m cannot exceed $n-1$ and $m \geq n$ since MAKE_SET operations are included in m .
- Then $q=m-n$ operations requires incremental no of updations $O(q^2)$ with i -th UNION operation requiring $O(i)$.
- Total time is $O(m^2)$ i.e. average $O(m)$ per operation
- Some heuristics that update representative of smaller list to that of larger list can reduce this complexity.

Linked list representation

- An element x is always on the smaller set if its representative pointer is updated.
- Hence first time, resulting set has at least two members, next time at least 4 and so on.
- For any $k \leq n$ after pointer of x is updated $\lg(k)$ times, resulting set has at least k members.
- Hence a pointer of an object is updated at most $\lg(n)$ times. For n objects, this is $O(n \lg n)$.
- Since there are $O(m)$ MAKE and FIND operations taking $O(1)$ time each, total time for entire operation is $O(m+n \lg n)$

Disjoint set forest

- MAKE-SET creates trees with just one node
- FIND-SET chases parent pointers upto the root
- UNION causes root of one tree to point to other
- Heuristics – union by rank, path compression
- Rank- approximates the logarithm of the subtree size, it is also the upper bound on height of the node
- Root with smaller rank is made to point to the one with larger rank
- Path compression – makes each node on find path point directly to the root – rank not affected by this

Disjoint Set Forest Algorithms

MAKE-SET(x)

$p[x] = x$

$\text{rank}[x] = 0$

FIND-SET(x)

 if $x \neq p[x]$

$p[x] = \text{FIND-SET}(p[x])$

 return $p[x]$

UNION (x, y)

$\text{LINK}(\text{FIND-SET}(x), \text{FIND-SET}(y))$

LINK(x, y)

 if $\text{rank}[x] > \text{rank}[y]$

$p[y] = x$

 else

$p[x] = y$

 if $\text{rank}[x] == \text{rank}[y]$

$\text{rank}[y]++$

Properties of rank

- P1- follows from definition $\text{rank}[x] \leq \text{rank}[p[x]]$
hence, Subtree rooted at $p[x]$ is larger.
- P2- Let $\text{size}(x)$ be no of nodes in the tree rooted at x . For all tree roots x , $\text{size}(x) \geq 2^{\text{rank}[x]}$
- P3- For any integer $r \geq 0$, at most $n/2^r$ nodes of rank r exists.
- P4- Every node has a rank at most floor ($\lg n$)

Property on size of tree rooted at x

- This can be proved by induction on no of LINK operations.
- Before first LINK on x, this is TRUE since $\text{rank}[x]=0$.
- Let rank, size before LINK be *rank*, *size* and after LINK it becomes *rank'*, *size'*.
- In operation LINK(x,y) let $\text{rank}[x] < \text{rank}[y]$.
- Node y is root of tree formed through LINK and we have $\text{size}'(y) = \text{size}(x) + \text{size}(y) \geq 2^{\text{rank}[x]} + 2^{\text{rank}[y]}$
- Which gives $\text{size}'(y) \geq 2^{\text{rank}[y]} \geq 2^{\text{rank}'[y]}$ [no rank changes other than y]
- When $\text{rank}[x] = \text{rank}[y]$, $\text{size}'(y) \geq 2 \cdot 2^{\text{rank}[y]} = 2^{\text{rank}[y]+1} = 2^{\text{rank}'[y]}$
- **Hence by induction, For all tree roots x, $\text{size}(x) \geq 2^{\text{rank}[x]}$**

Counting nodes within a rank r

- When rank r is assigned to x , attach a label x to all nodes of the tree rooted at x .
- At least 2^r nodes are labeled each time. When root changes for x , rank of root is at least $r+1$. Hence no new node is labeled x for this.
- Each node is therefore labeled at most once. There being n nodes in all, at most n labeled nodes with at least 2^r labels assigned for each node of rank r .
- If there are more than $n/2^r$ nodes of rank r , then more than $(n/2^r) \cdot 2^r$ i.e. more than n nodes would be labeled by a node of rank r , which is a contradiction.

Maximum rank possible for a node

- Let $r > \lg n$, then there are at most $n/2^r < 1$ node of rank r .
- But this is impossible as rank is integer.
- **Every node has a rank at most floor ($\lg n$)**

Dividing ranks into rank groups

- Rank 0, 1 \rightarrow rank group 1; Rank 2, $2^2-1 \rightarrow$ rank group 2
- Rank 4 to $2^{2^2}-1$ (15) \rightarrow rank group 3
- Rank 16 to $2^{2^{2^2}}-1$ (255) \rightarrow rank group 4
- Rank $F(g)$ to $2^{F(g)} - 1 \rightarrow$ rank group g
- $F(g) = 2^{2^{2 \dots g \text{ times}}} - 1$ so that $G(n) = \lg^*(n)$ take \lg till n reduces to 1.
- This puts rank r into group $\lg^*(r)$ for $r=0,1,\dots, \text{floor}(\lg n)$
- Highest group no will be $\lg^*(\lg n) = \lg^*(n) - 1$.
- Then j -th group has ranks $\{F(j-1)+1, F(j-1)+2, \dots, F(j)\}$

Time complexity for transitions

- Two cost types: within group and transition to higher rank group.
- **In Transition cost**, there can be $\lg^*(n) + 1$ transitions in all for each FIND-SET operation. Once a node has parent in a different group, it can no longer come back to previous group because of heuristics.
- For m FIND-SET operations, total cost of transitions is thus $m(\lg^*(n) + 1)$

Cost within group

- No of nodes are given by
 - $N(g) \leq \sum (n/2^r) = (n/2^{F(g-1)+1}) \sum (1/2^r)$
- The sum running from $r=0$ to $F(g) - F(g-1)+1$ can be changed to an infinite series sum for large g .
- So, $N(g) < n / F(g)$
- For $g=0$, $N(0) = 3n / 2F(0)$ Hence $N(g) \leq 3n / 2F(g)$ for all $g \geq 0$
- Hence considering all rank groups, denoting by $P(n)$ the total cost within groups, can be obtained

Cost within group

- Multiplying no-of-nodes by no-of-ranks and summing from $g=0$ to $\lg^*(n)-1$
- $P(n) \leq \sum [3n / 2 F(g)] [F(g) - F(g-1) - 1] \leq \sum (3n/2)$ since $F(g) \gg F(g-1)$ for large g .
- Hence $P(n) \leq n \lg^*(n)$ so that
 - $T(n) = m (\lg^*(n) + 1)$

Total time complexity

- Total cost is therefore $O(m \lg^*(n) + n \lg^*(n))$ (since $m \geq n$)
 $O(m \lg^*(n) + 1) + n \lg^*(n) = O(m \lg^* n)$
- There are $O(n)$ MAKE-SET and LINK or UNION operations with $O(1)$ time each.
- Total time complexity stays at $O(m \lg^* n)$
- Time per operation is therefore $O(\lg^* n)$ – amortized complexity

MST Lemma

- $G = (V, E)$ be weighted connected graph
- U is a strict subset of V i.e. nodes in G
- T is a promising subset of edges in E such that no edge in T leaves U
- e is a least cost edge that leaves U
- Then the set of edges $T' = T \cup \{e\}$ is promising.

MST - Kruskal's Algorithm

- J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem
Proceedings of the American Mathematical Society, Volume 7, pp. 48-50, 1956.
- Complexity is $O(e \log e)$ where e is the number of edges. Can be made even more efficient by a proper choice of data structures.
- At the end of the algorithm, we will be left with a single component that comprises all the vertices and this component will be an MST for G .

MST - Kruskal

```
Let  $G = (V, E)$  be the given graph, with  $|V| = n$ 
{
    Start with a graph  $T = (V,)$  consisting of only the
    vertices of  $G$  and no edges;
/* This can be viewed as  $n$  connected components, each vertex being one
connected component */
Arrange  $E$  in the order of increasing costs; → GREEDY
for ( $i = 1, in - 1, i++$ ) → DISJOINT SETS
{
    Select the next smallest cost edge;
    if (the edge connects two different connected components)
        add the edge to  $T$ ;
}
}
```

Kruskal – matroids and disjoint sets

MST-Kruskal (G, w)

$A \leftarrow \emptyset$

FOR each vertex v in $V[G]$

DO MAKE-SET(v)

Sort the edges of E in order of non-decreasing w

FOR each edge (u,v) in E in sorted order **DO**

IF FIND-SET(u) \neq FIND-SET(v) **THEN**

$A = A \cup \{(u,v)\}$

 UNION (u,v)

RETURN A

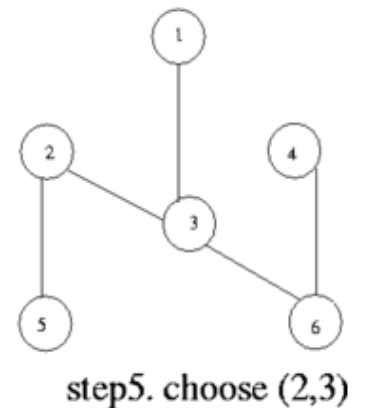
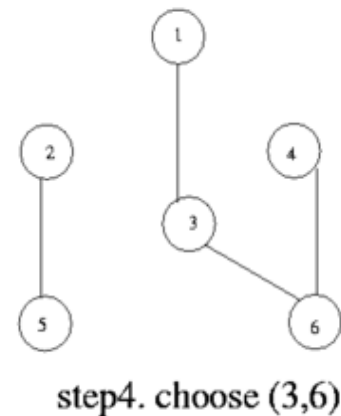
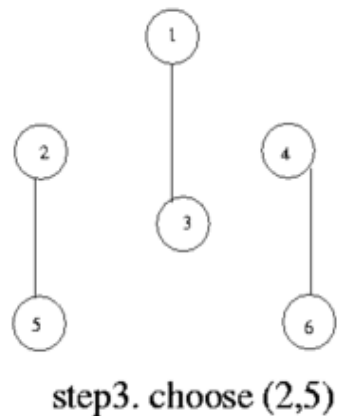
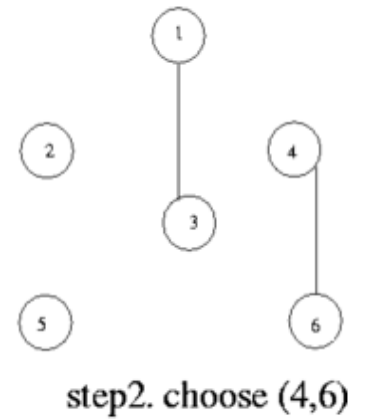
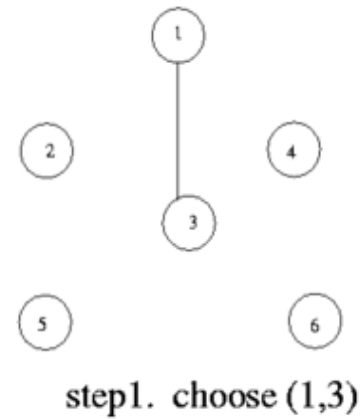
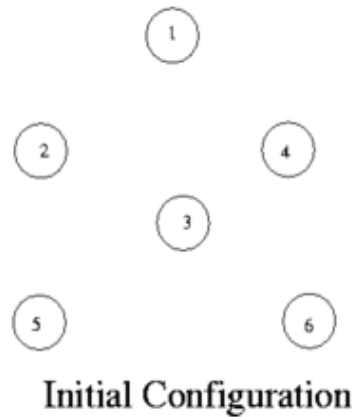
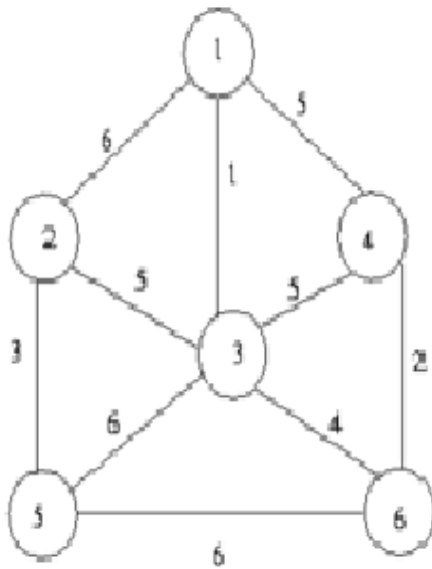
Theorem: Kruskal algorithm finds MST

- **Proof:** Let $G = (V, E)$ be a weighted, connected graph. Let T be the edge set that is grown in Kruskal's algorithm. The proof is by mathematical induction on the number of edges in T .
 - We show that if T is promising at any stage of the algorithm, then it is still promising when a new edge is added to it in Kruskal's algorithm
 - When the algorithm terminates, it will happen that T gives a solution to the problem and hence an MST.
- **Basis:** $T = \Phi$ is promising since a weighted connected graph always has at least one MST.
- **Induction Step:** Let T be promising just before adding a new edge $e = (u, v)$. The edges T divide the nodes of G into one or more connected components. u and v will be in different components. Let U be the set of nodes in the component that includes u .

Theorem: Kruskal algorithm finds MST

- Note that
 - U is a strict subset of V
 - T is a promising set of edges such that no edge in T leaves U (since an edge T either has both ends in U or has neither end in U)
 - e is a least cost edge that leaves U (since Kruskal's algorithm, being greedy, would have chosen e only after examining edges shorter than e)
- The above three conditions are precisely like in the MST Lemma and hence we can conclude that the $T \cup \{e\}$ is also promising. When the algorithm stops, T gives not merely a spanning tree but a minimal spanning tree since it is promising.

Kruskal - illustration



Running Time of Kruskal's Algorithm

- The total time for performing all the merge and find depends on the method used.
- $O(e \log e)$ without path compression
- $O(e(\lg^* e))$ with the path compression

Prim's algorithm - MST

R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, Volume 36, pp. 1389-1401, 1957.

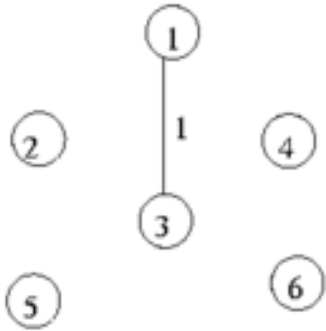
```
{   T =  $\Phi$ ;
    U = { 1 };
    while (U  $\neq$  V)
    {
        let (u, v) be the lowest cost edge
        such that u  $\in$  U and v  $\in$  V - U;
        T = T  $\cup$  {(u, v)}
        U = U  $\cup$  {v}
    }
}
```

- At each step, we can scan lowcost to find the vertex in V - U that is closest to U.
- Then we update lowcost and closest taking into account the new addition to U.
- Complexity: $O(n^2)$

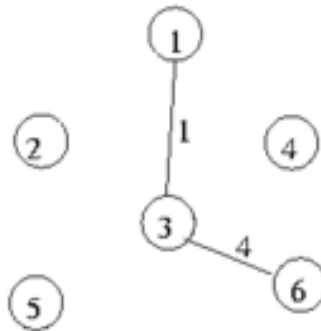
Proof of Correctness-Prim's Algorithm

- Let $G = (V, E)$ be a weighted, connected graph. Let T be the edge set that is grown in Prim's algorithm. The proof is by mathematical induction on the number of edges in T and using the MST Lemma.
- **Basis:** The empty set is promising since a connected, weighted graph always has at least one MST.
- **Induction Step:** Assume that T is promising just before the algorithm adds a new edge $e = (u, v)$. Let U be the set of nodes grown in Prim's algorithm. Then all three conditions in the MST Lemma are satisfied and therefore $T \cup e$ is also promising.
- When the algorithm stops, U includes all vertices of the graph and hence T is a spanning tree. Since T is also promising, it will be a MST.

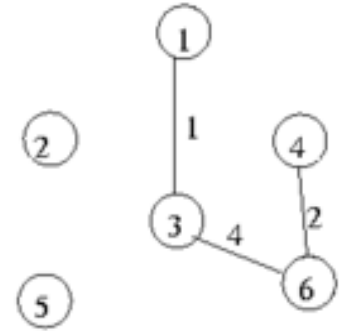
Prim's algorithm - illustration



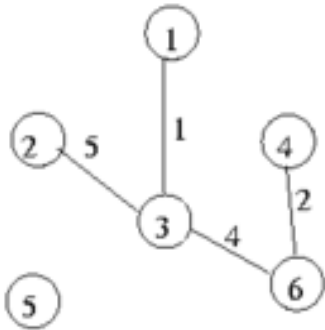
Iteration 1. $U = \{1\}$



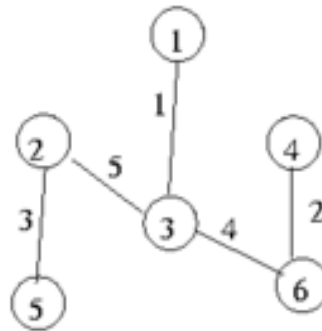
Iteration 2. $U = \{1,3\}$



Iteration 3. $U = \{1,3,6\}$



Iteration 4. $U = \{1,3,6,4\}$



Iteration 5. $U = \{1,3,6,4,2\}$

Difference between Prim and Kruskal

Prim's algorithm

- Initialize with a node
- Graph has to be connected
- Always keep a connected component, look at all edges from the current component to other vertices and find the smallest among them - then add the neighbouring vertex to the component, increasing size by 1.

Kruskal's algorithm

- Initialize with an edge
- Work on disconnected graph
- do not keep one connected component but a forest. At each stage, look at the globally smallest edge that does not create a cycle in the current forest. Such an edge has to necessarily merge two trees in the current forest into one.

Difference between Prim and Kruskal

Prim's algorithm

- In $N-1$ steps, every vertex would be merged to the current one if we have a connected graph.
- Next edge shall be the cheapest edge in the current vertex.
- Prim's algorithm is found to run faster in dense graphs with more number of edges than vertices

Kruskal's algorithm

- Since you start with N single-vertex trees, in $N-1$ steps, they would all merge into one if the graph was connected.
- Choose the cheapest edge, but it may not be in the current vertex.
- Kruskal's algorithm is found to run faster in sparse graphs.

Representation of polynomials

- Coefficient Representation
 - $A(x) = \sum a_j x^j$
- Point Value representation
 - $\langle y_0, y_1, \dots, y_{n-1} \rangle$ evaluated at $\langle x_0, x_1, \dots, x_{n-1} \rangle$
- Evaluation at given x
 - $A(x) = a_0 + x(a_1 + x(a_2 + \dots)) \dots = \sum a_j x^j$
 - Choose $\langle x_0, x_1, \dots, x_{n-1} \rangle$ as the $2n$ -th roots of unity
 - $\omega_n^k = \exp(2\pi i k/n) = \cos(2\pi k/n) + i \sin(2\pi k/n)$

Operation on polynomials

Coefficient representation

- Addition - $O(n)$
 - $C(x)=A(x)+B(x)$
 - $C[j]=a[j]+b[j]$
- Multiplication - $O(n^2)$
 - $C(x)=A(x) \circ B(x)$
 - $C[j] = \sum a[k]b[j-k]$
 - convolution
- Transform to point value
 - $\gamma = V \cdot a$

Point value representation

- Addition - $O(n)$
 - $C(x)=A(x)+B(x)$
 - $\langle \gamma_{c,i} \rangle = \langle \gamma_{a,i} + \gamma_{b,i} \rangle$
- Multiplication - $O(n)$
 - $C(x)=A(x) \cdot B(x)$
 - $\langle \gamma_{c,i} \rangle = \langle \gamma_{a,i} \cdot \gamma_{b,i} \rangle$
 - element wise
- Transform to coefficient
 - $\gamma = V^{-1} \cdot a$

Properties of roots of unity

- Group under multiplication: $\omega_n^k \omega_n^j = \omega_n^{k+j}$
- Cancellation: $\omega_{dn}^{dk} = \omega_n^k$
- Squaring: $(\omega_n^{k+n/2})^2 = \omega_n^{2k} \omega_n^n = (\omega_n^k)^2 = (\omega_{n/2}^k)$
 - Squares of n complex n -th roots = $n/2$ complex $n/2$ -th roots
- Summing all roots: $\sum (\omega_n^k)^j = ((\omega_n^k)^n - 1) / (\omega_n^k - 1) = 0$
- (k,j) th entry of V is (ω_n^{kj})
- (j,k) th entry of V^{-1} has to be $(\omega_n^{-kj})/n$, shown below
- $[V^{-1} V]_{jj'}$ is $\sum (\omega_n^{-kj}/n) (\omega_n^{kj'}) = \sum (\omega_n^{k(j'-j)}/n)$
- When $j=j'$, $[V^{-1} V]_{jj'} = 1$; 0 otherwise so that $[V^{-1} V] = I$

Discrete Fourier Transform

- $\langle y_0, y_1, \dots, y_{n-1} \rangle = \text{DFT}(a_0, a_1, \dots, a_{n-1})$
- $y_k = \sum a_j (\omega_n^{kj})$ with $A(x) = \sum a_j x^j$ and $x = \omega_n^{kj}$
- $A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \dots + a_{n-2} x^{n/2-1}$
- $A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \dots + a_{n-1} x^{n/2-1}$
- $A(x) = A^{[0]}(x^2) + x A^{[1]}(x^2) \rightarrow$ divide and conquer
- Evaluating $A^{[0]}(x^2)$ at $\omega_n^k \rightarrow$ Evaluating $A^{[0]}(x)$ at $\omega_{n/2}^k$
- Therefore problem splits into two equal subproblems
- $T(n) = 2 T(n/2) + O(n) \rightarrow T(n) = O(n \lg n)$

Recursive FFT algorithm (a)

- Basis: if $n==1$ return a // $n=\text{length}[a] = \text{power of } 2$
- Initialize: $\omega_n = \exp(2\pi i/n)$ and $\omega=1$
- Recursive DFT:
 - $y^{[0]} = \text{RFFT}(a^{[0]}) \Rightarrow y_k^{[0]} = A^{[0]}(\omega_{n/2}^k) = A^{[0]}(\omega_n^{2k})$
 - $y^{[1]} = \text{RFFT}(a^{[1]}) \Rightarrow y_k^{[1]} = A^{[1]}(\omega_{n/2}^k) = A^{[1]}(\omega_n^{2k})$
- Combine results
 - For $k=0$ to $n/2 - 1$
 - $y_k = y_k^{[0]} + \omega y_k^{[1]}; y_{k+n/2} = y_k^{[0]} - \omega y_k^{[1]}$
- Update $\omega = \omega \omega_n$
- Return column vector y
- Inverse DFT is same problem with y replacing a

Number Theoretic Algorithms

- Problem size is linear \Rightarrow proportional to number of bits needed to store the number in binary
- $O(n)$ for number n is exponential complexity
- **Fast exponentiation** – x^n – linear in width of n
 - Convert n to binary
 - Compute successive squares of x (takes $O(\lg n)$)
 - Use binary string to pick relevant powers of x
 - Example: $3^{11} \bmod 20 = (1 * 9 * 3) \bmod 20 = 7$
 - $(3^8 \bmod 20) (3^2 \bmod 20) (3^1 \bmod 20)$

GCD computation

euclid (a,b)

If $b=0$ then return a

Else return (euclid(b, a mod b))

Correctness: r_{m+2} is gcd (a,b)

$$a = q_1 b + r_1, b = q_2 r_1 + r_2, r_1 = q_3 r_2 + r_3; r_i = q_{i+2} r_{i+1} + r_{i+2}$$

$$r_{m-1} = q_{m+1} r_m + r_{m+1}; r_m = q_{m+2} r_{m+1} + r_{m+2}; r_{m+1} = q_{m+3} r_{m+2} + 0$$

Now r_{m+2} divides successively r_{m+1} , r_m , r_{m-1} , a , b

Complexity: $\text{gcd}(a,b) \rightarrow \text{gcd}(b,c) \rightarrow \text{gcd}(c,d)$ implies $b = kc + d$ with at least $k=1$; means $b \geq c+d$ and together with $a > b$ gives $a+b > 2(c+d)$. Therefore in every two steps, the sum of the numbers get halved. Hence number of steps (or calls) would be bounded by $\log(b)$ the smaller one.

Worst case: The Fibonacci sequence i.e. $a = F_{n+1}$ and $b = F_n$

Extended Euclid algorithm

Express GCD (a,b) as linear combination of a,b

Say, $d = \gcd(a,b) = ax + by \rightarrow$ to find integers x, y

Example: $a=289, b=204 \Rightarrow \gcd=17$ so that $x=5$ and $y=-7$

Extended_euclid (a,b)

 If $b==0$

 return (a,1,0)

$(d',x',y') = \text{Extended_euclid}(b, a \bmod b)$

$(d'',x'',y'') = (d', y', x' - \text{floor}(a/b) y')$

Apply this to $(a,b) \bmod n = 1$; a,b are multiplicative inverses mod n.

Example: Find multiplicative inverse of 50 mod 71 is 27,

50 and 71 are relatively prime, since $\gcd(50,71)=1$

Primality testing

- Complexity is normally exponential actually $O(\sqrt{n})$
- Can be reduced to linear time – approximation

Fermat's theorem: If p is prime, a is +ve integer,
 $a^{p-1} \bmod p = 1$ i.e. a and p are relatively prime

Is_prime(p)

choose a random no a such that $1 < a \leq p-1$

compute $x = (a^{p-1}) \bmod p$ [fast exponentiation]

if $x \neq 1$, p is composite

else repeat several times, using different a 's

Due to existence of pseudo-primes, condition may fail.

Pseudo-primes

- Number composite, yet obeys $a^{p-1} \bmod p = 1$ for certain choice of a – Base- a pseudoprime
- Base-2 pseudoprime: 341, 561, 645, 1105
- Carmichael number: 561, 1105, 1729 (all bases)
- Distribution of prime numbers:
 - $\pi(n)$ = no of primes $\leq n$
 - $\lim_{n \rightarrow \infty} [\pi(n) / (n/\ln n)] = 1$
 - Implies that density of primes is $(n/\ln n)$

Miller Rabin test for primality

WITNESS (a,n)

$x_0 \leftarrow a^d \bmod n$

for i=1 **to** r

$x_i \leftarrow x_{i-1}^2 \bmod n$

if $x_i = 1$ and $x_{i-1} \neq 1$ and $x_{i-1} \neq n-1$

return TRUE

if $x_t \neq 1$ **then return** TRUE

return FALSE

MillerRabin (n,s)

for j=1 **to** s

$a = \text{RANDOM}(1, n-1)$

if WITNESS(a,n) **return** Composite

return probably prime

- If there exists a nontrivial square root of 1, modulo n, then n is composite e.g. take $6^2 \bmod 35 = 1$ but $\sqrt{1} \neq 6$.
- When p^e divides $(x^2 - 1)$ i.e. $(x-1)$ or $(x+1)$ so that x^2 is 1 (mod p^e) which yields solution trivially 1.
- While computing each modular exponentiation, Miller Rabin test looks for a nontrivial square root of 1, modulo n during the squaring or power raisings.
- If it finds one, it stops and returns COMPOSITE. This way it fools 561, Carmichael number (shown for $a=7$)

Example: How Miller Rabin test works

- Take $p=1729$; $n=p-1= 27 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$ i.e. $r=6$, $d=27$
- Pick $a=11$ (randomly)
- Applying modular exponentiation $a^d \pmod n$ gives sequence (11, 121, 809, 919, 809)
- This is followed by the sequence upon squaring of 11^{27} giving (1331, 1065, **1**, ...)
- Existence of the **1** in this latter sequence shows the presence of non-trivial square root of 1.
- Otherwise $11^{1728} \bmod 1729 = 1$ implies prime.

Public key cryptosystem

Public key used is P; Private key used is S

- **Message:**
 - Message encrypted – P of receiver by sender
 - Message decrypted – S of receiver used
 - Cyphertext $C = P(M)$ used for encryption
 - $M = S(C) = S(P(M))$ to decrypt the message
- **Signature:**
 - Signature encrypted with own S by sender
 - Signature decrypted by recipient with P of sender
 - Signature encrypted using $\Sigma = S(\sigma)$
 - Signature decrypted using $\sigma = P(\Sigma) = P(S(\Sigma))$

Creation of public and private keys

- Select at random two large primes p and q
- Compute $n = pq$
- Select small odd integer e relatively prime to $\Phi(n) = (p-1)(q-1)$
- Compute d as multiplicative inverse of e modulo $\Phi(n)$ i.e. $d \cdot e \bmod (p-1)(q-1) = 1$.
- Publish $P = (e, n)$ as public key
- Publish $S = (d, n)$ as private key

RSA cryptosystem Protocol

- $P(M) = C = M^e \pmod n$
 - cyphertext created using public key of recipient, decryption would need private key of the intended recipient
- $S(C) = C^d \pmod n$
 - signed using private key of sender, verify with sender public key
- If p, q are 256 byte numbers, n is 512 bytes.
- $P(S(M)) = S(P(M)) = M^{ed} \pmod n$; $n=pq$
- Now $ed = 1 + k(p-1)(q-1)$
- Hence $M^{ed} \pmod n = M(M^{p-1})^{k(q-1)} \pmod n$
- Using Fermat's theorem, $M^{p-1} \pmod p = 1$
- Hence $M^{ed} \pmod n = M \pmod p = M \pmod q$
- Chinese Remainder Theorem $M^{ed} = M \pmod{pq}$

RSA cryptosystem Protocol

- AA (d_1, e_1, n_1) sends M to BB (d_2, e_2, n_2)
- AA Encrypts $Y_1 = M^{e_2} \pmod{n_2}$
- AA Signs $Y_2 = Y_1^{d_1} \pmod{n_1}$
- AA transmits Y_2
- BB verifies sign $Z_1 = Y_2^{e_1} \pmod{n_1}$
- BB decrypts $Z_2 = Z_1^{d_2} \pmod{n_2}$
- Encrypt-Sign-Transmit-Verify sign-Decrypt

Attacking RSA cryptosystem

- Find a number that leaves remainder 2 when divided by 3 (p) and 3 when divided by 5(q)
 - – Chinese Remainder Theorem
- Then $n \equiv 2 \pmod{3}$ and $n \equiv 3 \pmod{5}$ gives $n=8$
- Such number is unique in the domain $[1 .. pq]$
- n and e are known, can we find d ?
- We need to know $\Phi(n)$ from n
- Then we need to factorize n into its prime factors
 - but integer factorization is hard.

Single Source Shortest Path Problem

- Given a directed graph $G = (V, E)$, with non-negative costs on each edge, and a selected source node v in V , for all w in V , find the cost of the least cost path from v to w .
- The *cost* of a path is simply the sum of the costs on the edges traversed by the path.
- This problem is a general case of the more common subproblem, in which we seek the least cost path from v to a particular w in V . In the general case, this subproblem is no easier to solve than the SSSP problem.

Dijkstra's Algorithm

- Dijkstra's algorithm is a *greedy* algorithm for the SSSP problem.
- A "greedy" algorithm always makes the locally optimal choice under the assumption that this will lead to an optimal solution overall.
- Data structures used by Dijkstra's algorithm include:
 - a cost matrix C , where $C[i,j]$ is the weight on the edge connecting node i to node j . If there is no such edge, $C[i,j] = \text{infinity}$.
 - a set of nodes S , containing all the nodes whose shortest path from the source node is known. Initially, S contains only the source node.
 - a distance vector D , where $D[i]$ contains the cost of the shortest path (so far) from the source node to node i , using only those nodes in S as intermediaries.

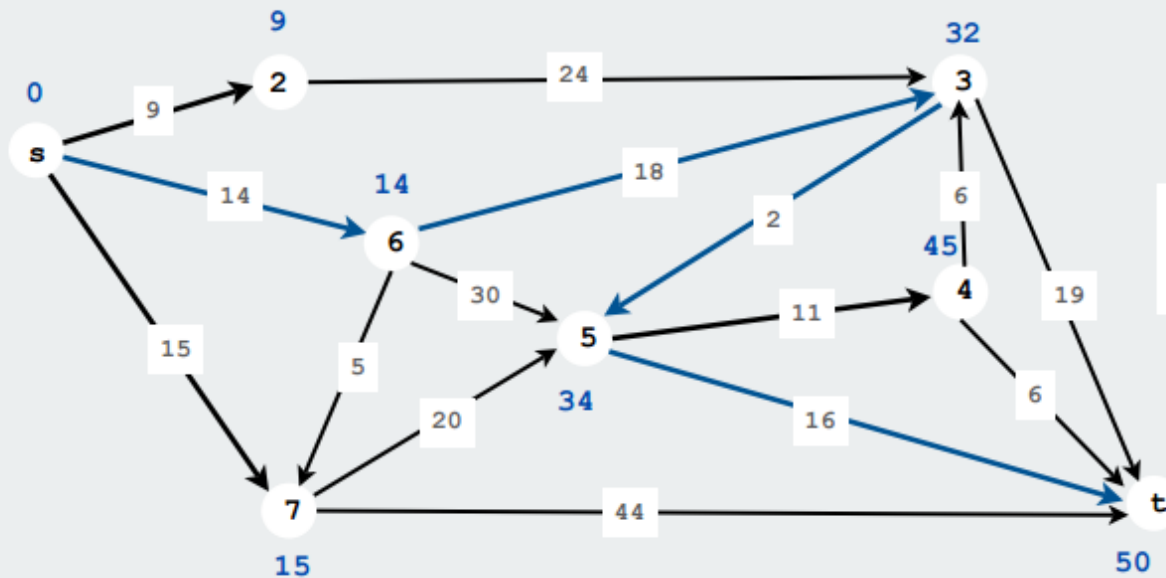
How Dijkstra's Algorithm Works

- On each iteration of the main loop, we add vertex w to S , where w has the least cost path from the source v ($D[w]$) involving only nodes in S .
- We know that $D[w]$ is the cost of the least cost path from the source v to w (even though it only uses nodes in S).
- If there is a lower cost path from the source v to w going through node x (where x is not in S) then
 - $D[x]$ would be less than $D[w]$
 - x would be selected before w
 - x would be in S

SSSP - One illustration

Given a **weighted digraph**, find the shortest directed path from s to t .

cost of path = sum of edge costs in path



Path: $s \rightarrow 6 \rightarrow 3 \rightarrow 5 \rightarrow t$
Cost: $14 + 18 + 2 + 16 = 50$

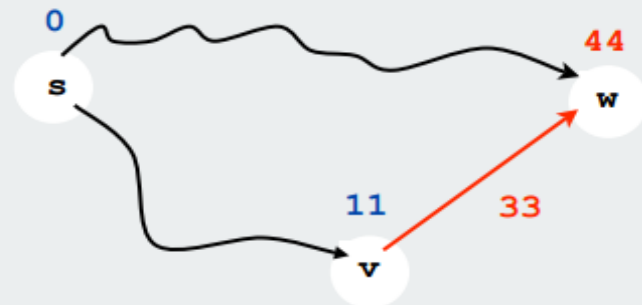
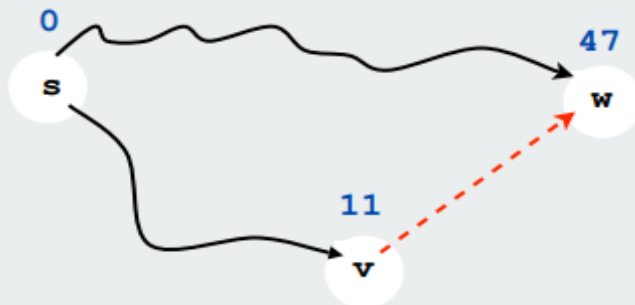
Edge relaxation

For all v , $\text{dist}[v]$ is the length of **some** path from s to v .

Relaxation along edge e from v to w .

- $\text{dist}[v]$ is length of some path from s to v
- $\text{dist}[w]$ is length of some path from s to w
- if $v-w$ gives a shorter path to w through v , update $\text{dist}[w]$ and $\text{pred}[w]$

```
if (dist[w] > dist[v] + e.weight())  
{  
    dist[w] = dist[v] + e.weight();  
    pred[w] = e;  
}
```



Relaxation sets $\text{dist}[w]$ to the length of a **shorter** path from s to w (if $v-w$ gives one)

Dijkstra's Algorithm

S : set of vertices for which the shortest path length from s is known.

Invariant: for v in S , $\text{dist}[v]$ is the length of the shortest path from s to v .

Initialize S to s , $\text{dist}[s]$ to 0, $\text{dist}[v]$ to ∞ for all other v

Repeat until S contains all vertices connected to s

- find e with v in S and w in S' that minimizes $\text{dist}[v] + e.\text{weight}()$
- relax along that edge
- add w to S

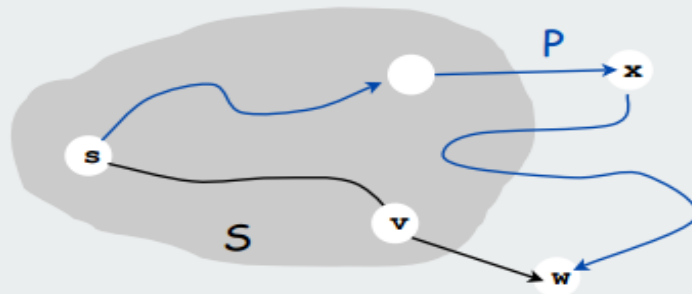
Correctness of the algorithm

S : set of vertices for which the shortest path length from s is known.

Invariant: for v in S , $\text{dist}[v]$ is the length of the shortest path from s to v .

Pf. (by induction on $|S|$)

- Let w be next vertex added to S .
- Let P^* be the s - w path through v .
- Consider any other s - w path P , and let x be first node on path outside S .
- P is already longer than P^* as soon as it reaches x by greedy choice.



Analysis of Dijkstra's Algorithm

- Consider the time spent in the two loops:
- The first loop has $O(N)$ iterations, where N is the number of nodes in G .
- The second (and outermost) loop is executed $O(N)$ times.
 - The first nested loop is $O(N)$ since we examine each vertex to determine whether or not it is in $V-S$.
 - The second nested loop is $O(N)$ since we examine each vertex to determine whether or not it is in $V-S$.
- The algorithm is $O(N^2)$.
- If we assume that there are many fewer edges than the maximum possible, we can do better than this.

Complexity of algorithms

- Polynomial time: worst case $O(n^k)$
- Super polynomial time: solvable but not in Polynomial time
- Unknown status: no Polynomial time algorithm found, no proof of Super Polynomial time lower bound

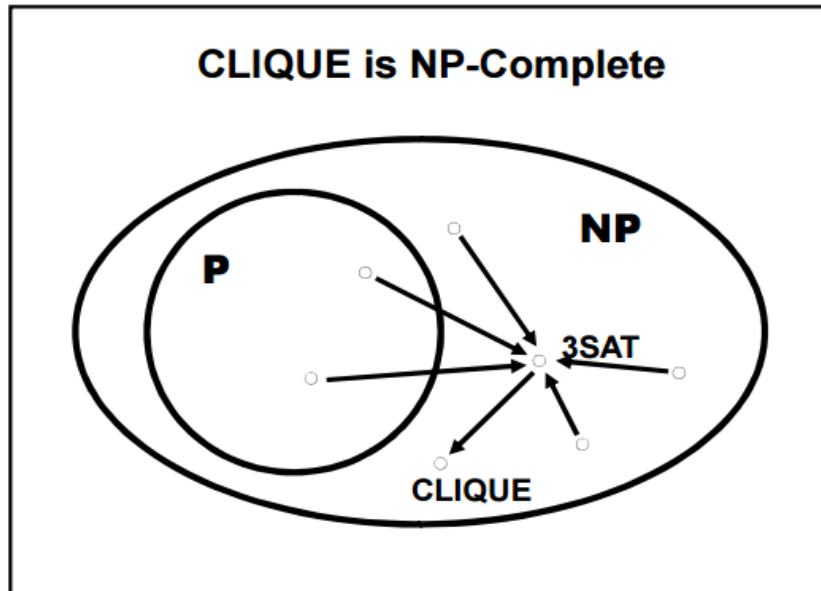
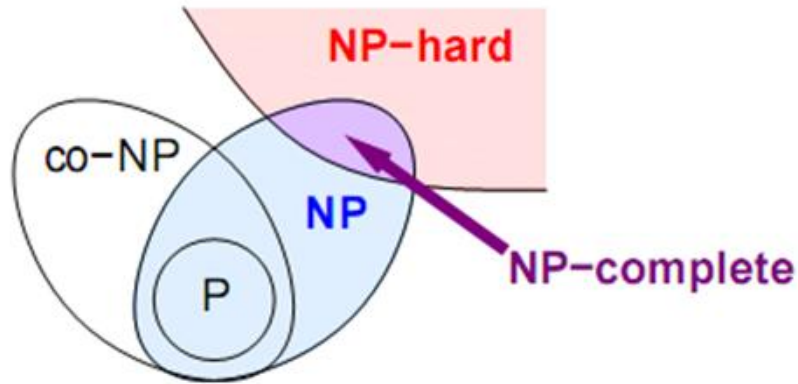
Models of computation

- Serial random access machine
- Parallel random access machine
- Abstract Turing machine

Complexity Classes

- P: problems that can be solved in *polynomial time* (typically in n , size of input) on a *deterministic Turing machine (DTM)*
 - Any normal computer simulates a DTM
- NP: problems that can be solved in *polynomial time* on a *non-deterministic Turing machine (NDTM)*
 - Informally, if we could “guess” the solution, we can *verify* the solution in P time (on a DTM)
 - NP does NOT stand for *non-polynomial*, since there are problems harder than NP
 - P is actually a subset of NP (we think)

Complexity Classes Overview



- NP-hard
 - At least as hard as any known NP problem (could be harder!)
 - Set of interrelated problems that can be solved by *reducing* to another known problem
- NP-Complete
 - A problem that is in NP and NP-hard
- Cook's Theorem
 - SATISFIABILITY (SAT) is NPC
- Other NPC problems
 - Reduce to SAT or previous reduced problem

Problem definitions

- Abstract problem Q is a binary relation on a set I of problem instances and set S of solutions
- $G=(V,E)$: Instances of shortest path
- Solution: sequence of vertices for abstract problem
- Decision problem: $I \rightarrow \{0,1\}$ is a given path of length less than some threshold
- The decision problem is as complex as the abstract problem
- The abstract problem is optimization problem and decision problem somehow maps it to binary.

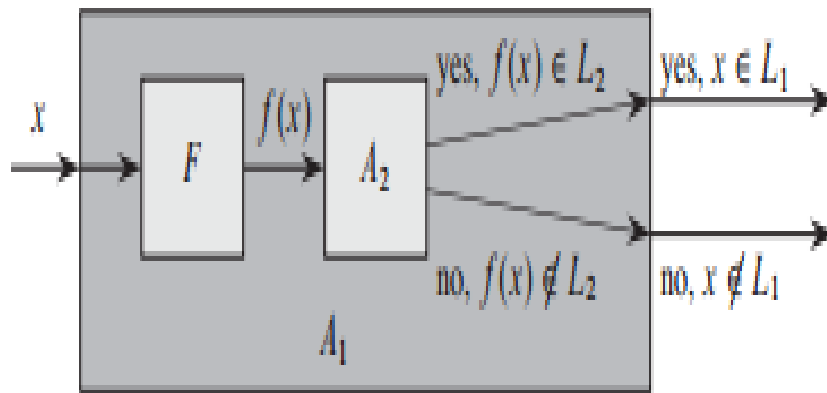
Problem encoding

- Unary encoding – integer k represented using k ones – somewhat like Roman number system
- Binary encoding – length $n = \text{floor}(\lg k)$
- Linear complexity in unary encoding \Rightarrow log complexity in binary encoding
- Linear complexity in binary encoding \Rightarrow exponential complexity in unary encoding
- Two encodings e_1 and e_2 are related polynomially
- $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$

P-time Reducibility

- Let Q be an abstract decision problem on an instance set I and encodings e_1 and e_2 are related polynomially on I – then $e_1(Q) \in P$ iff $e_2(Q) \in P$
- If a problem Q reduces to another problem Q' then Q is no harder to solve than Q'
- Language L_1 is P-time reducible to L_2 denoted as $L_1 \leq_p L_2$ implies there is a P-time computable reduction function $f: \{0,1\}^* \rightarrow \{0,1\}^*$ such that for all $x \in \{0,1\}^*$ we have $x \in L_1$ iff $f(x) \in L_2$
- Hence for $L_1 \leq_p L_2$ then $L_2 \in P$ implies $L_1 \in P$

Reduction mapping



- The algorithm F is a reduction algorithm
- Computes the reduction function f from L_1 to L_2 in P-time
- A_2 is P-time algorithm that decides L_2 .
- A_1 decides whether $x \in L_1$ by using F to transform any input x into $f(x)$ and then using A_2 to decide whether $f(x) \in L_2$

Notion of NP Completeness

- Class P – if there exists an algorithm A that decides L in polynomial time then $L \in P$
- For 2-input P-time algorithm A and constant c, $L = \{x \in \{0,1\}^* : \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y)=1\}$
- If algorithm A verifies language L in P-time, then $L \in NP$
- In case $L \in P$ then $L \in NP$ (solved \Rightarrow verifiable)
- Property-1: $L \in NP$; Property-2: $L' \leq_p L$ for every $L' \in NP$
- When both properties hold then $L \in NPC$
- If only property-2 holds, then $L \in NP\text{-Hard}$

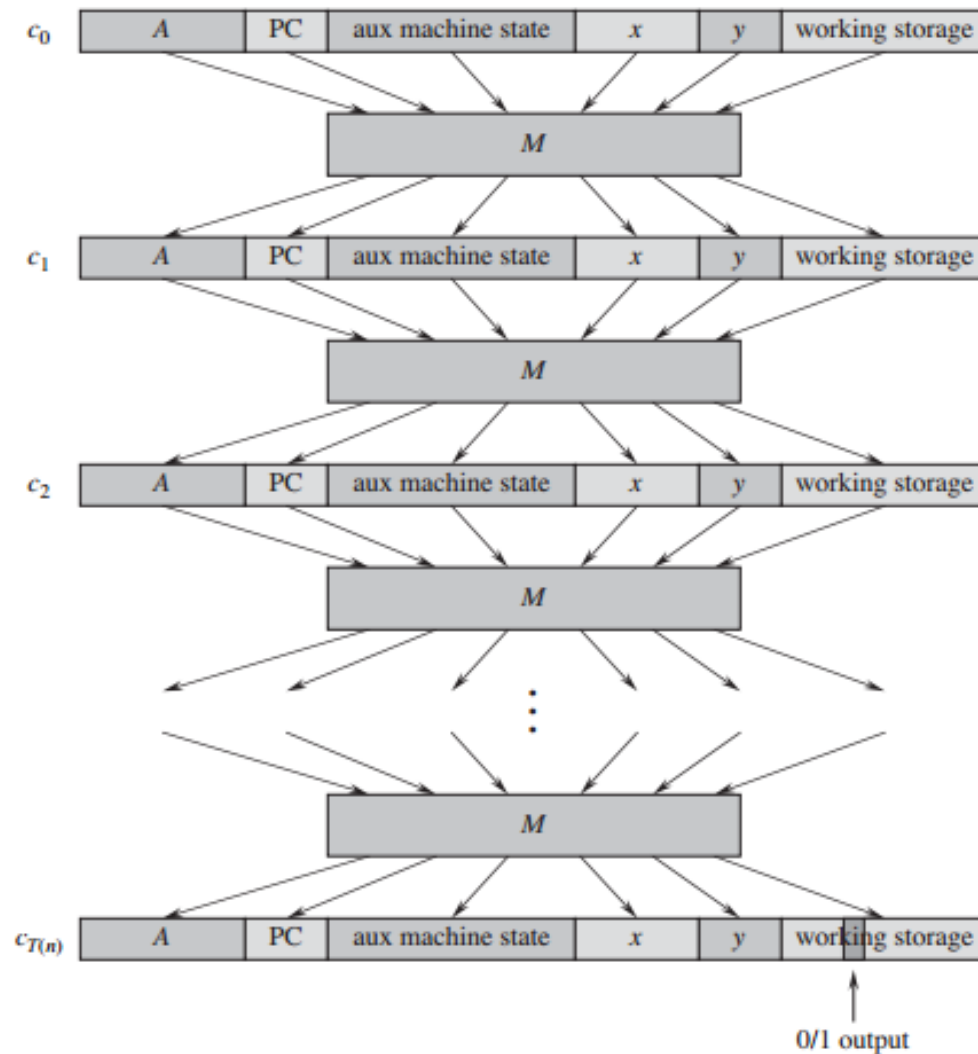
Theorem of NP-Completeness

- If any NPC problem is P-time solvable, $P=NP$
 - Suppose L belongs to both P and NPC. For any L' in NP, $L' \leq_p L$ (property-2 of NPC) Then such L' also belongs to P (reduce and solve)
- If any problem in NP is not P-time solvable, then all NPC problems are not P-time solvable
 - Suppose some L belongs to NP but not in P . Let L' be some NPC and to contradict assume that L' is in P . Then $L \leq_p L'$ (reduce) and hence L is also in P .

Circuit Satisfiability Problem

- Take any algorithm that produces output for given input in P-time \Rightarrow verification $\Rightarrow \in \text{NP}$
- Can map this into program steps
- Each program step maps to combinational circuit
- Paste these circuits - maps to overall circuit
- Program I/O maps to circuit I/O in P-time
- All such algorithms are reducible (\leq_p) to CSAT
- CSAT is therefore NPC
- Such proof outline possible only for CSAT

Algorithm as computation sequence



Proof of NP-completeness for some L

- Prove L belongs to NP (verification decision)
- Select a known NP complete language L'
- Describe an algorithm that computes f , which is a function mapping every instance of L' to L
- Prove – for all x , f satisfies $x \in L'$ iff $f(x) \in L$
- Prove – algorithm computing f runs in P-time
- $\text{CSAT} \leq_p \text{FSAT} \leq_p \text{3CNF-SAT} \leq_p \text{CLIQUE (graph)} \leq_p \text{VERTEX-COVER} \leq_p \text{SUBSET-SUM (0-1 knapsack)}$
- $\text{CSAT} \leq_p \text{FSAT} \leq_p \text{3CNF-SAT} \leq_p \text{HC} \leq_p \text{TSP}$

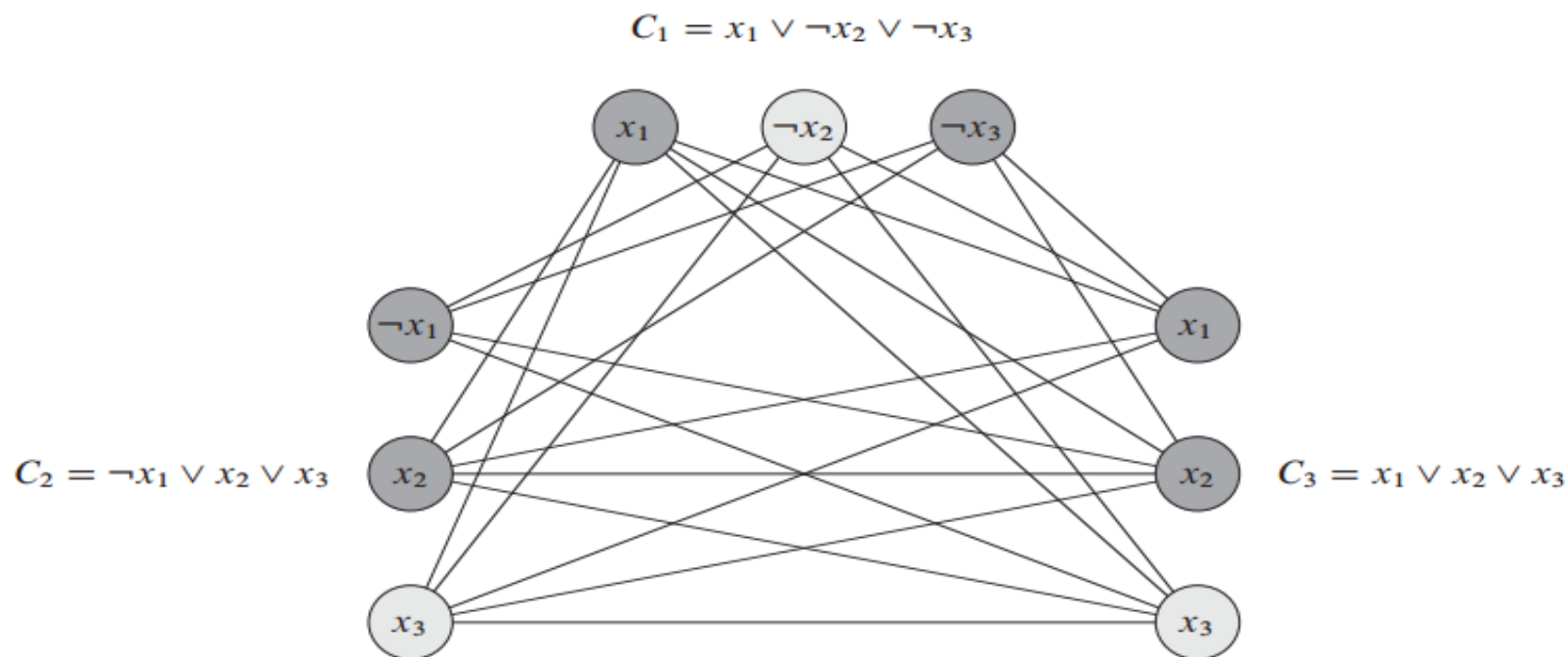
Clique of a graph

- Clique of size k : Find a subset of k vertices that are connected through edges to be found in E .
- Brute force - to check clique in all $|k|$ subsets
 - runs in superpolynomial time $C(|V|,k)$
- Start from 3CNF: Boolean formula with k AND-ed clauses, each clause having 3 OR-ed literals
- The graph should be such that the formula is satisfiable iff the graph has a clique of size k

Reduction:- $3\text{CNF-SAT} \leq_p \text{CLIQUE}$

- For each clause C_r place triple vertices v_{1r}, v_{2r}, v_{3r} for the 3 literals l_{1r}, l_{2r}, l_{3r}
- Construct an edge between v_{ir} and v_{js} when they fall in different clauses (r and s) AND their corresponding literals l_{ir} and l_{js} are consistent i.e. l_{ir} not negation of l_{js}
- When formula has a satisfying assignment, each clause has at least one 1 (OR within each clause and k such AND-ed clauses) resulting in k vertices. They form a clique since edges can be found by way of above construction.
- Conversely suppose G has a clique of size k . Since no edges in V connect same triple, this set has one vertex (read literal) per triple (read clause). We can assign 1 to each such literal since G has no edge between inconsistent literals. So each clause gets satisfied and the formula also gets satisfied.

Example: 3-CNF formula to graph



clique mapping: The graph G derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 either 0 or 1. This assignment satisfies C_1 with $\neg x_2$, and it satisfies C_2 and C_3 with x_3 , corresponding to the clique with lightly shaded vertices.

Hallmarks of the CLIQUE mapping

- The graph constructed here is of a special kind since vertices here occur as triplets with no edges between vertices in same triplet
- This CLIQUE happens in restricted case but the corresponding 3CNF case is very general
- But if we had a polynomial-time algorithm that solved CLIQUE on general graphs, it would also solve CLIQUE on restricted graphs.
- Opposite approach is however not enough – in case an easy 3CNF instance were mapped, the NP-hard problem does not get mapped
- The reduction uses instances, not the solution – actually we do not know whether we can decide 3CNF-SAT in P-time!

Intuitive Mapping: 3CNF-SAT to HC

- One approach is to follow the reduction from clique:
 $\text{CLIQUE} \leq_p \text{VERTEX-COVER} \leq_p \text{Hamiltonian Path} \leq_p \text{HC}$
- Direct approach - Encode an instance I of 3-SAT as a graph G such that I is satisfiable exactly when G has HC
- Create some graph that represents the variables
- Create some graph that represents the clauses (each clause has exactly three literals/variables)
- Hook up the variables with the clauses such that the formula gets encoded
- Show that this graph has HC iff the formula in conjunctive normal form is satisfiable.

Reduction – HC to TSP

- Travelling Salesman Problem works on complete graph $G=(V,E)$ with cost function c defined from $V \times V \rightarrow \mathbb{Z}$ with $k \in \mathbb{Z}$ and G has a TSP tour with cost at most k
- Let $G=(V,E)$ be an instance of HC. Form the complete graph $G'=(V,E')$ with cost function $c(V_i,V_j)$ with $k=0$
- $c(i,j) = 0$ if $(i,j) \in E$ and $c(i,j) = 1$ if $(i,j) \in E'-E$
- Instance of TSP is taken to be $TSP(G',c,0)$
- Since G has HC, G' has valid TSP tour of at most 0
- If G' has TSP tour of 0, the tour contains only edges from E , which in turn implies that G has HC $\Rightarrow HC \leq_p TSP$ (NP-hard)
- Since a TSP tour can be verified in P-time it is in NP and together with it being NP-hard therefore $TSP \in NPC$.

Branch & Bound approximation for TSP

- B&B algorithm performs a top-down recursive search through the tree of instances formed by the branch operation.
- Upon visiting an instance I , it checks whether $\text{bound}(I)$ is greater than the upper bound for some other instance that it already visited; if so, I may be safely discarded from the search and the recursion stops.
- This pruning step is usually implemented by maintaining a global variable that records the minimum upper bound seen among all instances examined so far.
- Generic B&B is related with backtracking as in DFS traversals
- And the journey continues...