Chapter 1 : Role of Algo. in Computing

Defn (Algorithm) well defined computational procedure that takes some values or a set of values as input and produces some value or a set of values as output.

* Algorithm is a tool for solving well-specified comput.

Example : Sort a sequence of numbers.

Sorting problem:

Input : seq. of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output : $a_1' \leq a_2' \leq \ldots \leq a_n'$.

Example : $\langle 31, 41, 59, 26, 41, 58 \rangle$ is input.

Output : $\langle 26, 31, 41, 41, 58, 59 \rangle$

There are many sorting algorithms. Which one to choose depends on:

* no. of items to be sorted
* whether the items are somewhat sorted
* Architecture of computer [CPU, GPU, FPGA]
* Storage device

Defn (Correct Algorithm) An algorithm that halts with correct output. Note: An algorithm that does not halt is incorrect.

Defn (Incorrect alg.) An algorithm that is not correct.

* Obviously, we are interested in correct alg.
* An algorithm can be in various forms:

Algorithm

English words (pseudocode)

Computer Progsam (C/C++, Java, Python)

hardware design (FPGA)

* Problems solved by algorithms
  1) Human Gnome Project
  2) Internet
  3) Electronic commerce

Specific Problems

*) Shortest route from one intersection to another in a road map

Difficulty: no. of possible routes can be huge!

* Model map as graph and find the shortest path from one vertex to another.

*) Given sequences:

$X = \langle x_1, x_2, \ldots, x_m \rangle$

$Y = \langle y_1, y_2, \ldots, y_n \rangle$

Find longest common subseq. of X and Y.

* length of longest common subseq. gives a measure of similarity.

Algorithms

* In a brute-force approach, we may enumerate all subseq. of X and check each subseq. to see whether it is also a subseq. of Y. But X has 2^m subsequences [No. of subsets of a set with m elements is 2^m].

Defn (Efficient Alg) An algo is defined to be efficient for a variety of reasons:

1) It is fast (polynomial run time)
2) It takes less resources, for eg, less memory

Defn (Inefficient Algo) An algo can be said to be inefficient for the following reasons:

1) No polynomial time algo. exists
2) Takes large memory

In view of efficient/inefficient algo., we consider three classes:

Defn (P class) : Sol^n found in polynomial time.

Defn (NP class) : No polynomial time algo. to find the answer, but if the answer is shown, it can be verified in polynomial time.

$P = NP$? unsolved!

Example of NP: subset sum problem

Q Given a set of integers, does the some nonempty subset of them sum to 0?

A No poly. time algo known, but easy to check if the answer is known.

Example: $\{-2, -3, 15, 14, 7, -10\}$

* Easy to check that: $\{-2, -3, -10, 15\}$ sums to zero.
* Existing algo is exponential time.

Parallelism

* Processor clock speed can't be increased indefinitely because it leads to heating.
* Since clock speed can't be increased, consider multiple cores, but need to progsam the code to benefit from parallelism.
* Even single core machines are not really sequential: there are superscalar & vector units.
* Instructions common in computers are:
  - arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling, etc
  - data movement: load, store, copy
  - control: conditional and unconditional branch.

Algorithms

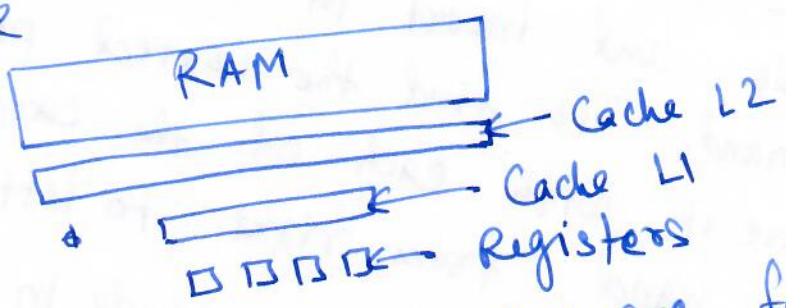
* When we design algorithm, we do not see or we ignore the underlying arch.
* Algorithms are usually developed with certain idealistic model of computer: Random Access Machines.

Random Access Machines ≠ Random Access Memory

Def^n (Random Access Machines): Imagine a computer where access to any memory is same cost.

Q Can Random Access Machines be real?

A No. Not all memory can be fit in fastest possible memory. Recall the design of computer architecture



Since registers are costly, they are few in numbers, data that is expected to be accessed frequently or in near future is cached in Cache. Cache is slower but larger memory compared to registers. Such a design makes access to data from RAM to register different.

* An efficient algo can be written by coding an efficient data movement (such programming is called: High Performance Computing)

Let us look at our 1st algorithm

Insertion Sort

Input: Sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output: Sequence of n sorted numbers

$\langle a_1', a_2', \ldots, a_n' \rangle$

$a_1' \leq a_2' \leq \ldots < a_n'$

Idea: How people sort a hand of playing cards.

step 1 Start with an empty left hand and cards face down on the table.

step 2 Remove one card at a time from the table and insert in correct position in the left hand. To find the correct position, we compare it with each of the cards already in the hand from right to left.

step 3 At all times, cards in left hand are sorted.

Example:

a) 

b) 

c) 

d) 

e) 

← sorted array :-)

Algorithms

Time to write pseudocode:

pseudocode

```
INSERTION-SORT(A)
1. for j = 2 : A.length
2.     key = A[j]
3.     // insert A[j] into the sorted seq.
       // A[1,..,j-1]
       i = j-1
4.     while i>0 and A[i]>key
5.         A[i+1] = A[i]
6.         i = i-1
7.
8.     A[i+1] = key
```

Remarks:

* $A[1,\cdots,j-1]$: currently sorted cards
* $A[j+1,\cdots,n]$: pile of cards on table

Note: $A[1,\cdots,j-1]$ being in sorted orders in j^{th} iteration is called loop invariant.

Loop invariance of insertion sort: At the start of each iteration of the for loop of lines 1-8, the subarray $A[1,\cdots,j-1]$ consists of elements in sorted orders.

* Loop invariance used to prove correctness of algo.

Q How can loop invariance used to prove correctness?

A If we somehow show that loop invariance holds for all j, then for $j = n+1$, certainly, $A[1, \ldots, n]$ is sorted, which proves that insertion sort <u>does</u> sort a given set of n integers..

Q. How do we prove loop invariance property?

A To prove correctness of an algorithm we do the following:

1) find a loop invariant, the loop invariant found should be such that it shows the correctness of the result. For example, for insertion sort, taking $A[1, \ldots j-1]$ is sorted until jth iteration is a <u>nice</u> loop invariant.

2) Once we find a suitable loop invariant, we prove that this invariant property holds. This is similar to induction proof. For example, someone makes the following observation (invariant poop.)

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2} \longrightarrow (*)$$

To prove (*), we prove the induction step. For example: We assume (*) is true for some k, and we show that it is true for k+1; assuming k is arbitrary is essential.

Similarly, to show that $A[1,\ldots,j-1]$ is sorted show that for arbitrary k, $A[1,\ldots,k]$ is sorted assuming that $A[1,\ldots,k-1]$ was already sorted.

3) Recall that an algo. is correct when it terminates and after termination produces correct result. Hence unlike ~~ind~~ (*) which holds for all n, for insertion sort, even though inductively $A[1,\ldots,j-1]$ may remain sorted for arbitrary j, to show correctness of algo, we must show that it terminates. That is, there must be a stopping criteria in Algo.

Proving correctness requires 3 steps

(1) Initialization: true prior to the 1st iteration

(2) Maintenance: true before an iteration, true before next iteration

(3) Termination: when loop terminates, invariant shows that the algo is correct.

Check Loop invariance for Insertion Sort

Initialization: Loop invariant holds before 1st iteration i.e. before $j=2$ because $A[1]$ is by default sorted because it contains only one element.

Maintenance: Like induction step in induction proof let us assume that loop invariance is true for $j=k$, i.e., $A[1\ldots k]$ are sorted. We need to show that loop invariance is true for $j=k+1$.

In steps ⑤, ⑥, and ⑦ of INSERTION-SORT Algo. we will compare $A[k+1]$ with $A[1], \ldots, A[k]$ and insert $A[k+1]$ in the "right" place, i.e., we insert $A[k+1]$ s.t. $A[1\ldots k+1]$ are sorted. Hence, we proved that if loop invariance holds for $j=k$, it holds for $j=k+1$. Hence, maintenance step holds.

Termination: When $j > A.length = n$, i.e., as soon as $j=n+1$, the algo. terminates because from step ① in Algo for loop runs from 2 to n. We want to know when the loop terminates, whether $A[1\ldots n]$ remains sorted. But the array $A[1\ldots n]$ will remain sorted, because maintenance step says that for $j=n+1$, $A[1\ldots n]$ will be sorted.

But if $A[1\ldots n]$ is sorted + Algo. terminates

$\Rightarrow$ Algo. is correct.

Algorithms

Analysis of Algorithms

By analysis, we would mean estimate the running time of algorithm as a fn of input size. Obviously, these can be multiple inputs. For example, an input can be a graph, i.e., a set of vertices and a set of edges.

Running time: No. of "primitive ops".

Here primitive operations can be: comparisons, addition, multiplication, etc. Hoxcues, they can be further classified. For example, arithmetic ops are often classified as FLOPS (Floating point ops). These primitive ops may take different no. of cycles to complete, but nevertheless they are comparable and constant. For example, comparing two numbers may take two cycles, whereas multiplying two numbers may take six cycles.

Cost of Insertion Sort Algo

	INSERTION-SORT(A)	cost	times
1.	for $j = 2$ to A.length	c_1	n
2.	key = A[j]	c_2	$n-1$
3.	i = j-1	c_3	$n-1$
4.	while i>0 and A[i]>key	c_4	$\sum_{j=2}^{n} t_j$
5.	A[i+1] = A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
6.	i = i-1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7.	A[i+1] = key	c_4	$n-1$

In step ① of Algo, there is a cost involved in checking whether j is in the range 2 to A.length=n.

t_j : no. of times the while loop in step 5 is executed for that value of j.

* When a for loop or while loop exits, the test (in for loop or while loop) is executed one time more than the loop body.

* So, when the while loop in step 5 got executed t_j times, then the body of the while loop, i.e., the steps ⑤ and ⑥ got executed $t_j - 1$ times.

This is the reason why:

Cost of step ④ : $\sum_{j=2}^{n} t_j$

Cost of step ⑤ : $\sum_{j=2}^{n} t_j - 1$ [Cost of each assignment is c_5]

Cost of step ⑥ : $\sum_{j=2}^{n} t_j - 1$ ["]

* If a statement costs c_i and if that statement executes n times, then the total cost is $c_i n$.

$T(n)$: running time of insertion sort

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} t_j - 1 + c_6 \sum_{j=2}^{n} t_j - 1 + c_7(n-1)$$

Worst case: Array reverse sorted $\Rightarrow$ $A[j]$ must be compared with each element in the sorted array $A[1 \cdots j-1]$ $\Rightarrow$ $t_j = j$, $j = 2, 3, \ldots n$.

We have:

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} t_j - 1 = \sum_{j=2}^{n} j - 1 = \frac{n(n-1)}{2}$$

$$\Rightarrow T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1).$$

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right) n - \left(c_2 + c_3 + c_4 + c_7\right).$$

$= a n^2 + bn + c$, where a, b, c are constants. Hence $T(n)$ is a quadratic function of n.

Best Case: When the array is already sorted, we have best case. In this case $A[i] \leq key$ for all $j = 2, 3, \ldots n$ in step ④ of algo, Hence, while loop at step ④ is run once, i.e., $t_j = 1$ for all $j = 2, 3, \ldots, n$.

$$\Rightarrow T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) = (c_1 + c_2 + c_3 + c_4 + c_7) n - (c_2 + c_3 + c_4 + c_7).$$

$= an + b$, where a and b are constants. Hence, $T(n)$ is a Linear fn of n.

Average Case Analysis : Requires Probability Concepts * covered in Adv. Algo & Complexity course.

Q When one algo. is more efficient than other?

A. when the worst case running time of one is of lower order than the other.

For eg. an algorithm whose worst case run. time is of order "linear" and 2nd algo whose worst case run. time is of order "quadratic" then 1st algo is said to be more efficient.

Divide and Conquer

Divide: divide the problem into a number of subproblems

Conquer: Conquer the subproblems by solving them recursively

Combine: Combine the solutions to the subproblems into the solution for the original problem.

Merge Sort

Divide: divide the n-element sequence to be sorted into two subseq. of $n/2$ element each.

Conquer: Sort the two subseq. recursively using merge sort.

Combine: Merge the two sorted subseq. to produce the sorted answer.

Merge step: By calling Merge(A, p, q, r) where A is an array and p, q, and r are indices into the array s.t. $p \le q \le r$. The procedure assume that A[p...q] and A[q+1...r] are in sorted orders. It merges A[p,..q] and A[q+1..r] to form a sorted subarray that replaces A[p...r].

Example of Merge

[1 3 7 9] [2 4 6 10]

Take min(1, 2) as 1st element and put in Z = [1].

[1 3 7 9] [2 4 6 10]

Since the minimum was the element of 1st array, move the pointer ↑ of the 1st array one step forward. Then take min(3, 2) and put in

Z = [1 2]

[1 3 7 9] [2 4 6 10]

Since, last time the min. was from 2nd array, move the pointer ↑ in 2nd array one step forward.

[1 3 7 9] [2 4 6 10]

Following the steps: Z = [1 2 3 4] ...

Following the steps till the end we merge the two sorted arrays into a final sorted array Z.

Merge (A, p, q, r)

1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. let $L[1 \cdots n_1 + 1]$ and $R[1 \cdots n_2 + 1]$ be new arrays
4. for $i = 1$ to n_1
5. $\quad L[i] = A[p + i - 1]$
6. for $j = 1$ to n_2
7. $\quad R[i] = A[q + j]$
8. $L[n_1 + 1] = \infty$
9. $R[n_2 + 1] = \infty$
10. $i = 1$
11. $j = 1$
12. for $k = p$ to r
13. $\quad$ if $L[i] \leq R[j]$
14. $\quad\quad A[k] = L[i]$
15. $\quad\quad i = i + 1$
16. $\quad$ else $A[k] = R[j]$
17. $\quad\quad j = j + 1$

Loop invariant for Merge

At the start of each iteration of the for loop of lines 12-17, the subarray $A[p \cdots k-1]$ contains $k-p$ smallest elements of $L[1 \cdots n_1+1]$ and $R[1 \cdots n_2+1]$ in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have been copied back into A.

To prove the correctness of the algorithm, we need to show 3 steps:

Initialization: Prior to the iteration of for loop in step (12), $k = p \Rightarrow A[p \cdots k-1] = \phi$. Since $k-1 = p-1$, this array contains no elements. Since it is empty array: "the statement $A[p \cdots k-1]$ contains 0 smallest elements of L & R" is vacously true. Also, prior to the iteration of for loop, in steps (10) and (11), $i=1$ & $j=1$. So, $L[i]$ and $R[i]$ are the smallest elements of their arrays that has not been copied back to A.

Maintainance: Without loss of generality, assume $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Initially, $A[p, \cdots k-1]$ contains $k-p$ smallest elements. After $L[i]$ is copied into $A[k]$ in step (14), $A[p \cdots k]$ contain $k-p+1$ smallest elements. Incrementing k (in for loop) and i (in line 15) reestablishes the loop invariant for the next iteration. By symmetry

if $L[i] > R[j]$, then lines 16-17 perform appropriate action to maintain the loop invariant.

Termination : At termination $k = r+1$. By the loop invariant, the subarray $A[p \ldots k-1] = A[p \ldots r]$ contains $k-p = r-p+1$ smallest elements of $L[1 \ldots n_1+1]$ and $R[1 \ldots n_2+1]$ in sorted order. Since arrays L & R together contain

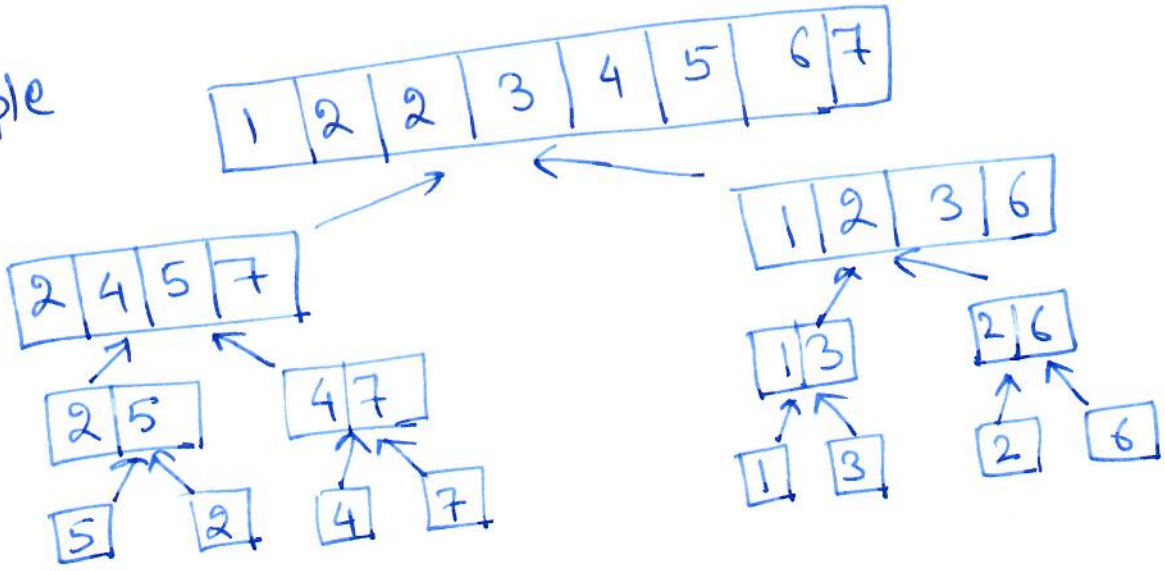
$$n_1 + n_2 + 2 = r - p + 3 \quad \longrightarrow (*)$$

elements. Array $A[p \ldots r]$ has $r-p+1$ elements. So, from (*) 2 elements, the largest ones have not been copied into A. We know that these two largest ones are sentinals ∞.

MERGE-SORT (A, p, r)

1. if $p < r$
2. $q = \lfloor (p+r)/2 \rfloor$
3. MERGE-SORT (A, p, q)
4. MERGE-SORT (A, q+1, r)
5. MERGE (A, p, q, r)

Example



Algorithms

$T(n)$: running time on a problem of size n

If problem size is small, i.e., $n \le c$

$$T(n) = \Theta(1) \leftarrow \text{const. time}$$

If a problem is divided into a subproblem each of size $1/b$. what is a and b for merge sort?

$T(n/b)$: time to solve subproblem of size n/b

$aT(n/b)$: time to solve a number of subproblems

$c(n)$: time to combine solutions of subprobs into the sol^n of original problem.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

For merge-sort: ① assume original problem size is a power of 2

② each divide step yields two subproblems of size $n/2$.

Divide: this step computes the middle of the subarray which takes const. time

$$D(n) = \Theta(1)$$

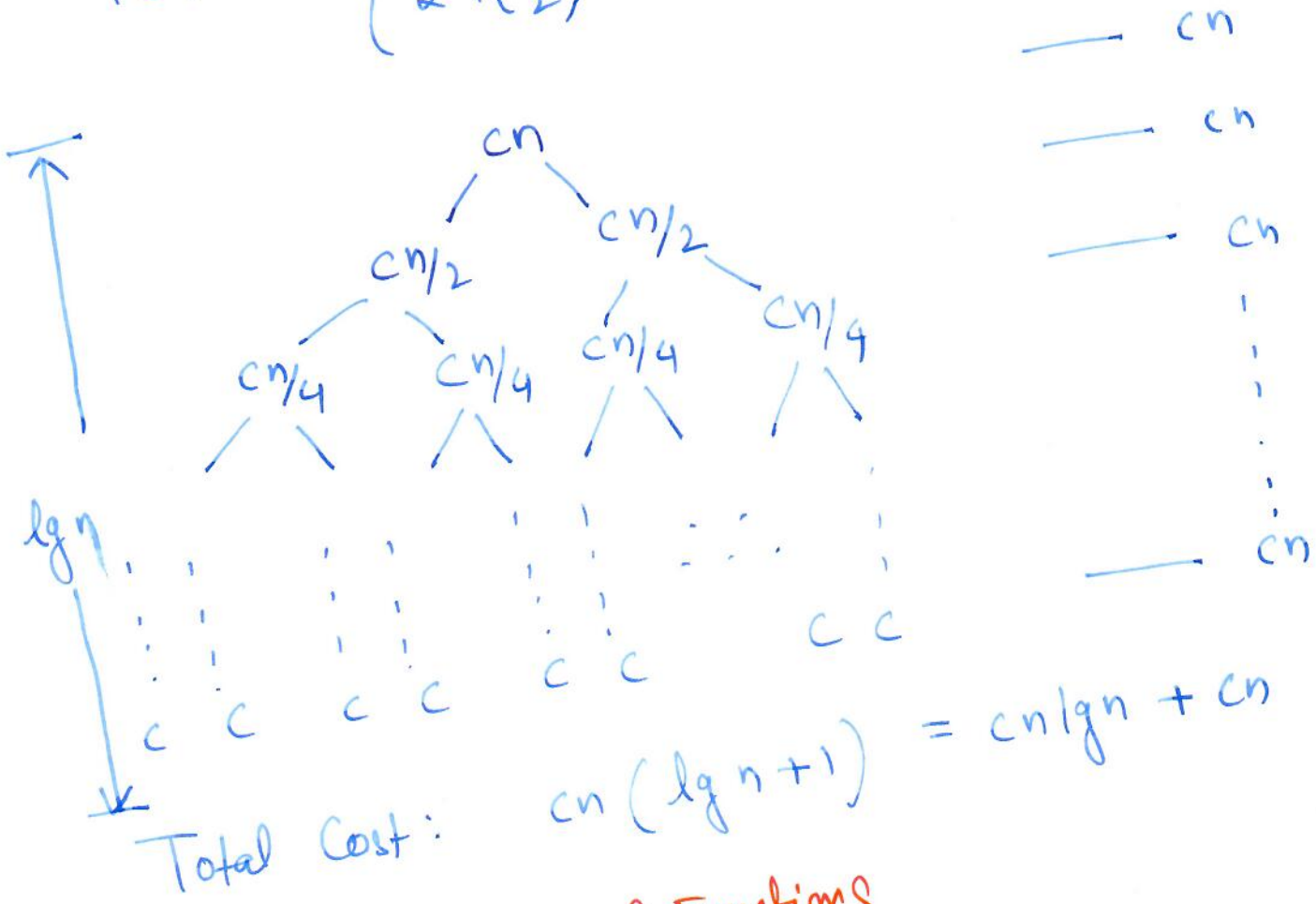
Conquer: Solve two subprobs of size $n/2$, which contributes $2T(n/2)$

Combine: Merge on n elements takes $\Theta(n)$ time

$$C(n) = \Theta(n)$$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n>1 \end{cases}$$

$$T(n) = \begin{cases} c & \text{if } n=1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{if } n>1 \end{cases}$$



Growth of Functions

Asymptotic efficiency: Consider the orders of growth of the running time for large input sizes. That is, we are concerned with how the running time of an algorithm increases with input size, as the size of the input increases without bound or as size tends to ∞.

Asymptotic Notation

Sometimes we are interested in worst case running time, but we wish to characterize the running time no matter what the input.

Θ-notation

$\Theta(g(n)) = \{ f(n)$: there exists positive constants $c_1, c_2,$ and n_0 st. $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$

* If there is a function $h(n) \in \Theta(g(n))$, then sometimes we write

$$h(n) = \Theta(g(n))$$

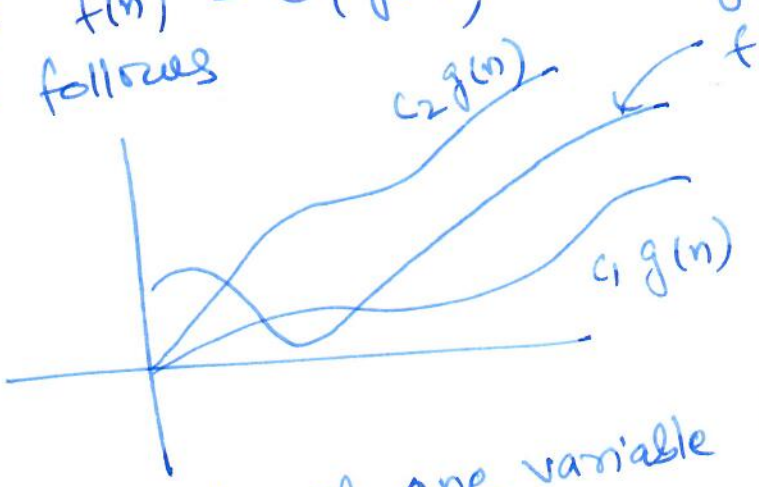
Note that here by $=$ we mean $\in$.

* A function $h(n) = \Theta(g(n))$ if $h(n)$ can be sandwitched between $c_1 g(n)$ and $c_2 g(n)$ c_1, c_2 constants for sufficiently large n $(n \ge n_0)$

* Since the function lying in $\Theta(g(n))$ are indicative of run time, they are assumed to be ≥ 0, hence the lower bound is 0.

Run time < 0 is absurd.

(*) If $f(n) = \Theta(g(n))$ the graphs may look as follows



(*) A fn of one variable is linear if for $\alpha, \beta \in \mathbb{R}$

~~$f(\alpha n) = \alpha f(n)$~~

$f(\alpha n + \beta m) = \alpha f(n) + \beta f(m)$ $\quad m, n \in \mathbb{N}$.

Example: $f(n) = an + b$, $a, b \in \mathbb{R}$ is a linear function.

Let $\alpha, \beta \in \mathbb{R}$, then

~~$f(\alpha n) = a(\alpha n) + b = \alpha$~~

$f(\alpha n + \beta m) = a(\alpha n + \beta m) + b$

$= \alpha an + \beta am + b$

$= \alpha f(n) + \beta f(n)$

* A non-linear fn may have the same order of growth as constant fn. For eg. $\sin(n)$ and constant fn c are in $\Theta(1)$.

* In the def^n of $\Theta(g(n))$, $f(n)$ and $g(n)$ are allowed to be negative, but they should be +ve after some n_0 onwards, i.e., they should be ≥ 0 for $n \geq n_0$.

Examples

$$\frac{1}{2}n^2 - 3n = \Theta(n^2) \longrightarrow (*)$$

For (*) above to be true, we must find c_1, c_2 constants and n_0 s.t $\forall n \geq n_0$

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

Dividing by n^2,

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

$c_1 \leq \frac{1}{2} - \frac{3}{n}$ holds for $n \geq 7$ & $c_1 \leq \frac{1}{14}$

$\frac{1}{2} - \frac{3}{n} \leq c_2$ holds for $n \geq 1$ and $c_2 \geq 1/2$.

So, choosing $c_1 = 1/14$, $c_2 = 1/2$ and $n_0 = \min(1, 7) = 1$,

(*) holds.

(*) Note that a different fn $f(n) = \frac{1}{4}n^2 - 3n$ would require different constants.

Example $6n^3 \neq \Theta(n^2)$

Ans. On the contrary, let us assume that $\exists$ constants c_1 and c_2 s.t.

$$c_1 n^2 \leq 6n^3 \leq c_2 n^2$$

Dividing by n^3 on both sides:

$$\frac{c_1}{n} \leq 6 \leq \frac{c_2}{n}$$

The right inequality is

$$6 \leq \frac{c_2}{n}$$

is difficult to satisfy, because whatever constant $c_2 > 0$ we choose as $n \to \infty$ $\frac{c_2}{n} \to 0$, hence $6 \leq \frac{c_2}{n}$ for n large enough is violated.

Hence $6n^3 \neq \Theta(n^2)$.

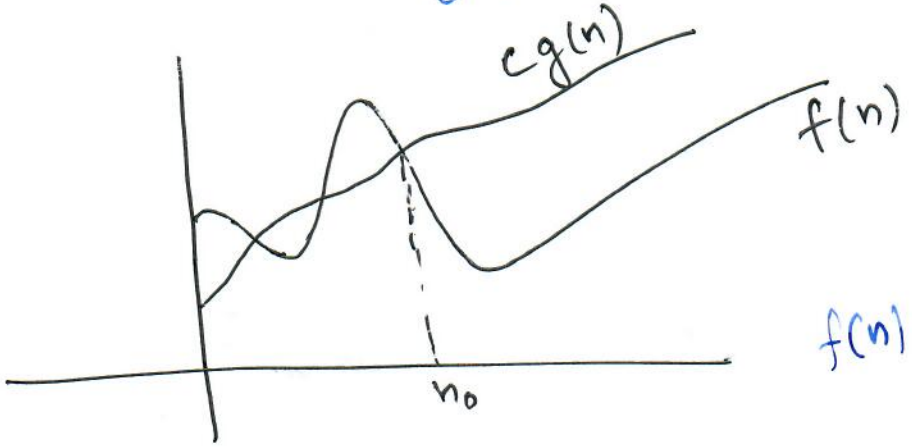
Note that we did not bother checking left ineq.

O-Notation

In contrast to Θ notation, which asymptotically bounds a function from above and below, O-notation is used to provide only the asympt. upper bound.

Given $g(n)$, denote $O(g(n))$ read as "big-Oh of g of n".

$$O(g(n)) = \{ f(n) : \exists\ c \text{ and } n_0 > 0 \text{ s.t. } 0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0 \}$$



$f(n) = O(g(n))$.

Q If $f(n) = \Theta(g(n))$, can write $f(n) = O(g(n))$?

* Θ notation is "stronger" than O-notation.

$$\Theta(g(n)) \subseteq O(g(n))$$

* Any quadratic fn an^2+bn+c, $a>0$ is in $\Theta(n^2)$ is also in $O(n^2)$.

Q Let $f(n) = an+b$. Is $f(n) \in O(n^2)$?

Remark: ① Sometimes in literature O- notation is used in place of Θ-notation. For example

$$2+3n^2 = O(n^2)$$

but more precisely

$$2+3n^2 = \Theta(n^2).$$

② Also when proving estimates, we look for tight bounds, for example,

$$2+3n^2 = O(n^3)$$

is also correct, but it is not a tight bound.
One should try to prove

$$2+3n^2 = O(n^2).$$

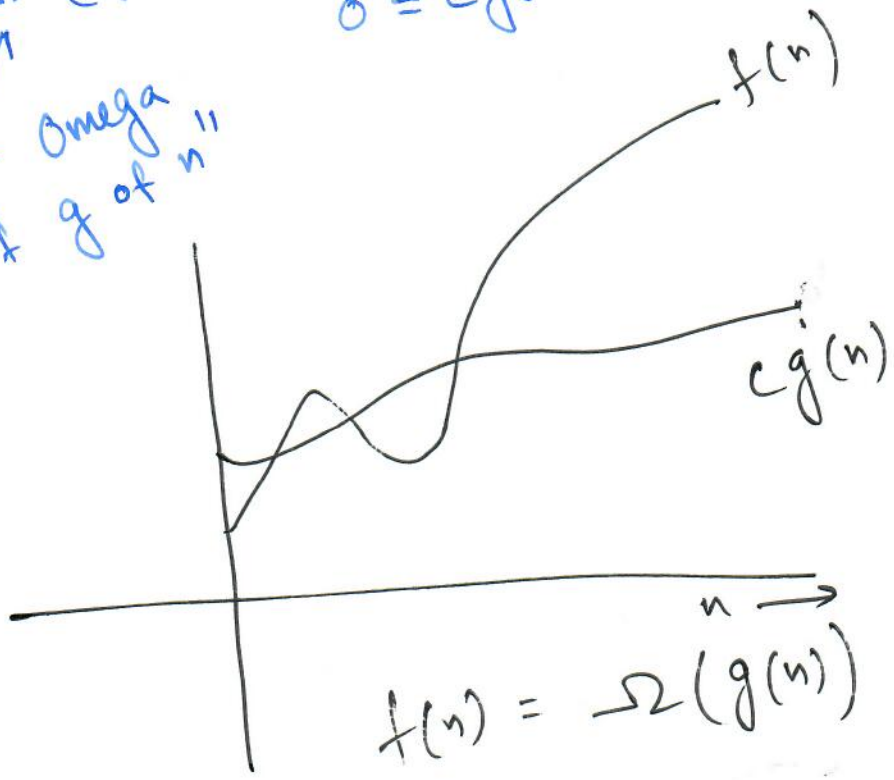
③ The $\Theta(n^2)$ bound on worst-case running time of insertion sort does not imply $\Theta(n^2)$ bound on every input. However, it is valid to say that insertion sort is $O(n^2)$ for any input.

Ω - Notation

In contrast to O-notation that requires upper bound, Ω-notation provides/requires asymptotic lower bound.

$$\Omega(g(n)) = \{ f(n) : \exists \text{ constants } c \text{ and } n_0 \text{ s.t. } 0 \le cg(n) \le f(n) \quad \forall n \ge n_0 \}$$

Read: "big-omega of g of n"



Theorem

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Proof: Since $f(n) = O(g(n))$

$\Rightarrow f(n) \le c_2 g(n) \quad \forall n \ge n_1$

Since $f(n) = \Omega(g(n))$

$\Rightarrow c_1 g(n) \le f(n) \quad \forall n \ge n_2$

Choose $n_0 = \max(n_1, n_2)$

$\Rightarrow c_1 g(n) \le f(n) \le c_2 g(n) \quad \forall n \ge n_0$

$\Rightarrow f(n) = \Theta(g(n))$.

The other way proof is trivial. ∎

Remarks

① Best case running time of insertion sort is $\Omega(n)$.

② Worst case running time of insertion sort is $\Omega(n^2)$.

o-Notation

$o(g(n)) = \{ f(n)$: for any constant $c > 0$, $\exists$ a constant $n_0 > 0$ s.t. $0 \le f(n) < c g(n)$ $\forall n \ge n_0 \}$

Remark: Big O i.e $O(\cdot)$ and small o, i.e., $o(\cdot)$ definitions look similar except that

in $f(n) = O(g(n))$

$0 \le f(n) \le cg(n)$ holds for some constant $c > 0$,

but in $f(n) = o(g(n))$

$0 \le f(n) < cg(n)$

holds for all constants $c > 0$.

For example

$2n = O(n^2)$ but $2n^2 \neq o(n^2)$

because for $2n^2 = o(n^2)$ we must have

$2n^2 \le cn^2$ for any constant c.

Now if we choose $c = 1$, then clearly

$2n^2 \nleq n^2$

ω- Notation

$\omega(g(n)) = \{ f(n) :$ for any constant $c > 0$,

$\exists$ a constant $n_0 > 0$ s.t. $0 \le cg(n) < f(n)$

$\forall n \ge n_0 \}$

Example $\frac{n^2}{2} = \omega(n)$ but $\frac{n^2}{2} \neq \omega(n^2)$

Theorem $f(n) \in \omega(g(n))$ iff $g(n) \in o(f(n))$

Claim: $f(n) \in \omega(g(n)) \Rightarrow g(n) \in o(f(n))$

Since $f(n) \in \omega(g(n))$

$\Rightarrow cg(n) \leq f(n)$ for any constant $c > 0$ for some n_0, $\forall n \geq n_0$.

$\Rightarrow g(n) < \frac{1}{c} f(n)$ [choose $c' = 1/c$]

$= c' f(n)$

$\Rightarrow g(n) < c' f(n)$ [Note since c is any constant, c' is any constant]

$\Rightarrow g(n) \in o(f(n))$

The other way proof is similar. ∎

Comparing Functions

1) If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.

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

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

4) If $f(n) = o(g(n))$ and $g(n) = o(h(n))$ then $f(n) = o(h(n))$.

5) If $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$, then $f(n) = \omega(h(n))$.

Following reflexivity and symmetry properties are satisfied:

Reflexivity:

$f(n) = \Theta(f(n))$

$f(n) = O(f(n))$

$f(n) = \Omega(f(n))$

Symmetry: $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$.

Transpose symmetry:

$f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

$f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$

- $f(n) = O(g(n))$ is like $a \leq b$
- $f(n) = \Omega(g(n))$ is like $a \geq b$
- $f(n) = \Theta(g(n))$ is like $a = b$
- $f(n) = o(g(n))$ is like $a < b$
- $f(n) = \omega(g(n))$ is like $a > b$.

Trichotomy: For any two real numbers a and b, exactly one of the following holds:

$a < b$ or
$a = b$ or
$a > b$.

For two functions $f(n)$ and $g(n)$, neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds.
For example, n and $n^{1+\sin n}$ can't be compared.

Recurrences: An equation or inequality that describes a fn in terms of its value on smaller inputs.

For example: For Merge sort

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

Here the running time $T(n)$ is a recurrence.
The solution is

$$T(n) = \Theta(n \lg n) \qquad [\lg = \log_2]$$

If the divided subproblem are of unequal sizes 2/3 to 1/3 split, and divide and Combine takes linear time, then

$$T(n) = T(2n/3) + T(n/3) + \Theta(n)$$

$\Theta(n)$: cost of combine.

Similar recurrences arise when estimating the run time of other algorithms. We need to know how to solve these recurrences.

Methods to Solve Recurrences

Substitution Method: Guess a bound and use induction to prove that the guess is correct.

Example 1. Solve the following recurrence:

$$T(n) = T(n/2) + T(\sqrt{n}) + n$$

Clearly, $T(n) = \Omega(n)$.

We guess $T(n) = \Theta(n)$, and verify by substitution method. Since $T(n) = \Omega(n)$ is trivial, we assume $T(n) = O(n)$, i.e.,

$$T(n) \le Cn, \quad C \ge 4.$$

we have

$$\begin{aligned} T(n) &= T(n/2) + T(\sqrt{n}) + n \\ &\le Cn/2 + C\sqrt{n} + n \\ &= \left(\frac{C}{2} + 1\right)n + C\sqrt{n} \\ &\le \left(\frac{C}{2} + 1\right)n + \frac{C}{4}n, \quad \text{for } n \text{ large enough.} \\ &= \left(\frac{C}{2} + \frac{C}{4} + 1\right)n \end{aligned}$$

$\leq \left(\frac{c}{2} + \frac{c}{4} + \frac{c}{4}\right) n$ (since $1 \leq c/4$)

$\leq cn$

$= O(n)$.

Example 2 Use substitution method to solve

$$T(n) = T(n/2) + T(n/4) + n$$

A From the 1st recursive call we have at least

$$T(n) = \Omega(n).$$

We guess that $T(n) = \Theta(n)$. Since, we already have $T(n) = \Omega(n)$, let us assume $T(n) = O(n)$, i.e., $T(n) \leq cn$, with $c > 1, 4$. [Usually you get idea about the suitable constant, in this case $c > 1, 4$ during derivation of $T(n) = O(n)$.]

For $1 \leq n < 4$ we have

$$T(n) = \Theta(1) \leq cn$$

For $n > 1, 4$

$$1 \leq \frac{n}{2} < n$$

$$1 \leq \frac{n}{4} < n$$

By induction

$$T\left(\frac{n}{2}\right) \leq c\frac{n}{2} \text{ and } T\left(\frac{n}{4}\right) \leq c\frac{n}{4}$$

$$\Rightarrow T(n) = \left(\frac{c}{2} + \frac{c}{4} + 1\right) n = \left(\frac{3}{4}c + 1\right) n \leq cn$$

(as $c > 1, 4$)

[At this point you decide on suitable

Note: In proof by substitution method, you assume $T(n) \leq cn$

Then if $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + n$ as in prev. example, then you prove

$T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + n \leq cn$ for <u>same constant c</u>. However, you are allowed to pick this constant.

For example, by starting with the assumpt. $T(n) \leq cn$, you prove

$T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + n \leq c'n$ → (*)

where $c' \neq c$, then the proof may be incorrect.

For example if indeed $c' \leq c$, then clearly

$T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + n \leq c'n < cn$.

But if $c' > c$, then proving (*) above <u>does not</u> show that $T(n) = O(n)$!

Recursion Tree Method

Convert the recursion into a tree whose nodes represent the costs incurred at various levels of the recursion.

Idea: Write a recursion tree, each node represents the cost of a single subproblem somewhere in the set of recursive function. Sum the costs at each level of the tree to obtain a set of pre-level costs, the sum of all the pre-level costs determine the total cost of all levels.

Note: ① A recursion tree can used to generate a good guess for substitution method

② Recursion tree can be used as a direct proof. (Eg. Merge sort).

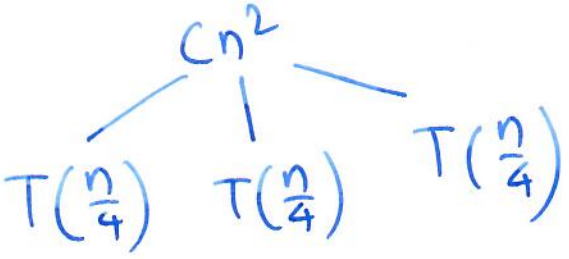
Example $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$.

$\lfloor \frac{n}{4} \rfloor \approx \frac{n}{4}$ asymptotically.

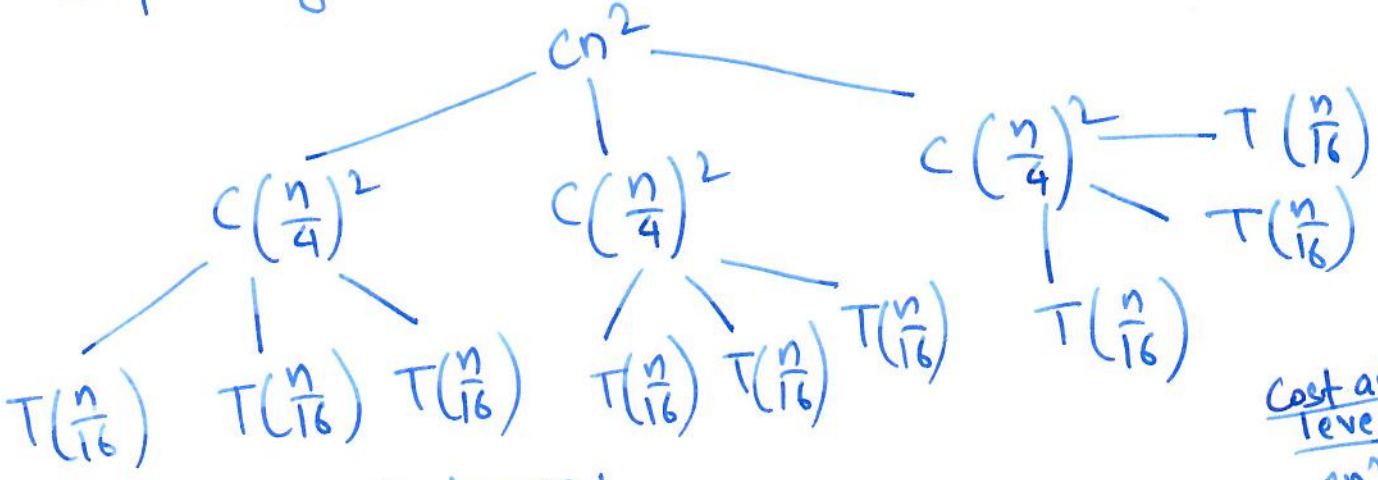
Consider recursion tree for

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2, \quad c > 0$$

Assum n is an exact power of 4.



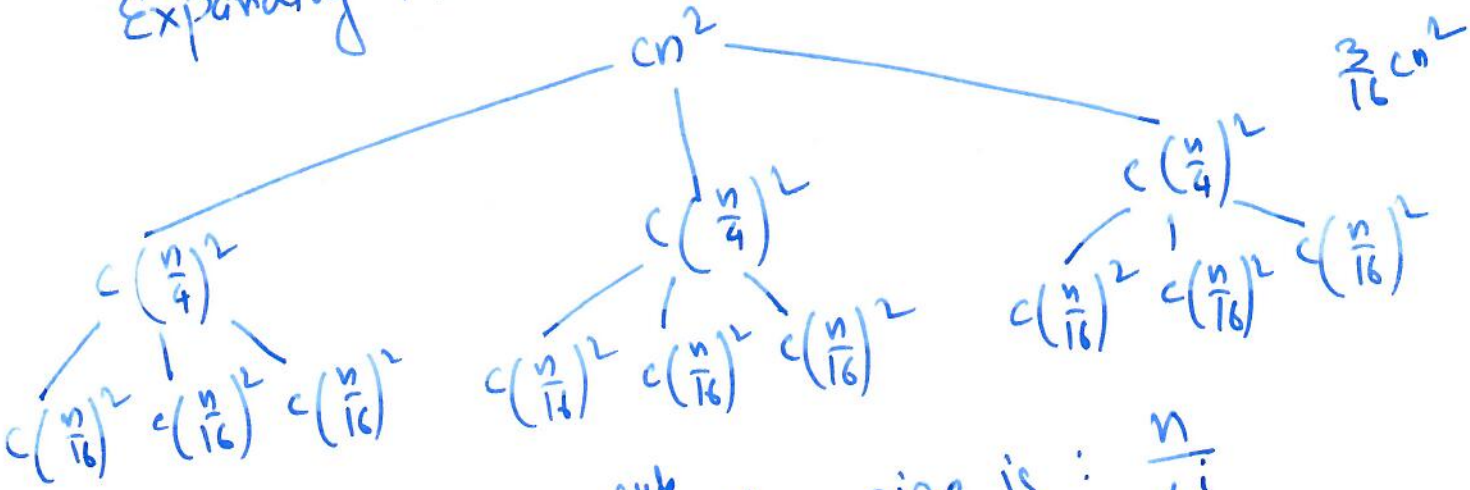
Expanding the leaves:



Cost at level

cn^2

Expanding the leaves:



At depth i, the sub problem size is : $\frac{n}{4^i}$

At the last level (bottom most level of tree), the subproblem size is 1. Hence to find height:

$$\frac{n}{4^i} = 1$$

$$\Rightarrow n = 4^i \quad \Rightarrow \log_4 n = i$$

where i is the height of the recursion tree

Number of levels = height + 1

$= \log_4 n + 1$

Levels are: $0, 1, 2, \ldots, \log_4 n$.

Number of nodes at depth $i = 3^i$.

Each node at depth i, for $i = 0, 1, \ldots \log_4 n - 1$ has cost: $c\left(\frac{n}{4^i}\right)^2$

Total cost over all nodes at depth i: $3^i c\left(\frac{n}{4^i}\right)^2$

$= \left(\frac{3}{16}\right)^i cn^2$

Number of nodes at bottom level: $3^{\log_4 n}$

$= n^{\log_4 3}$

Total cost of leaf nodes: $n^{\log_4 3} T(1)$

$= \Theta\left(n^{\log_4 3}\right)$.

All the costs:

$$T(n) = cn^2 + \frac{3}{16} cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta\left(n^{\log_4 3}\right).$$

$$= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right)$$

$$= \frac{\left(\frac{3}{16}\right)^{\log_4 n} - 1}{\left(\frac{3}{16}\right) - 1} cn^2 + \Theta\left(n^{\log_4 3}\right)$$

$$\leq \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right)$$

$$= \frac{1}{1 - \left(\frac{3}{16}\right)} cn^2 + \Theta\left(n^{\log_4 3}\right)$$

$$= \frac{16}{13} cn^2 + \Theta\left(n^{\log_4 3}\right)$$

$$= O(n^2).$$

First recursive call gives $\Theta(n^2) \Rightarrow T(n)$ is atleast $\Omega(n^2)$ & from above $O(n^2)$

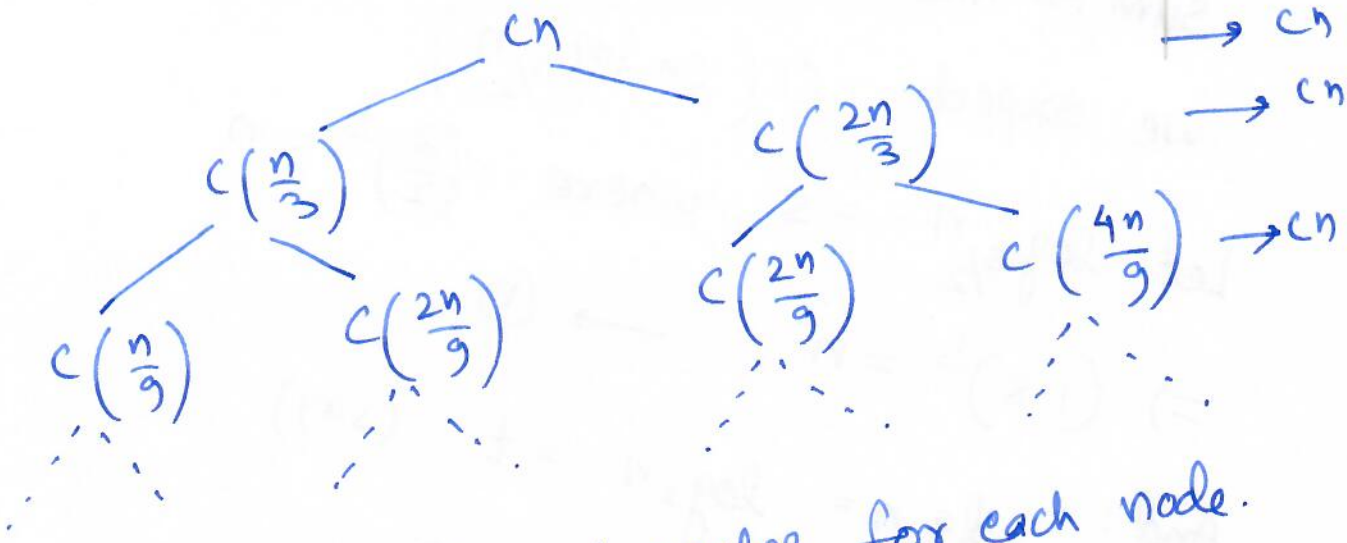
$\Rightarrow T(n) = \Theta(n^2)$.

Example 2 (use recursion tree to get a good guess for substitution method)

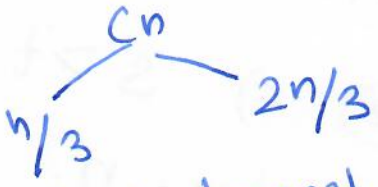
A. we first look at recursion tree for

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn$$

then get a good guess to solve the above recurrence by substitution method.



Q There are two branches for each node.



which will lead to longest path from root?

A $n \to \left(\frac{2}{3}\right)n \to \left(\frac{2}{3}\right)^2 n \to \dots \to 1.$

Q What is the height of the tree assuming leaf nodes are of size 1?

A. At level k, each subproblem size is: $\left(\frac{2}{3}\right)^k n$.

At leaf subproblem size is 1, so, to get the height:

$$\left(\frac{2}{3}\right)^k n = 1$$

$$\Rightarrow n = \left(\frac{3}{2}\right)^k$$

$$\Rightarrow \log_{3/2} n = k.$$

Sum of the cost at each level is cn.

we expect $O(cn \log_{3/2} n)$

Let $\log_{3/2} n = s$, where $\left(\frac{3}{2}\right)^s = n$

$\Rightarrow (1.5)^s = n \longrightarrow (1)$

And: $\lg n = \log_2 n = t$ (say)

$\Rightarrow 2^t = n \longrightarrow (2)$

From (1) and (2) $\Rightarrow s > t$

$\Rightarrow n \lg n \leq n \log_{3/2} n.$

(*) If the recursion tree were complete binary tree of height $\log_{3/2} n$, there would be $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$ nodes at bottom level. Nodes at bottom level are also called leaves.

(*) Cost of each leaf is constant $\Rightarrow$ total cost of Leaves: $\Theta(n^{\log_{3/2} 2})$

(*) If, however, tree is not complete binary tree, then levels in the bottom contribute $< cn$.

(*) A good guess for the substitution method is $O(n \lg n)$

Now we use substitution method to verify that $O(n \lg n)$ is an upper bound. That is we claim the following:

Claim: $T(n) \le dn \lg n$, d constant

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn$$

$$\le d\left(\frac{n}{3}\right)\lg\left(\frac{n}{3}\right) + d\left(\frac{2n}{3}\right)\lg\left(\frac{2n}{3}\right) + cn$$

$$= \left(d\left(\frac{n}{3}\right)\lg n - d\left(\frac{n}{3}\right)\lg 3\right) + cn$$

$$+ \left(d\left(\frac{2n}{3}\right)\lg n - d\left(\frac{2n}{3}\right)\lg\left(\frac{3}{2}\right)\right)$$

$$= dn\lg n - d\left(\left(\frac{n}{3}\right)\lg 3 + \left(\frac{2n}{3}\right)\lg\left(\frac{3}{2}\right)\right) + cn$$

$$= dn\lg n - d\left(\left(\frac{n}{3}\right)\lg 3 + \left(\frac{2n}{3}\right)\lg 3 - \left(\frac{2n}{3}\right)\lg 2\right) + cn.$$

$$= dn\lg n - dn\left(\lg 3 - \frac{2}{3}\right) + cn$$

$$\le dn\lg n.$$

Base condition and substitution method

Let us assume that we made a guess that $T(n) \le cn\log n$, i.e, $T(n) = O(n\log n)$.

But if $T(1) = 1$ is given as base condn.

we have for $n=1$, $T(n) \le cn\lg n$

$\Rightarrow T(1) \le c \cdot 1 \lg 1 = 0$

$\Rightarrow 1 \le 0$, a contradiction.

But we need to prove:

$T(n) \le cn \lg n$ for $n \ge n_0$.

From $T(1) = 1 \Rightarrow T(2) = 2T(\lfloor 2/2 \rfloor) + 2$

$= 2T(1) + 2 = 4$

and $T(3) = 5$, By choosing c large enough

$\Rightarrow T(2) \le c\, 2 \lg 2$

$T(3) \le c\, 3 \lg 3$.

In this case, $c \ge 2$ suffices for $n = 2$ & $n = 3$

and so on to hold.

Remark: For most recurrence, it is straightforw
-ard to extend boundary conditions to make
inductive hypothesis work for small n.

Making a good guess for substitution method

(*) There is no general method to make a guess

(*) Experience and creativity required

Example: $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$

For this: $T(n) = cn \lg n$ is again a good gues

Dealing with off-by-constant in substitution method

Consider $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$

Guess: $T(n) = O(n)$

i.e., $T(n) \le cn$, c some const.

$T(n) \le c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1$

$= cn + 1 \not\le cn$ for any c. $\rightarrow$ (*)

↑ off by one! :-c

Temptation to try larger guess $O(n^2)$?

Actually $T(n) = O(n)$ is correct.

To fix off-by-constant problem in (*), consider new guess:

$T(n) \le cn - d$, $d \ge 0$

$T(n) \le (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1$

$= cn - 2d + 1$

$\le cn - d$, $d \ge 1$.

Solving recurrence by change of variables

Consider: $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$

looks difficult to solve. Consider change of variable: $m = \lg n$

$\Rightarrow T(2^m) = 2T(2^{m/2}) + m$

Rename: $S(m) = T(2^m)$

$\Rightarrow S(m) = 2\left(S\left(\frac{m}{2}\right)\right) + m$

Solution to this recurrence is:

$S(m) = O(m \lg m)$.

Now changing from $S(m)$ to $T(n)$:

$T(n) = T(2^m) = S(m) = O(m \lg m)$

$= O(\lg n \lg \lg n)$.

Master method for Solving Recurrences

(*) cookbook for solving

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$, $a \geq 1$, $b > 1$ const.

(*) $f(n)$ asymp. positive fn.

Remark: (1) Need to memorize 3 cases

(2) $\frac{n}{b}$ may not be integers, then replace it with $\lfloor n/b \rfloor$ and $\lceil n/b \rceil$.

Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by recurrence:

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$,

then $T(n)$ has following asympt. bounds:

1) If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some const. $\epsilon > 0$, then

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

2) If $f(n) = \Theta\left(n^{\log_b a}\right)$, then

$$T(n) = \Theta\left(n^{\log_b a} \lg n\right)$$

3) If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some const. $\epsilon > 0$, and if

$$a f\left(\frac{n}{b}\right) \le c f(n)$$

for some $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Remarks: For the case 1) above

$$f(n) = O\left(n^{\log_b a - \epsilon}\right)$$

$$\Rightarrow f(n) \le C n^{\log_b a - \epsilon}$$

$$= c n^{\log_b a} \cdot n^{-\epsilon} = \frac{c n^{\log_b a}}{n^{\epsilon}} \longrightarrow (*)$$

Since there is a polynomial factor n^{ϵ} that is divided on R.H.S of (*), we say that $f(n)$ is "polynomially smaller" than $n^{\log_b a}$.

In other words, $f(n)$ is asymptotically smaller than $n^{\log_b a}$ by a factor n^{ϵ} for some $\epsilon > 0$.

② Similarly, for the case 3) of master thm, $f(n)$ is polynomially larger than $n^{\log_b a}$, additionally, $af\left(\frac{n}{b}\right) \le cf(n)$ must be sat.

Cases where master theorem can't be applied

① $f(n)$ is smaller than $n^{\log_b a}$ but not polynomially smaller

② $f(n)$ is larger than $n^{\log_b a}$ but not polynomially larger

③ Regularity condition $af\left(\frac{n}{b}\right) \le cf(n)$ does not hold in 3rd case of master thm.

Examples of Master Theorem

Example 1

$T(n) = 9T\left(\frac{n}{3}\right) + n$

Here $a = 9$, $b = 3$, $f(n) = n$

we have $n^{\log_b a} = n^{\log_3 9} = n^{\log_3 3^2} = n^2$

$= \Theta(n^2)$

Since $f(n) = n = \Theta\left(n^{\log_3 9 - \epsilon}\right)$, $\epsilon = 1$.

we apply case 1) of master thm.

$\Rightarrow T(n) = \Theta(n^2)$.

Example 2

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

$a = 1, \quad b = \frac{3}{2}, \quad f(n) = 1.$

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1 = f(n)$$

i.e., $f(n) = \Theta(n^{\log_b a})$

$\Rightarrow$ Case 2) of master thm applies:

$$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(\lg n).$$

Example 3

$$T(n) = 3T\left(\frac{n}{4}\right) + n \lg n,$$

$a = 3, \quad b = 4, \quad f(n) = n \lg n$

$$n^{\log_b a} = n^{\log_4 3} = n^{0.793} \quad \left[\because 4^{0.793} = 3\right]$$

we have:

$$n^{\log_4 3 + \epsilon} = n^{\epsilon} \cdot n^{\log_4 3} = n^{\epsilon} \cdot n^{0.793}$$

Choosing $\epsilon = 0.2$

$$n^{\log_4 3 + \epsilon} = n^{0.993} < n < n \lg n.$$

$\Rightarrow f(n) = n \lg n = \Omega(n^{\log_4 3 + \epsilon})$

Case 3) of master thm applies, provided the regularity condn holds, we have

$$af\left(\frac{n}{b}\right) = 3\left(\frac{n}{4}\right) \lg\left(\frac{n}{4}\right) \leq \frac{3}{4} n \lg n$$

$$= cf(n), \quad c = 3/4.$$

Hence, from case 3) of master thm.

$$T(n) = \Theta(n \lg n).$$

Example where master thm does not apply

$$T(n) = 2T\left(\frac{n}{2}\right) + n \lg n.$$

Here: $a = 2, \; b = 2, \; f(n) = n \lg n.$

$$n^{\log_b a} = n^{\log_2 2} = n^1 = n.$$

Here $f(n) = n \lg n \geq n = n^{\log_b a}.$

Since $\lg n$ is not polynomially larger, i.e.,

$\lg n \not\geq n^{\epsilon}$ for any $\epsilon > 0$,

we have the case that $f(n)$ is not polynomially larger than $n^{\log_b a}$.

Matrix Multiplication and Recurrences

Let $A = (a_{ij}), \; B = (b_{ij})$ be square $n \times n$ matrices

$C = A \cdot B$ ↑ multiplication of A & B.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

There are n^2 matrix entries in C; each is sum of n values.

Square-Matrix-Multiply (A, B)

```
1.  n = A.rows
2.  Let C be a new matrix
3.  for i = 1 to n
4.      for j = 1 to n
5.          cij = 0
6.          for k = 1 to n
7.              cij = cij + aik bkj
8.  return C.
```

Cost: Each of the triply nested for loops runs exactly n iterations, and each execution of line 7 takes const. time. Hence total cost: $\Theta(n^3)$.

Block Matrix Multiplication

(*) Assume n is a power of 2

(*) Partition matrices A, B, and C into four $\frac{n}{2} \times \frac{n}{2}$ matrices.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

→ (*)

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Square Matrix Multiply Recursive (A, B)

square_matrix_multiply_recursive(A, B)

1. n = A.rows
2. Let C be a new nxn matrix
3. if n == 1
4. $c_{11} = a_{11} \cdot b_{11}$
5. else partition A, B, and C as in ⊛
6. C_{11} = square_matrix_multiply_recursive(A_{11}, B_{11}) + square_matrix_multiply_recursive(A_{12}, B_{21})

   C_{12} = square_matrix_multiply_recursive(A_{11}, B_{12}) + square_matrix_multiply_recursive(A_{12}, B_{22})

   C_{21} = square_matrix_multiply_recursive(A_{21}, B_{11}) + square_matrix_multiply_recursive(A_{22}, B_{21})

   C_{22} = square_matrix_multiply_recursive(A_{21}, B_{12}) + square_matrix_multiply_recursive(A_{22}, B_{22})
7. return C

Cost Analysis of Block Matrix-Multiply

In base case (bottom level of recursion): From step ④ of algo, we perform one scalar mult.

$$T(1) = \Theta(1)$$

Recursive case: (when $n > 1$)

Cost of partitioning matrices: $\Theta(1)$ (to find the indices of rows & columns)

In Line 6: square_matrix_multiply_recursive( ) is called 8 times for $\frac{n}{2} \times \frac{n}{2}$ matrices.

Hence total time: $8T\left(\frac{n}{2}\right)$

In Line 6: 4 matrix additions of matrices with $\frac{n}{2} \times \frac{n}{2}$ sizes.

$\Rightarrow$ Cost of 4 matrix additions: $\Theta(n^2)$

$$T(n) = \Theta(1) + 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

$$= 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Total Cost (base + recursive)

$$T(n) = \begin{cases} \Theta(1), & \text{if } n=1, \\ 8\,T\left(\frac{n}{2}\right) + \Theta(n^2), & \text{if } n>1. \end{cases}$$

Using Master Method: $T(n) = \Theta(n^3)$.

Strassen's Matrix Multiplication

Idea: Instead of 8 recursive multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices, perform only 7, but pay for several new additions of $\frac{n}{2} \times \frac{n}{2}$ matrices, but constant no. of additional additions. Hence, total cost is less than vanila matrix multiplication.

Method: 1. Divide A, B, and C into $\frac{n}{2} \times \frac{n}{2}$ submatrices. This take $\Theta(1)$ time by index calculation.

2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$ each of which is $\frac{n}{2} \times \frac{n}{2}$, which are sum or difference of two matrices in step 1. All these 10 matrices can be created in $\Theta(n^2)$ time.

3) Using submatrices created in step-1 and 10 matrices created in step 2, recursively compute 7 matrix products $P_1, P_2, \cdots, P_7$. Each matrix P_i is $\frac{n}{2} \times \frac{n}{2}$.

4) Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. All 4 matrices can be computed in $\Theta(n^2)$ time.

Recursion for Run Time of Strassen

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 7\,T\left(\frac{n}{2}\right) + \Theta(n^2) & \text{if } n > 1. \end{cases}$$

we have one-matrix mult. less compared to classical approach.

Steps in Strassen

Step 1: a) Partition the matrix A in 4 parts:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

b) Partition the matrix B into 4 parts

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}.$$

Step 2 Create the Following 10 matrices

$S_1 = B_{12} - B_{22}$

$S_2 = A_{11} + A_{12}$

$S_3 = A_{21} + A_{22}$

$S_4 = B_{21} - B_{11}$

$S_5 = A_{11} + A_{22}$

$S_6 = B_{11} + B_{22}$

$S_7 = A_{12} - A_{22}$

$S_8 = B_{21} + B_{22}$

$S_9 = A_{11} - A_{21}$

$S_{10} = B_{11} + B_{12}$

Add or subtract $n/2 \times \frac{n}{2}$ matrices: $\Theta(n^2)$.

Step 3: Create the following matrices:

$P_1 = A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22}$

$P_2 = S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22}$,

$P_3 = S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11}$

$P_4 = A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11}$

$P_5 = S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}$

$P_6 = S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}$.

$P_7 = S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}$.

Step 4

$C_{11} = P_5 + P_4 - P_2 + P_6$

$= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} - A_{22} \cdot B_{11}$
$+ A_{22} \cdot B_{21} - A_{11} \cdot B_{22} - A_{12} \cdot B_{22} - A_{22} \cdot B_{22} - A_{22} \cdot B_{21}$
$+ A_{12} \cdot B_{22} + A_{12} \cdot B_{21}$

$= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$

$C_{12} = P_1 + P_2$

$= A_{11} \cdot B_{12} - A_{11} B_{22} + A_{11} B_{22} + A_{12} \cdot B_{22}$

$= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$

$C_{21} = P_3 + P_4$

$= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} - A_{22} B_{11} + A_{22} B_{21}$

$= A_{21} B_{11} + A_{22} B_{21}$

$C_{22} = P_5 + P_1 - P_3 - P_7$

$= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} - A_{11} \cdot B_{22}$
$+ A_{11} \cdot B_{12} - A_{22} \cdot B_{11} - A_{21} \cdot B_{11} - A_{11} \cdot B_{11} - A_{11} \cdot B_{12}$
$+ A_{21} \cdot B_{11} + A_{21} \cdot B_{12} = A_{22} \cdot B_{22} + A_{21} \cdot B_{12}$

Q In a 3×3 partitioned matrix:

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix}$$

$$C = A \cdot B = \begin{bmatrix} \sum_{j=1}^{3} A_{1j} B_{j1} & \sum_{j=1}^{3} A_{1j} B_{j2} & \sum_{j=1}^{3} A_{1j} B_{j3} \\ \sum_{j=1}^{3} A_{2j} B_{j1} & \sum_{j=1}^{3} A_{2j} B_{j2} & \sum_{j=1}^{3} A_{2j} B_{j3} \\ \sum_{j=1}^{3} A_{3j} B_{j1} & \sum_{j=1}^{3} A_{3j} B_{j2} & \sum_{j=1}^{3} A_{3j} B_{j3} \end{bmatrix}$$

Total #multiplications : 27

Total #additions : 18

Design a "Strassen" like matrix multiplication with lesser no. of matrix multiplications (possibly at the cost of additional additions).

Solved Examples

Q Give a real world example that requires sorting.

A. keep track of a bunch of people's file folders and be able to look up a given name quickly.

Q Other than speed, what other measures of efficiency might one use in a real-world setting?

A One might measure memory usage: both disk and RAM, no. of mem. accesses, etc.

Q Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64 n \lg n$ steps. For which values of n does insertion sort beat merge sort?

A

$$8n^2 < 64 n \lg n$$

$$\Rightarrow n < 8 \lg n \Rightarrow n \leq 43.$$

Q What is the smallest value of n s.t. an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

A. 15

Q For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algo. to solve the problem takes $f(n)$ microsecs.

	1 sec.	1 min.	1 hr.	1 day	1 month	1 year	1 century
lgn	2^{10^6}	$2^{6\times10^7}$	$2^{3.6\times10^9}$	$2^{8.6\times10^{10}}$	$2^{2.5\times10^{12}}$	$2^{3.15\times10^{13}}$	$2^{3.1\times10^{15}}$
$\sqrt{n}$	10^{12}	3.6×10^{15}	1.3×10^{19}	7.46×10^{21}	6.72×10^{24}	9.9×10^{26}	9.9×10^{30}
n	1×10^6	6×10^7	3.6×10^9	8.6×10^{10}	2.59×10^{12}	3.15×10^{13}	3.16×10^{15}
nlgn	62746	2801417	133378058	2755147513	71870856404	797633893349	6.8×10^{13}
n^2	1000	7745	60000	293938	1609968	5615692	56176151
n^3	100	391	1532	4420	13736	31593	146679
2^n	19	25	31	36	41	44	51
n!	9	11	12	13	15	16	17

Q Rewrite insertion-sort procedure to sort into nonincreasing instead of nondecreasing order.

A.

Non increasing Insertion-Sort (A)

```
1.  for j = 2 to A.length do
2.      key = A[j]
3.      // insext A[j] into the sorted seq. A[1···j-1]
4.      i = j-1
5.      while i>0 and A[i] < key do
6.          A[i+1] = A[i]
7.          i = i-1
8.      end while
9.      A[i+1] = key
10. end for
```

Q Consides the seasching problem:

Input: A sequence of n numbers

$A = \langle a_1, a_2, \cdots, a_n \rangle$ and a value v.

Output: An index i s.t.

$v = A[i]$

or $v = NIL$ if v does not appear in A.

Write pseudocode for linear search, which scans through the seq., looking for v. Using a loop invariant, prove that your algo. is correct.

Pseudocode:

```
Linear-Search(A, v)
1.  for i ← 1 to length[A]
2.      if A[i] == v
3.          then return i
4.  return NIL
```

Loop invariant: On each iteration of the loop body the invariant upon entering is that there is no index $k < j$ so that $A[k] = v$.

Q Consides sorting n numbers ~~sorted in de~~ stored in array A by first finding the smallest element of A and exchanging it with the element in A[1]. Then find the second smallest element of A, and exchange it with A[2]. Continue in this manner for the 1st n-1 elements of A. Write a pseudocode for this algo., which is known as Selection sort. What loop invariant does this algo. maintain? Give best case and worst case running time?

A.

Selection Sort

```
1.  for i = 1 to n-1
2.      min = i
3.      for j = i+1 to n do
4.          if A[j] < A[min] then
5.              min = j
6.          end if
7.      end for
8.      swap A[min] and A[i]
9.  end for
```

Loop invariant: At each iteration of the for loop of lines 1 through 9, the subarray $A[1 \cdots i-1]$ contains $i-1$ smallest elements of A in increasing order.

Best case: $\Theta(n^2)$

Worst case: $\Theta(n^2)$

$$\sum_{i=1}^{n} n-i = n(n-1) - \sum_{i=1}^{n-1} i = n^2 - n - \frac{n^2-n}{2} = \frac{n^2-n}{2} = \Theta(n^2).$$

Q How can we modify almost any algo to have a good best-case run. time?

A. ① Check if the input is some specially chosen input for which algorithm can output precomputed result.

(2) Modify the algo. to produce an output randomly. Then check if this random output is correct. Checking that the random output is correct may be very fast.

For example: for selection sort, make the algo produce a random permutation of input array. Then check whether the output is sorted. In this case, checking that a list is sorted takes $O(n)$.

In either case: best case run time depends on "lucky" input or "lucky" output.

Q Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \\ 2T\left(\frac{n}{2}\right) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \lg n$.

A Try.

Q We can express insertion sort as a recursive procedure as follows. In order to sort $A[1 \cdots n]$ we recursively sort $A[1 \cdots n-1]$ and then insert $A[n]$ into the sorted array $A[1 \cdots n-1]$. Write a

recurrence for the run time of this method.

A. Let $T(n)$ denote the run time for insertion sort called on an array of size n.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c, \\ T(n-1) + I(n) & \text{otherwise.} \end{cases}$$

where $I(n)$ denotes the amount of time it takes to insert $A[n]$ into the sorted array $A[1 \cdots n-1]$. Since we have to shift as many as $n-1$ elements once we find the correct place to insert $A[n]$ we have $I(n) = \Theta(n)$.

Q If the input seq. A is sorted, we can check the midpoint of the sequence against value v that we search in A, and eliminate half of the seq. from further consideration. This binary search procedure is repeated on the remaining half. Show that the worst-case run time is $\Theta(\lg n)$.

A. The pseudocode follows:

BinSearch(1, n, v)

```
1. if (l > h) then
2.     return NIL
3. end if
4. m = ⌊(l+n)/2⌋
5. if A[m] = v
6.     return m
7. end if
8. if A[m] < v
9.     return BinSearch(l, m, v)
10. end if
11. return BinSearch(m+1, n, v)
```

$$T(n) = T\left(\frac{n}{2}\right) + C$$

$$\Rightarrow T(n) \in \Theta(\lg n).$$

Q The while loop 5-7 of the insertion sort uses linear search to scan through the sorted subarray $A[1 \cdots j-1]$. Can we use binary search instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

A No. Even if we obtain the location of the index i in the array where we would need to place the element $A[j]$ using binary search, we still must move all the elements of A

to insert $A[j]$ into the array $A[1 \cdots j]$. This shuffling of the elements requires at most $\Theta(j)$ work. Thus, worst case complexity of insertion sort is not changed from $\Theta(n^2)$ by using bin. search.

Q Describe a $\Theta(n \lg n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

A A brute-force approach requires picking two elements out of n elements, and checking whether they sum to x. This requires

$$\binom{n}{2} = \frac{n!}{(n-2)!\,2!} = \frac{n(n-1)}{2} = \Theta(n^2).$$

Another way:

Sort the array S by dropping duplicates. This can be done in $\Theta(n \lg n)$, for eg, using merge sort.

If $a < \frac{x}{2}$, $b < \frac{x}{2}$

$\Rightarrow a + b < x$

II^{ly} $a > x/2$, $b > \frac{x}{2}$

$\Rightarrow a + b > x$

Since there are no duplicates, if there are two nos. that sum to x, one of them must be $< \frac{x}{2}$ and other one must be $> \frac{x}{2}$.

(*) Hence, find the location where $\frac{x}{2}$ could be placed in the sorted list. This location splits sorted list into two sorted list A and B. In A, all elements are $< \frac{x}{2}$ and in B all elements are $> \frac{x}{2}$.

(*) If two elements sum to x: one comes from A and other from B.

Idea: Take each value of $a \in A$ and compute a target of $b = x - a$: if we find $b \in B$ we are done.

Cost: $|A| + |B| = \Theta(n)$.

Total cost: $\Theta(n \lg n) + \Theta(n) = \Theta(n \lg n)$.

Q Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion of A.

a) List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$

b) What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions? How many does it have?

Algorithms

A a) The five inversions are:
$(2,1)$, $(3,1)$, $(8,6)$, $(8,1)$, and $(6,1)$.

b) The n element array with most inversions is $\langle n, n-1, \ldots, 2, 1 \rangle$. It has

$$n-1 + n-2 + \cdots + 2 + 1 = \frac{n(n-1)}{2} \text{ inversions}$$

c) What is the relationship betn the run time of insertion sort and the no. of inversions in the input array?

A The run time of insertion sort is const. times the no. of inversions. Let $I(i)$ denote the number of $j < i$ s.t. $A[j] > A[i]$. Then $\sum_{i=1}^{n} I(i)$ equals the no. of inversions in A. The while loop of insert. sort will execute once for each element of A which has index less than j is larger than $A[j]$. Thus it will execute $I(j)$ times.

Q Show that for any real constants a and b, where $b > 0$,

$$(n+a)^b = \Theta(n^b)$$

A. Let $c = 2^b$ and $n_0 \geq 2a$. Then for all $n \geq n_0$ we have

$$(n+a)^b \leq \left(n + \frac{n_0}{2}\right)^b \leq \left(n + \frac{n}{2}\right)^b$$
$$\leq (n+n)^b = (2n)^b = 2^b n^b = cn^b.$$

so that
$$(n+a)^b = O(n^b). \longrightarrow (1)$$

Now let $n_0 \geq \frac{-a}{1-\frac{1}{2^{1/b}}}$ and $c = \frac{1}{2}$.

Then $n \geq n_0 \iff n - \frac{n}{2^{1/b}} \geq -a$

$$\iff n + a \geq \left(\frac{1}{2}\right)^{a/b} n$$

$$\iff (n+a)^b \geq cn^b. \longrightarrow (2)$$

$$\Rightarrow (n+a)^b = \Omega(n^b).$$

From (1) and (2)
$$(n+a)^b = \Theta(n^b).$$

Q Explain why the statement, "The running time of algorithm A is at least $O(n^2)$," is meaningless.

A. Try.

Q Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

Q Show that
$$k \ln k = \Theta(n)$$
$$\Rightarrow k = \Theta\left(n/\ln n\right).$$

Q Rank the following functions by orders of growth; that is, find an arrangement $g_1, g_2, \ldots g_{30}$ of functions satisfying $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, $\ldots$ $g_{29} = \Omega(g_{30})$. Partition your list into equivalence classes s.t. the functions $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$ $\quad 2^{\lg^* n}$ $\quad (\sqrt{2})^{\lg n}$ $\quad n^2$ $\quad n!$ $\quad (\lg n)!$

$\left(\frac{3}{2}\right)^n$ $\quad n^3$ $\quad \lg^2 n$ $\quad \lg(n!)$ $\quad 2^{2^n}$ $\quad n^{1/\lg n}$

$\ln\ln n$ $\quad \lg^* n$ $\quad n \cdot 2^n$ $\quad n^{\lg\lg n}$ $\quad \ln n$ $\quad 1$

$2^{\lg n}$ $\quad (\lg n)^{\lg n}$ $\quad e^n$ $\quad 4^{\lg n}$ $\quad (n+1)!$ $\quad \sqrt{\lg n}$

$\lg^*(\lg n)$ $\quad 2^{\sqrt{2\lg n}}$ $\quad n$ $\quad 2^n$ $\quad n \lg n$ $\quad 2^{2^{n+1}}$

A. $2^{2^{n+1}}$, 2^{2^n}, $(n+1)!$, $n!$, $n2^n$, e^n, 2^n, $\left(\frac{3}{2}\right)^n$, $(\lg(n))!$, $n^{\lg(\lg(n))}$, $\lg(n)^{\lg(n)}$, n^3, n^2, $4^{\lg n}$, $n \lg n$, $\lg n!$, $n, 2^{\lg n}$, $(\sqrt{2})^{\lg n}$, $2^{\sqrt{2\lg n}}$, $\lg^2 n$, $\ln n$, $\sqrt{\lg n}$, $\sqrt{\lg n}$, $\ln(\ln n)$, $2^{\lg^* n}$, $\lg^* n$, $\lg^*(\lg n)$, $\lg(\lg^* n)$, 1, $n^{1/\lg n}$.

Q Use Strassen's algo to compute the matrix product.

$$\begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix} \begin{bmatrix} 6 & 8 \\ 4 & 2 \end{bmatrix}.$$

Show your work.

A Try

Q Show how to multiply the complex numbers $a+bi$ and $c+di$ using only three multiplications of real numbers. The algorithm should take a, b, c, and d as input and produce the real component $ac-bd$ and the imaginary component $ad+bc$

A
$$P_1 = (a+b)\,c = ac+bc$$
$$P_2 = b(c+d) = bc+bd$$
$$P_3 = (a-b)\,d = ad+bd$$

Real part: $P_1 - P_2$

Imag. part: $P_2 + P_3$

Q Using the master method, show that the solution to the recurrence

$$T(n) = 4T\left(\frac{n}{3}\right) + n$$

is $T(n) = \Theta\left(n^{\log_3 4}\right)$.

Show that a substitution proof with the assumpt. $T(n) \le cn^{\log_3 4}$ fails. Then show how to subtract off a lower-order term to make a subst. proof work.

A Trying $T(n) \le cn^{\log_3 4}$

$$T(n) = 4T\left(\frac{n}{3}\right) + n \le 4c\left(\frac{n}{3}\right)^{\log_3 4} + n$$
$$= 4cn^{\log_3 4} + n$$

Clearly $T(n) \le cn^{\log_3 4}$.

Suppose we make the hypothesis:

$$T(n) \le cn^{\log_3 4} - 3n$$

$$\Rightarrow T(n) = 4T\left(\frac{n}{3}\right) + n \le 4\left(c\left(\frac{n}{3}\right)^{\log_3 4} - n\right) + n$$
$$= cn^{\log_3 4} - 4n + n$$
$$= cn^{\log_3 4} - 3n.$$

Q Use a recursion tree to give an asympt. tight solution to the recurrence

$$T(n) = T(n-a) + T(a) + cn, \quad a \ge 1, c > 0 \text{ consts.}$$

$T(a) + cn$

$\downarrow$

$T(a) + c(n-a)$

$\downarrow$

$T(a) + c(n-2a)$

$\vdots$

$T(1)$

Each node of the recursion tree has only one child $\Rightarrow$ cost at each level is cost of the node.

No. of Levels : $\lceil n/a \rceil$

$$\sum_{i=0}^{\lceil n/a \rceil - 1} T(a) + c(n-ia)$$

$$= \lceil n/a \rceil T(a) + c\lceil n/a \rceil n - ca\frac{\lceil n/a \rceil (\lceil n/a \rceil - 1)}{2}$$

Asympt. $\lceil n/a \rceil \approx n/a$

$$\frac{c}{2a}n^2 + \left(\frac{T(a)}{a} + \frac{c}{2}\right)n = \Theta(n^2).$$

Other Sorting Algorithms (Heapsort, Quicksort, Counting Sort, Radix sort, Bucket Sort)

Algorithm	Worst-case run. time	Average-case run. time
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heap Sort	$O(n \lg n)$	—
Quick Sort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting Sort	$\Theta(k+n)$	$\Theta(k+n)$
Radix Sort	$\Theta(d(n+k))$	$\Theta(d(n+k))$
Bucket Sort	$\Theta(n^2)$	$\Theta(n)$

ith Order Statistics: The ith order statistics of a set of n numbers is the ith smallest number in the set.

Method: Sort the input, then index the ith element

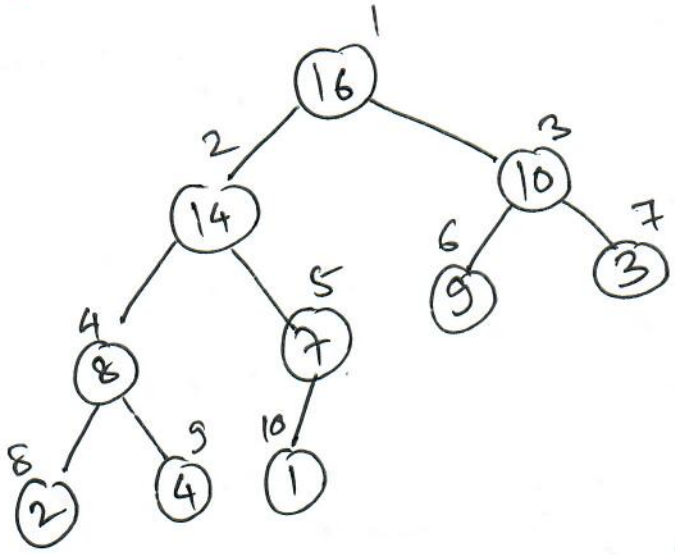
Time: $\Omega(n \lg n)$

(*) Data Structures play an imp. role in complexity of Algorithm. One such imp. data str. is Heap.

Heaps

Binary Heap: Array object that can be viewed as nearly complete binary tree.

Example



16	14	10	8	7	9	3	2	4	1

Binary tree represented as an array.

Left child of $A[i] = A[2i]$

Right " $A[i] = A[2i+1]$

For example: Left child of node numbered (2) is node numbered 4 and right child of node numbered 2 is node numbered 5.

Value of node 2 = 14

Value of node 4 = 8

Value of node 5 = 7.

Height of a node in a heap: number of edges on the longest simple downward path from the node to a leaf.

Height of a Heap: Height of the root of the heap.

Height of a heap of n elements: $\Theta(\lg n)$

Basic Heap Operations

Max-heapify: runs in $O(\lg n)$ time; maintains max-heap property.

Build-Max-Heap: runs in linear time; produces max-heap from unordered input array.

Heap-Sort: runs in $O(n \lg n)$ time, sorts an array in place

Max-heap-insert, Heap-extract-max, Heap-increase-key, and Heap-maximum procedures run in $O(\lg n)$; allows heap data str to be implemented as priority queue.

Algorithms

Parent (i)
return $\lfloor i/2 \rfloor$

Left (i)
return $2i$

Right (i)
return $2i+1$

Heaps

max-heap

- Satisfies max-heap property:
  $A[\text{Parent}(i)] \geq A[i]$
  for every node i
- Largest element is at root
- Heap-sort uses max-heap

min-heap

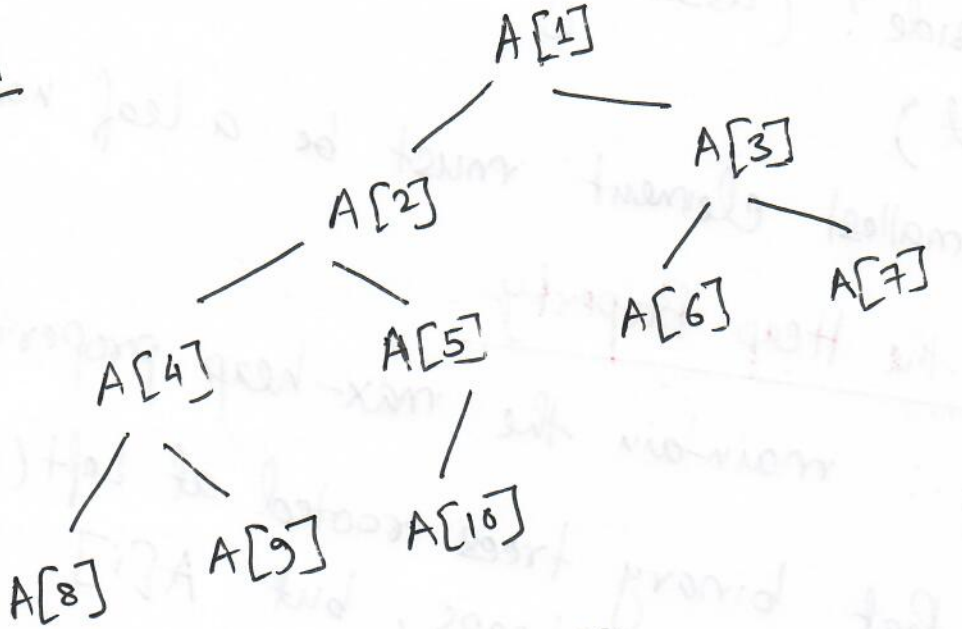
- satisfies min-heap property
  $A[\text{Parent}(i)] \leq A[i]$
  for every node i
- Smallest element is at root
- Implements priority queues

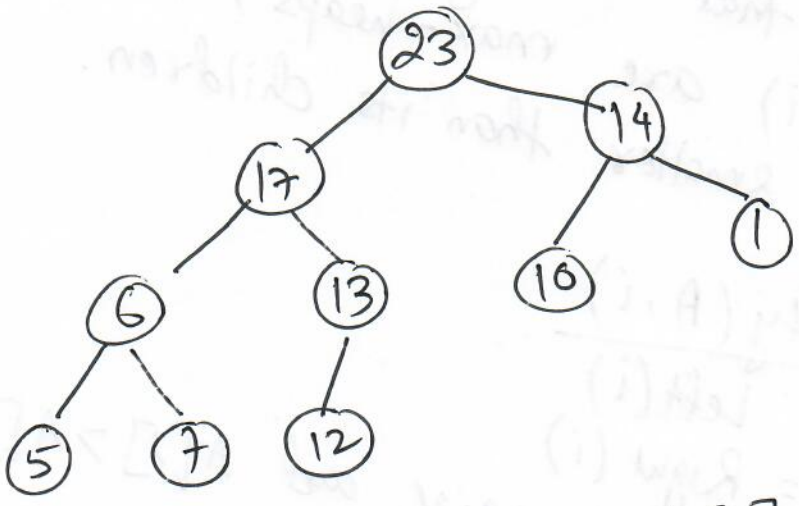
Q Is the array with values

⟨23, 17, 14, 6, 13, 10, 1, 5, 7, 12⟩

a max-heap?

A





Since $A[\text{Parent}(i)] \geq A[i]$ $\forall$ node i

It is a max-heap.

Q Is an array that is in sorted order a min-heap?

A Yes, it is.

Q Where in a max-heap might the smallest element reside? (assuming that all elements are distinct)

A The smallest element must be a leaf node.

Maintaining the Heap Property

Max-heapify : main'ain the max-heap property.

* Assumes that binary trees rooted at Left(i) and Right(i) are max-heaps, but A[i] might be smaller than its children.

Max-Heapify(A, i)

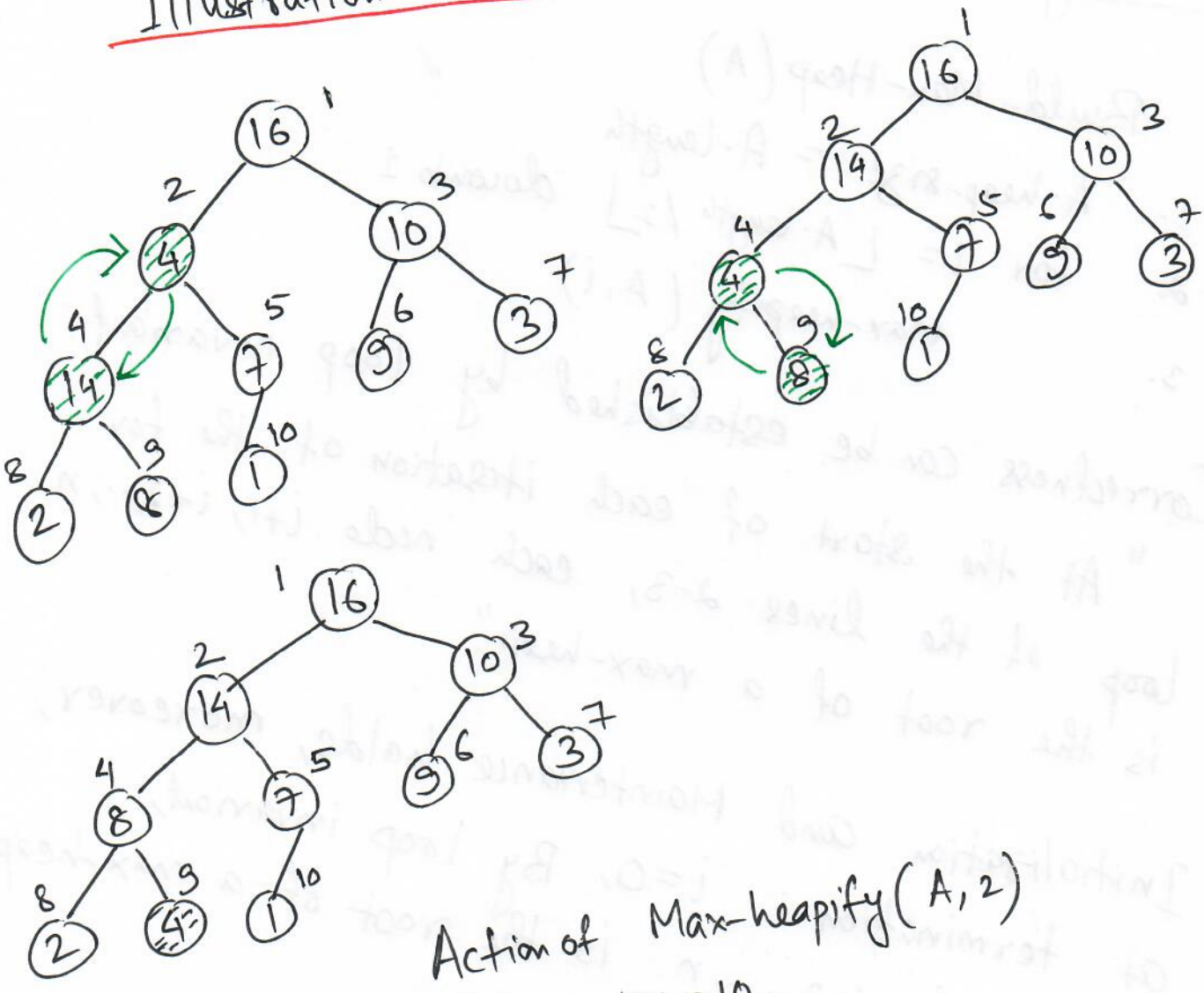
```
l = Left(i)
r = Right(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    Max-heapify(A, largest)
```

Run Time: $T(n) \leq T(2n/3) + \Theta(1)$

(See page 155, CLRS).

$\Rightarrow T(n) = O(\lg n)$.

Illustration of Max-heapify



Action of Max-heapify(A, 2)

A.heap-size = 10.

Q What is the effect of calling Max-heapify (A, i) when the element A[i] is larger than its children?

A The array remains unchanged (see algo).

Building a Heap

```
Build-Max-Heap(A)
1. A.heap-size = A.length
2. for i = ⌊A.length/2⌋ downto 1
3.         Max-heapify(A, i)
```

Correctness can be established by loop invariant:
"At the start of each iteration of the for loop of the lines 2-3, each node i+1, i+2, ..., n is the root of a max-heap".

Initialization and Maintenance holds, moreover, at termination: i=0, By loop invariant, each node 1, 2, ..., n is the root of a max-heap.

Run Time: O(n), page 159, CLRS

Q Illustrate the operation of Build-Max-Heap on the array A = ⟨5, 3, 17, 10, 84, 19, 6, 22, 9⟩.

Algorithms

Heapsort Algorithm

```
Heapsort(A)
1.  Build-Max-Heap(A)
2.  for i = A.length downto 2
3.      exchange A[1] with A[i]
4.      A.heap-size = A.heap-size - 1
5.      Max-Heapify(A, 1)
```

QuickSort

* worst case run-time : $\Theta(n^2)$
* expected run-time : $\Theta(n \lg n)$ ↑ const. factors are small.
* Sort-in-place

```
Quicksort(A, p, r)
1.  if p<r
2.      q = partition(A, p, r)
3.      Quicksort(A, p, q-1)
4.      Quicksort(A, q+1, r)
```

To sort entire array A: Quicksort(A, 1, A.length)

```
Partition(A, p, r)
1.  x = A[r]
2.  i = p-1
3.  for j = p to r-1
        if A[j] ≤ x
4.          i = i+1
5.          exchange A[i] with A[j]
6.
7.  exchange A[i+1] with A[r]
8.  return i+1.
```

Sorting in Linear Time

Comparison sorts: sorting algos that require comparison betn the input elements.

(*) No comparison sort exists that is better than $\Omega(n \lg n)$.

Th. Any comparison sort algo requires $\Omega(n \lg n)$ comparisons in the worst case.

Rk Due to above thm. Heapsort & mergesort being comparison sorts are asymptotically optimal.

Counting Sort

(*) It assumes that each of the n input elements is an integer in the range 0 to k, for some k. When $k = O(n)$, sort runs in $\Theta(n)$ time.

```
Counting-Sort(A, B, k)
1.  let C[0...k] be a new array
2.  for i = 0 to k
3.      C[i] = 0
4.  for j = 1 to A.length
5.      C[A[j]] = C[A[j]] + 1
6.  // C[i] now contains the no. of elements equal to i
7.  for i = 1 to k
8.      C[i] = C[i] + C[i-1]
9.  // C[i] now contains the no. of elements less
       than or equal to i
10. for j = A.length downto 1
11.     B[C[A[j]]] = A[j]
12.     C[A[j]] = C[A[j]] - 1
```

Dynamic Programming

Motivating Example

Consider Fibonacci numbers : 1, 1, 2, 3, ...

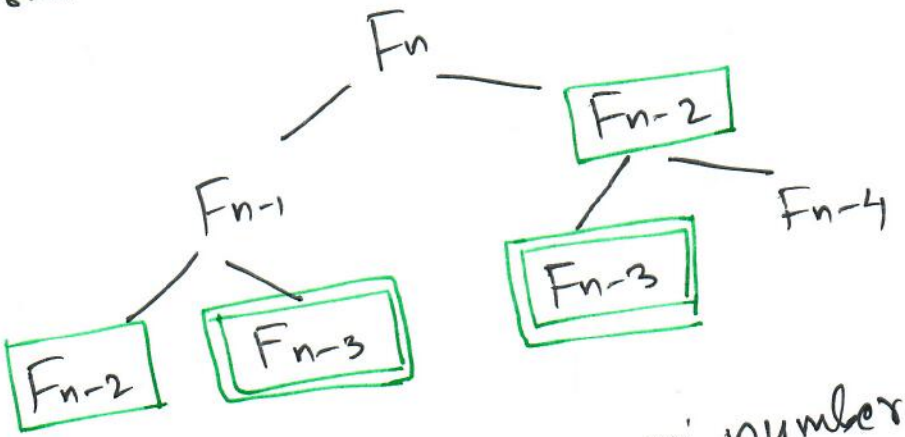
The n^{th} Fibonacci no.

$$F_n = F_{n-1} + F_{n-2}$$

Recursive Algo :

```
fib (n)
  if n ≤ 2
    | return 1
  else
    | return f(n-1) + f(n-2)
```

Recursion Tree :



Observation : Some Fibonacci numbers are computed more than once. Example : F_{n-2} & F_{n-3} are computed twice.

Algorithm

(*) Can avoid duplicate computations by saving first time computed result.

Consider the following:

```
fib = [ ]    // empty array
for k in range (1, n+1)
    if k ≤ 2 : f = 1
    else
        f = fib[k-1] + fib[k-2]
        fib[k] = f
return fib[n]
```

(*) By creating and array fib and saving the results, repeated computation of same Fibonacci no. is avoided.

(*) Avoiding duplicate computation by "memorizing" is an imp. feature of Dyn. prog.

Sequence of steps of DP

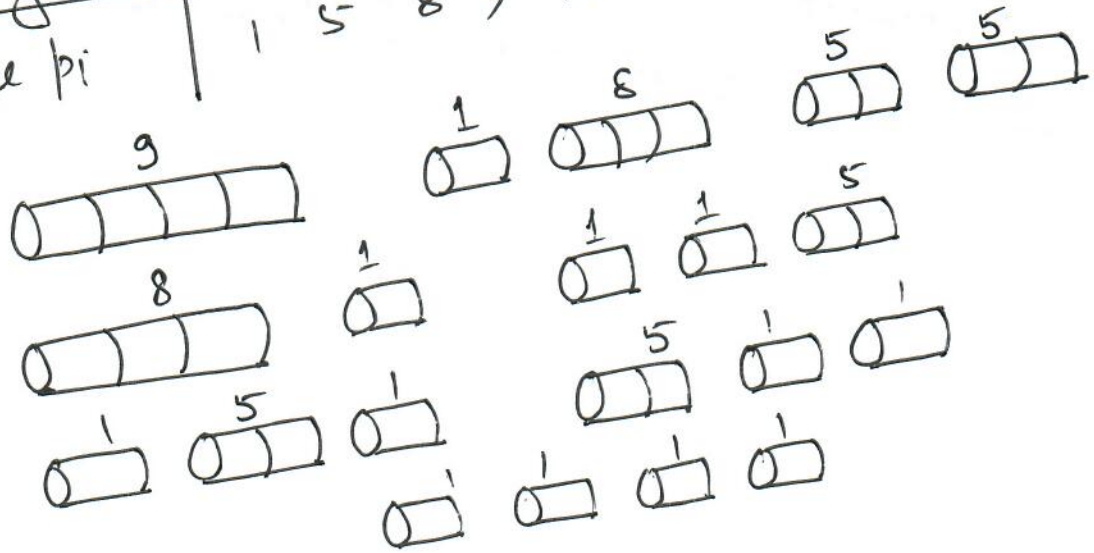
1. Characterize the str. of an optimal soln
2. Recursively define the value of an optimal soln
3. Compute the optimal soln
4. Construct an optimal soln from the computed info.

Rod Cutting

Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$ determine maximum revenue r_n obtainable by cutting up the rod and selling the pieces. If the prices p_n of a rod of length n is large enough, an optimal soln require no cutting.

Example $n = 4$.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



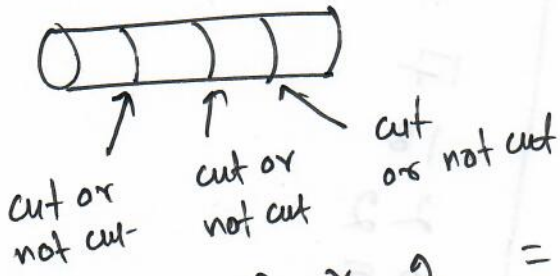
Conclusion: cutting 4-inch rod into 2-inch pieces

$$p_1 + p_2 = 5 + 5 = 10$$

is optimal.

No. of different ways to cut a rod:

For example for a 4-inch rod



$$2 \times 2 \times 2 = 8 = 2^3 = 2^{4-1}$$ possibilities

For a rod of length n: 2^{n-1} possibilities.

Formulation of the Problem

Cut the rod into k pieces $1 \le k \le n$

$$n = i_1 + i_2 + \cdots + i_k$$

where $i_1, i_2, \ldots, i_k$ are rod lengths such that the revenue

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

is maximized.

For the rod of length 10, optimal solutions are:

rod length	Max revenue
1	1
2	5
3	8
$4 = 2+2$	10
$5 = 2+3$	13
6	17
$7 = 1+6$	18
$8 = 2+6$	22
$9 = 3+6$	25
10	10

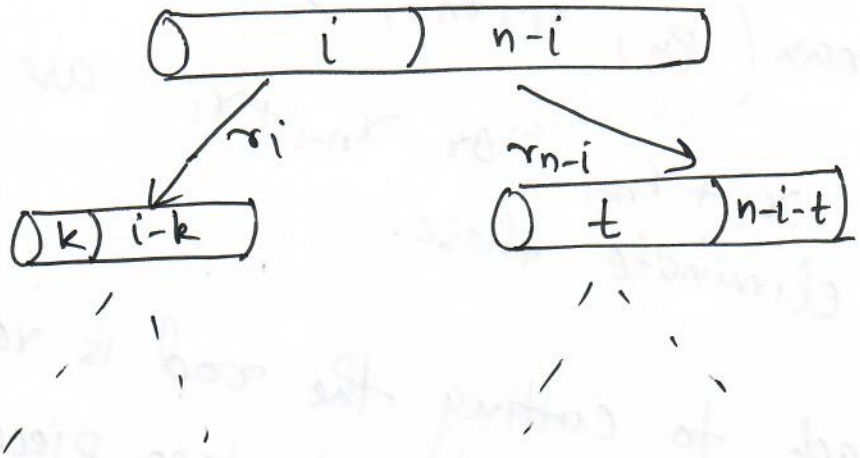
Revenue for cutting the rod of length n

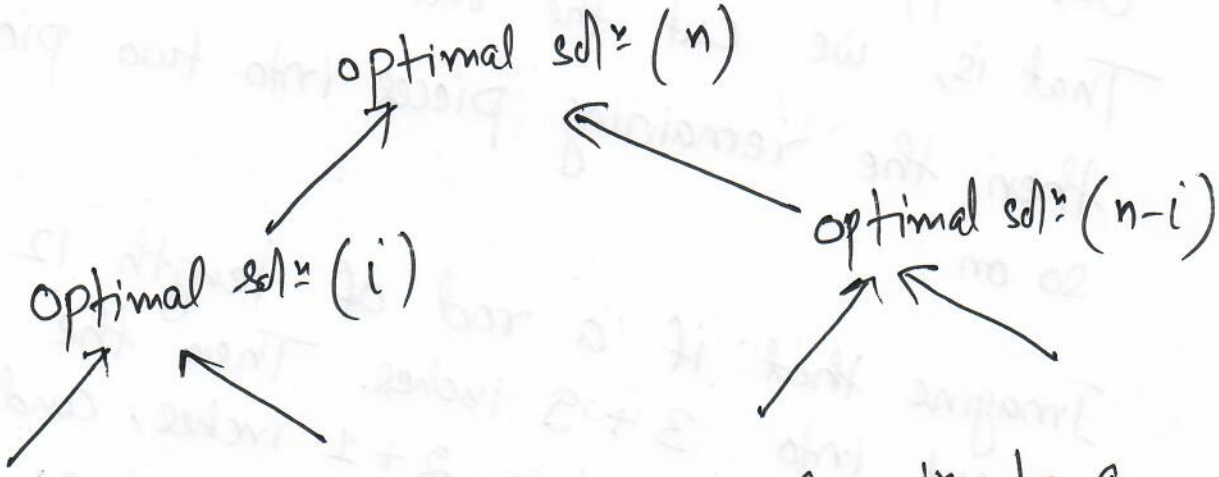
$$r_n = \max(p_n, r_1+r_{n-1}, r_2+r_{n-2}, \ldots, r_{n-1}+r_1) \longrightarrow (*)(*)$$

p_n: no cuts

$r_i + r_{n-i}$: max. revenue for cutting the rod into pieces i & n-i

(*) To solve original problem of size n, solve smaller subproblems of same type, but smaller sizes.





Optimal Substructure: Optimal soln to a problem follows from the optimal soln to related subproblems, which can be solved independently.

Q Can we simplify the recursion for r_n in (*)?

A First, we note that rod is symmetrical, meaning, whether we cut the rod of length 7 into 2 inches and 5 inches or we cut the rod into 5 inches and 2 inches, the revenue will be same.

Hence in (*)(+), ~~the~~ in the expression

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots r_{n-1} + r_1\right)$$

the terms $r_i + r_{n-i}$ or $r_{n-i} + r_i$ are same, hence, we eliminate those.

Our approach to cutting the rod is recursive. That is, we cut the rod into two pieces, then the remaining pieces into two pieces and so on.

Imagine that if a rod of length 12 inches is cut into 3 + 9 inches. Then the piece of 3 inch is cut into 2 + 1 inches, and rod of 9 inches is cut into 7 + 2 inches. So, the rod is cut as follows:

$$12 = 3 + 9 = 2 + 1 + 7 + 2 \longrightarrow (+)$$

Let us assume that this decomposition gives the maximum revenue.

Another approach to arrive at the decomposition (+) would have been

$$12 = 2 + 10 = 2 + 1 + 9 = 2 + 1 + 7 + 2 \longrightarrow (++)$$

We note that we could also arrive at an optimal decomposition by first cutting the rod such that 1st piece is not decomposible, i.e., the 1st piece remains cut. This is an imp. observation because it simplifies the recursion:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}) \longrightarrow (+++)$$

One can rigorously prove that (+++) is equivalent to original recursion

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1).$$

Based on (+++), algo is:

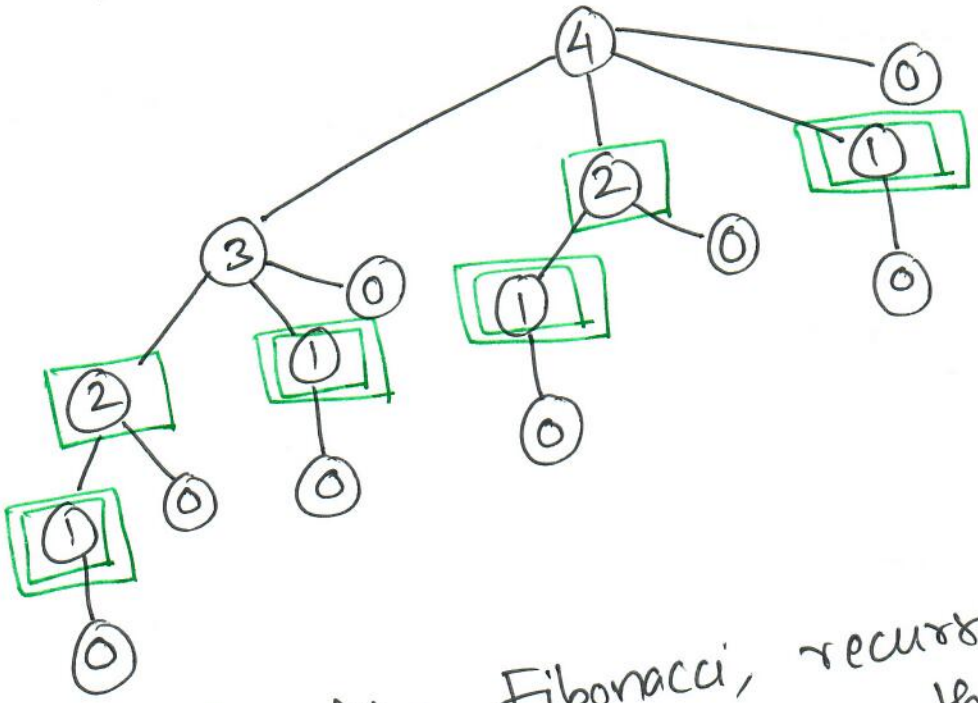
Cut-Rod (p, n)

```
1.  if n == 0
2.      return 0
3.  q = -∞
4.  for i = 1 to n
5.      q = max(q, p[i] + Cut-Rod(p, n-i))
6.  return q
```

- If $n=0$, no revenue is possible, so Cut-Rod returns 0 in Line: 2.
- Line 3 initializes max. revenue to $-\infty$
- Lines 4-5 computes

$$q = \max_{1 \le i \le n} (p_i + \text{Cut-Rod}(p, n-i))$$

For $n=4$, recursion tree:



- Just like Fibonacci, recursion tree, there are repeated calls, i.e, there are overlapping subproblems.
- Remove overlapping subproblems by "memoizing".

Run time of Cut-Rod

$T(n)$: no. of calls made to Cut-Rod when called with 2nd parameter equal to n.

$T(0)$: initial call at root

$T(0) = 1$.

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

$T(j)$: no. of calls (including recursive calls) due to the call Cut-Rod $(p, n-i)$, $j = n-i$.

Since $T(0) = 1$,

$$T(1) = 1 + \sum_{j=0}^{1-1} T(j) = 1 + T(0) = 1+1 = 2 = 2^1$$

$$T(2) = 1 + \sum_{j=0}^{1} T(j) = 1 + T(0) + T(1)$$
$$= 1 + 1 + 2 = 4 = 2^2$$

$$T(3) = 1 + \sum_{j=0}^{2} T(j) = 1 + T(0) + T(1) + T(2)$$
$$= 1 + 1 + 2 + 4 = 8 = 2^3$$

- - -

$$T(j) = 2^j$$

$$\Rightarrow T(n) = 1 + \sum_{j=0}^{n-1} 2^j = 1 + \frac{2^n - 1}{2-1}$$

$$= 1 + 2^n - 1 = 2^n.$$

Hence, Cut-Rod is an exponential time algo.

Removing Overlapping Subprobs

Top-Down Cut-Rod

Memoized-Cut-Rod (p, n)

```
1. let r[0...n] be a new array
2. for i = 0 to n
       r[i] = -∞
3.
4. return Memoized-Cut-Rod-Aux(p, n, r)
```

Memoized-Cut-Rod-Aux (p, n, r)

```
1. if r[n] >= 0
       return r[n]
2.
3. if n == 0
       q = 0
4. else q = -∞
5.      for i = 1 to n
6.          q = max(q, p[i] + Memoized-Cut-Rod-Aux(p, n-i,
7.
8. r[n] = q
   return q
```

- Array $r[0 \cdots n]$ is used to memoize computed value
- In Line 1 of Memoized-Cut-Rod-Aux, it is checked whether a computed value exists, if yes, then it is returned in Line 2.

Bottom-up version

```
Bottom-Up-Cut-Rod(p, n)
1.  Let r[0..n] be a new array
2.  r[0] = 0
3.  for j = 1 to n
4.      q = -∞
5.      for i = 1 to j
6.          q = max(q, p[i] + r[j-i])
7.      r[j] = q
8.  return r[n]
```

How to write Bottom-up when Top-down algo is known?

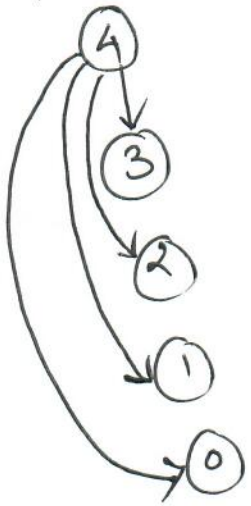
Idea: Look at the subproblem graph

When Top-down: Memoized-Cut-Rod is called with $n = 4$:

- Lines 6-7 of Memoized-Cut-Rod-Aux is called $n = 4$ times, i.e., the following recursive calls are made:

① Memoized-Cut-Rod-Aux $(p, 3, r)$
② Memoized-Cut-Rod-Aux $(p, 2, r)$
③ Memoized-Cut-Rod-Aux $(p, 1, r)$
④ Memoized-Cut-Rod-Aux $(p, 0, r)$

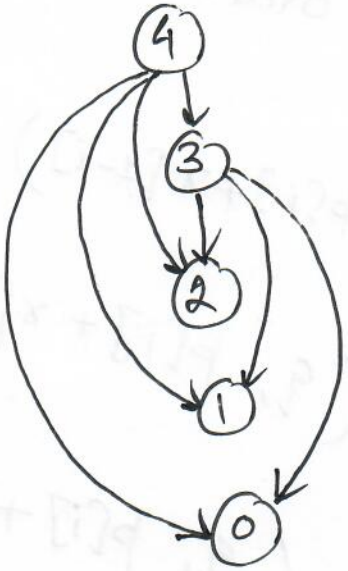
Subproblem graph:



In ① above Memoized-Cut-Rod-Aux $(p, 3, r)$ makes the following calls:

⑤ Memoized-Cut-Rod-Aux $(p, 2, r)$
⑥ Memoized-Cut-Rod-Aux $(p, 1, r)$
⑦ Memoized-Cut-Rod-Aux $(p, 0, r)$

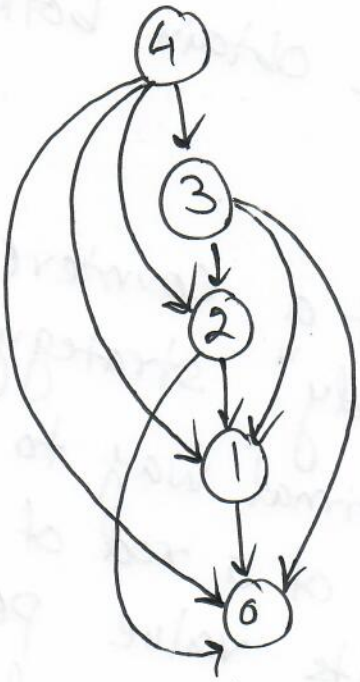
With these calls, the subproblem graph looks like:



Continuing in this, way

Memoized-Cut-Rod-Aux(p, 2, r) and
Memoized-Cut-Rod-Aux(p, 1, r)

is called. The subproblem graph looks like:



Doing Topological sort:

This tells us that:
we can solve subproblems of sizes $j = 0, 1, \ldots, n$ in that order.

① $r[1] = \max_{1 \le i \le 1} (q, p[i] + r[1-i])$

② $r[2] = \max_{1 \le i \le 2} (q, p[i] + r[2-i])$

③ $r[3] = \max_{1 \le i \le 3} (q, p[i] + r[3-i])$

- - -
- - -

Thus, by making the computation in the order above, we obtain bottom-up algo.

Examples

Q1 Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the density of a rod of length i to be p_i/i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where

$1 \le i \le n$, having max. density. It then continues to apply greedy strategy to the remaining piece of length $n-i$.

A. Let $p_1 = 0$, $p_2 = 4$, $p_3 = 7$, and $n = 4$. The greedy strategy would first cut off a piece of length 3 since it has highest density. The remaining rod has length 1, so the total price would be 7. On the other hand, two rods of length 2 yield a price of 8.

Q2. Consides a modification of the rod-cutting problem in which in addition to a price p_i for each rod, each cut incurs a fixed ~~cut~~ cost of c. The revenue associated with a soln is now the sum of the prices of the pieces minus the cost of making the cuts. Give a dynamic programming algorithm to solve this modified problem.

A Hint:

$$r_n = \max\{p_n, r_1 + r_{n-1} - c, r_2 + r_{n-2} - c, \ldots, r_{n-1} + r_1 - c\}$$

Matrix-Chain Multiplication

Problem: Given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, we wish to compute the product

$$A_1 A_2 \cdots A_n$$

such that the cost of mult. is minimum

Parenthesizing a matrix

$\langle A_1, A_2, A_3, A_4 \rangle$

Diff. ways of paranth.

$A_1 (A_2 (A_3 A_4))$

$A_1 ((A_2 A_3) A_4)$

$(A_1 A_2)(A_3 A_4)$

$(A_1 (A_2 A_3)) A_4$

$((A_1 A_2) A_3) A_4$

Q Why consider this problem?

A Consider $\langle A_1, A_2, A_3 \rangle$

$A_1 \in \mathbb{R}^{10 \times 100}$, $A_2 \in \mathbb{R}^{100 \times 5}$, $A_3 \in \mathbb{R}^{5 \times 50}$

$\text{Cost}(A_1 * A_2) = 10 \times 100 \times 5 = 5000$

$$\text{Cost}\left((A_1 * A_2) * A_3\right) = 10 \times 5 \times 50$$
$$= 2500 + 5000$$
$$= 7500$$

$$\text{Cost}\left(A_2 * A_3\right) = 25000$$

$$\text{Cost}\left(A_1 * (A_2 * A_3)\right) = 25000 + 50,000$$
$$= 75,000$$

$\Rightarrow$ $(A_1 * A_2) * A_3$ is 10 times faster than $A_1 * (A_2 * A_3)$

Conclusion: By suitable parenthesization, we can reduce the number of multiplications/additions.

Problem Formulation

Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices where $i = 1, 2, \ldots, n$

$$A_i \in \mathbb{R}^{p_{i-1} \times p_i},$$

find a fully parenthesization of the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Counting the number of Parenthesizations

Consides the product:

$$A_1 A_2 \cdots A_n$$

Consider putting a parenthesis after A_k like so:

$$A_1 A_2 \cdots A_{k-1} A_k \,|\, A_{k+1} \cdots A_n$$

(put parenthesis here)

Let $P(k)$ be the no. of ways to parenthesize

$$A_1 A_2 \cdots A_{k-1} A_k$$

and $P(n-k)$ be the no. of ways to parenthesize

$$A_{k+1} A_{k+2} \cdots A_n$$

Given that a parenthesis is put after A_k, the total no. of ways to do parenthesization is: $P(k)\,P(n-k)$,

because for every parenthesization of $A_1 \cdots A_k$ there are $P(n-k)$ parenthesization of

$$A_{k+1} \cdots A_n.$$

But we can vary k, i.e, we can put parenthesis after A_1, or after A_2, ... or after A_{n-1} $\Rightarrow$ no. of ways to choose $k = n-1$,

$k = 1, 2, \cdots, n-1$.

Hence $P(n)$: total no. of ways of parenthesizing $A_1 \cdots A_n$.

$$P(n) = \sum_{k=1}^{n-1} P(k)\,P(n-k) \quad \text{if } n \geq 2$$

If $n = 1$, then $P(n) = 1$.

Claim: $P(n) \geq 2^n - 1$.

Pf: For $n = 1$, $P(1) = 1 \geq 2^1 - 1 = 1$ ok.

Assume $P(k) \geq 2^k - 1$

By induction:

$$P(n) = \sum_{k=1}^{n-1} P(k)\,P(n-k)$$

$$\geq \sum_{k=1}^{n-1} 2^k 2^{n-k} = (n-1)(2^n - 1) \geq 2^n - 1.$$

$\Rightarrow$ $P(n) = \Omega(2^n)$, hence, there are exponential no. of ways of parenthesizing.

Hence, brute-force method is hopeless!

Dynamic Prog. to Rescue!

we recall the steps of DP:

1) Characterize the str. of an optimal sol^n
2) Recursively define the value of an optimal sol^n.
3) Compute the value of an optimal sol^n
4) Construct an optimal sol^n from the computed information.

Step 1 Suppose to optimally parenthesize

$A_1 A_2 \cdots A_k A_{k+1} \cdots A_j$

$A_1, A_2 \cdots A_{k-1}\ A_k\ A_{k+1} \cdots A_j$

we split the product bet^n A_k and A_{k+1}.
Then it must be true that $A_1 A_2 \cdots A_k$ and $A_{k+1} \cdots A_j$ must have optimal parenthesization, because, if not, then by replacing the optimal parenthesization of $A_1, A_2 \cdots A_k$ or $A_{k+1} \cdots A_j$ in the assumed optimal parenthesization of $A_1 \cdots A_j$ we get the cost that is lower than the optimum.

Hence, optimal sol^n of subproblems, in this case, cost of parenthesizing $A_1 A_2 \cdots A_k$ and

Optimal soln of $A_{k+1} \cdots A_j$ are part of the optimal soln of original problem, in this case $A_1 \cdots A_j$.

Note: The general strategy for proving optimal substr. is via cut-paste argument.

(1) Assume soln to a problem is optimal

(2) Divide the problem into two subproblems.

This division of problem into two subproblems is due to problem requirement: For example,

A) Rod Cutting: The original rod is cut into pieces k and $n-k$

B) Matrix Chain: The matrix chain $A_1 \cdots A_j$ is divided into $A_1 \cdots A_k$ and $A_{k+1} \cdots A_j$

In A) problem requirement: cut the rod

In B) problem requirement: put the parenthesis

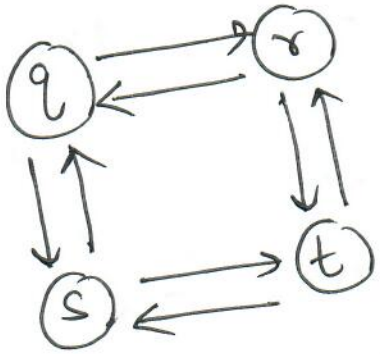
(3) Given an optimal soln, and considering two subproblems, the optimal soln to two subproblems must be optimal, if not then we cut the optimal soln of the subproblem and paste in the original soln to get a cost lower than optimal, a contradiction.

Note: Such cut-paste argument needs to be applied to check whether we arrive at a contradiction. If we do arrive at a contradiction, then optimal soln to subprob. must be "part" of optimal soln to original problem.

Problem without Optimal Substructure

The problem of finding longest simple path betn two vertices in a directed graph does not have optimal substr. property.
[Recall: A simple path betn two nodes is a path without cycles]

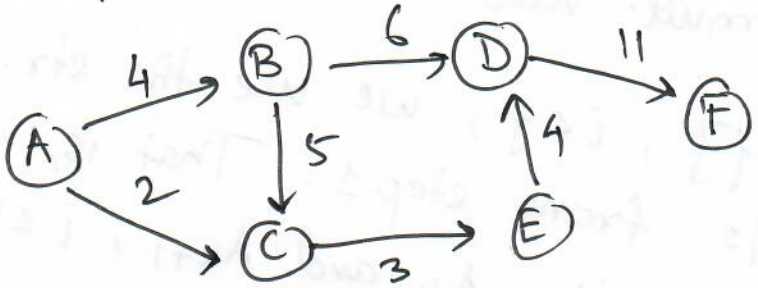
Consider:



Longest path from q to t: $q \to r \to t$
or
$q \to s \to t$

The soln to the longest path from q to r which is $q \to s \to t \to r$ is not part of the soln to the longest path from q to t that goes via r.

On the other hand, shortest path problem has optimal substr. For example:



Shortest path from (A) to (D) : (A) → (C) → (E) → (D)
with wt. = 9

Shortest path from (A) to (E) : (A) → (C) → (E)
with wt. = 5

⇒ The shortest path from (A) to (E) is part of the shortest path from (A) to (D). Hence, this problem has optimal substr.

Step 2 Recursive Solution to Matrix Chain Mult. Problem

Let $m[i,j]$: minimum number of scalar multiplications needed to compute $A_i A_2 \cdots A_j$

⇒ $m[1,n]$: minimum no. of scalar multipli-cations needed to Compute $A_1 A_2 \cdots A_n$

Define $m[i,j]$ recursively

- if $i=j$, chain contains only one matrix A_i so no scalar mult. needed. $\Rightarrow m[i,i]=0 \ \forall i=1,\dots n.$
- For $m[i,j]$, $i<j$, we use the str. of optimal soln from step 1. That is, split $A_1 A_2 \cdots A_j$ betn A_k and A_{k+1}, $i \le k < j$.

$$m[i,j] = \min_{k} \text{ cost for computing } A_1 \cdots A_k$$
$$+ \min_{k} \text{ cost for computing } A_{k+1} \cdots A_j$$
$$+ \text{cost for computing the product } (A_1 \cdots A_k)(A_{k+1} \cdots A_j)$$

That is:

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \longrightarrow (*)$$

Q How did we get last term: $p_{i-1} p_k p_j$?

A. Recall the dimensions of A_i in the matrix chain $\langle A_1, A_2, \dots, A_n \rangle$ are $p_{i-1} \times p_i$.

$\Rightarrow$ Dimension of the matrix $A_i \cdots A_k = p_{i-1} \times p_k$

p_{i-1} : no. of rows in A_i

p_k : no. of cols in A_k

Similarly dimension of matrix $A_{k+1} \cdots A_j = p_k \times p_j$

⇒ Cost ~~Dimension~~ of

$$(A_i \cdots A_k)(A_{k+1} \cdots A_j) = p_{i-1} p_k p_j .$$

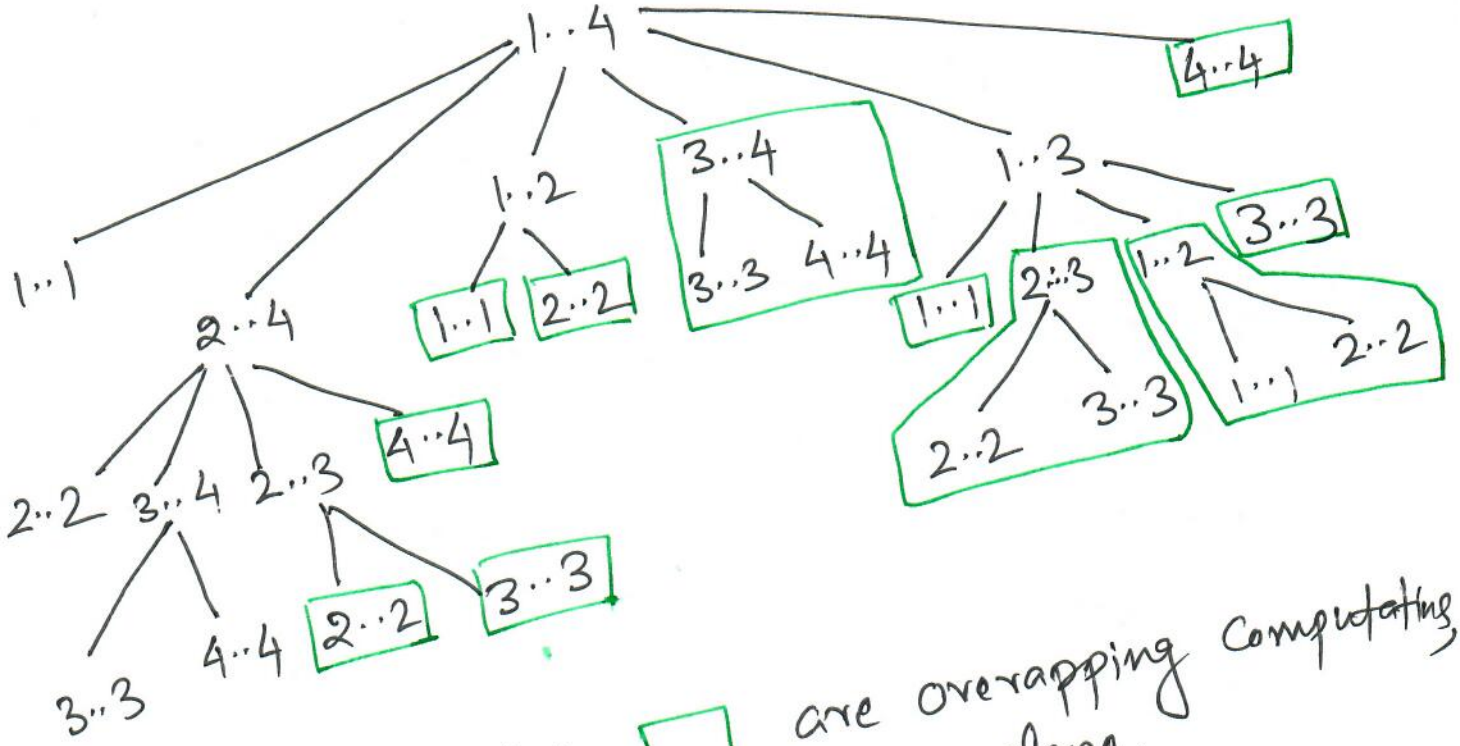
In the recursion ⊕, we already assumed that we know where to split, but actually, we don't know k, we need to find it. Hence, optimal k will be the one that leads to minimum value of $m[i,k] + m[k+1, j] + p_{i-1} p_k p_j$.

Hence:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k \le j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & \text{if } i < j. \end{cases}$$

Note: recursive algo may encounter each subproblem many times in different branches of its recursion tree. That is, we may have overlapping subproblems.

Consider the chain $\langle A_1, A_2, A_3, A_4 \rangle$. Let the product $A_1 A_2 A_3 A_4$ be denoted by $1 \cdots 4$.



Computations boxed in ▭ are overlapping computations, i.e., computations that has been done.

Recursive-Matrix-Chain (p, i, j)

```
1.  if i == j
        return 0
2.
3.  m[i,j] = ∞
4.  for k = i to j-1
5.      q = Recursive-Matrix-Chain(p, i, k)
            + Recursive-Matrix-Chain(p, k+1, j)
            + p_{i-1} p_k p_j
6.      if q < m[i,j]
7.          m[i,j] = q
8.  return m[i,j]
```

The memoized version is:

Memoized-Matrix-Chain(p)

```
1.  n = p.length - 1
2.  let m[1...n, 1...n] be a new table
3.  for i = 1 to n
4.      for j = i to n
5.          m[i,j] = ∞
6.  return Lookup-Chain(m, p, 1, n)
```

Lookup-Chain(m, p, i, j)

```
1.  if m[i,j] < ∞
        return m[i,j]
2.
3.  if i == j
4.      m[i,j] = 0
5.  else for k = i to j-1
6.      q = Lookup-Chain(m, p, i, k)
            + Lookup-Chain(m, p, k+1, j) + p_{i-1} p_k p_j
7.      if q < m[i,j]
8.          m[i,j] = q
9.  return m[i,j]
```

Bottom-Up Matrix Chain Algo

Matrix-Chain-Order (p)

```
1.  n = p.length - 1
2.  let m[1..n, 1...n] and s[1.. n-1, 2..n] be
                        new tables

3.  for i = 1 to n
        m[i,i] = 0
4.
5.  for l = 2 to n
        for i = 1 to n-l+1
6.          j = i+l-1
7.          m[i,j] = ∞
8.          for k = i to j-1
9.              q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
10.             if q < m[i,j]
11.                 m[i,j] = q
12.                 s[i,j] = k
13.
14. return m and s
```

Step 4 Constructing an Optimal Soln.

```
print-optimal-parens (s, i, j)
1.  if i == j
2.      print "A_i"
3.  else print "("
4.      print-optimal-parens (s, i, s[i,j])
5.      print-optimal-parens (s, s[i,j]+1, j)
6.      print ")"
```

Worked out example (see page 376, CLRS)

Q Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$

A $(A_1 A_2)((A_3 A_4)(A_5 A_6))$

which requires: $5*50*6 + 3*12*5 + 5*10*3 + 3*5*6 + 5*3*6$

$= 1500 + 180 + 150 + 90 + 90$

$= 2010.$

Q Draw the recursion tree for Merge-Sort procedure. Explain why memoization fails to speed up a good divide-and-conquer algorithm such as Merge-Sort.

A Let [i..j] denote the call to Merge sort to sort the elements in positions i through j of the original array. The recursion tree will have [1..n] as its root, and at any node [i..j] will have [i..(j-i)/2] and [(j-i)/2+1..j] as its left and right children, resp.

The memoization approach fails to speed up Merge sort because the subproblems aren't overlapping: Sorting one list of size n isn't the same as sorting another list of size n, so there is no savings in sorting solutions to subproblems, since each sol'n is used atmost once.

Q Consider a variant of the matrix-chain mult. problem in which the goal is to parenthesize the sequence of matrices so as to maximize rather than minimize, the no. of scalar mult. Does this problem exhibit optimal substr.?

A. Yes. (How?)

Q As stated, in DP we first solve the subprobs and then choose which of them to use in an optimal soln to the problem. Prof. Capulet claims that we do not always need to solve all the subproblems in order to find an optimal soln. She suggests that we can find an optimal soln to the matrix-chain mult problem by always choosing the matrix A_k at which to split the subproduct $A_i A_{i+1} \cdots A_j$ by selecting k to minimize the quantity $p_{i-1} p_k p_j$ before solving the subprobs. Does this lead to optimal soln?

A. No. Choose A_1, A_2, A_3, and A_4 with

$$p_0, p_1, p_2, p_3, p_4 = 1000, 100, 20, 10, 100$$

Then $p_0 p_k p_4$ is minimized when $k = 3$.

Now, to find parenth. of $A_1 A_2 A_3$, we choose k for which $p_0 p_k p_3$ is minimized,

$\Rightarrow k = 2$

Hence, full parenth. is

$$((A_1 A_2) A_3) A_4.$$

Cost = 12,200,000, But optimal cost is 11,020,000.

Q Suppose that in the rod-cutting problem we also had limit l_i on the number of pieces of length i that we are allowed to produce, for $i = 1, 2, \ldots, n$. Show that the optimal-substr. property described for rod-cutting no longer holds.

A. The optimal substr. prop. doesn't hold because no. of pieces of length i used on one side of the cut affects the number allowed on the other. There is information about the particular sol^n on one side of the cut that changes what is allowed on the other.

For example: ∘ consides rod of length 4.

∘ Let $l_1 = 2$, $l_2 = l_3 = l_4 = 1$.

∘ Each piece has same worth regardless of length

Optimal sol^n: cut rod into 4 pieces, each of length 1: First cut in middle, optimal sol^n for two left over rods is to cut them in middle, which isn't allowed, because it increases the total no. of rods of length 1, and max allowed is 2

Longest Common Subsequence

Def: (subsequence) Given a sequence $X = \langle x_1, x_2, \cdots x_m \rangle$ another sequence $Z = \langle z_1, z_2, \cdots, z_k \rangle$ is a subseq. of X if there exists a strictly increasing sequence $\langle i_1, i_2, \cdots i_k \rangle$ of indices of X s.t $\forall j = 1, 2, \cdots k$, we have $x_{i_j} = z_j$.

Example: $Z = \langle B, C, D, B \rangle$ is a subseq. of $X = \langle A, B, C, B, D, A, B \rangle$ with corresp. index $\langle 2, 3, 5, 7 \rangle$.

Def: (Common subsequence) Given two sequences X and Y, Z is a common subseq. of X and Y if Z is a subsequence of both X and Y.

Example: $X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

$Z = \langle B, C, A \rangle$ is a common subseq. of both X and Y.

But $Z = \langle B, C, B, A \rangle$ is a longest common subseq.

Longest Common Subsequence Problem

Given $X = \langle x_1, x_2, \ldots x_m \rangle$

$Y = \langle y_1, y_2, \ldots y_n \rangle$

Find a maximum length longest common subseq. of X and Y.

Solving LCS using DP

Step 1: Optimal substr.

Th. (optimal substr. of an LCS)

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}

Pf page 392, CLRS (3rd ed.)

Step 2: A recursive soln:

Theorem above implies that we should examine either one or two subproblems when finding an LCS of $X = \langle x_1, x_2, \cdots, x_m \rangle$ and $Y = \langle y_1, y_2, \cdots y_n \rangle$. If $x_m = y_n$, we must find an LCS of X_{m-1} and Y_{n-1}. Appending $x_m = y_n$ to this LCS yields an LCS of X and Y. If $x_m \neq y_n$, then we must solve two subproblems: finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}. Whichever of these LCS is longer is an LCS of X and Y.

Existence of overlapping subproblems: To find an LCS of X and Y, we may need to find the LCS of X and Y_{n-1} and of X_{m-1} and Y. But each of these subproblems has the subproblem of finding an LCS of X_{m-1} and Y_{n-1}.

Recursive Formula

Let $c[i,j]$ = length of an LCS of sequences X_i and Y_j.

If either $i=0$ or $j=0$: LCS has length 0.

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0, \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i = y_j, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i \neq y_j \end{cases}$$

Step 3: Computing the length of an LCS

Implementing recursive algo for $c[i,j]$, we get exp. time algo. We see that there are $\Theta(mn)$ distinct subproblems. Using bottom-up approach:

```
LCS-Length(X, Y)
1.  m = X.length
2.  n = Y.length
3.  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
4.  for i = 1 to m
5.      c[i,0] = 0
6.  for j = 0 to n
7.      c[0,j] = 0
8.  for i = 1 to m
9.      for j = 1 to n
10.         if x_i == y_j
11.             c[i,j] = c[i-1,j-1] + 1
12.             b[i,j] = "↖"
13.         elseif c[i-1,j] ≥ c[i,j-1]
14.             c[i,j] = c[i-1,j]
15.             b[i,j] = "↑"
16.         else c[i,j] = c[i,j-1]
```

17. $b[i,j] = "\leftarrow"$
18. return c and b

Run Time : $\Theta(mn)$

Step 4 : Constructing an LCS

- The b table returned by LCS-length is used to construct LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots y_n \rangle$.
- Begin at $b[m,n]$ and trace thro. the table by following the arrows.

"$\nwarrow$" : implies $x_i = y_j$

Print-LCS (b, X, i, j)

1. if i == 0 or j == 0
2. return
3. if $b[i,j] == "\nwarrow"$
4. print_LCS (b, X, i-1, j-1)
5. print x_i
6. elseif $b[i,j] == "\uparrow"$
7. print_LCS (b, X, i-1, j)
8. else print-LCS (b, X, i, j-1)

Run Time : $O(m+n)$

Solved Examples

Image Compression by seam Carving

We are given a color picture consisting of an $m \times n$ array $A[1..m, 1..n]$ of pixels, where each pixel specifies a triple of red, green, and blue (RGB) intensities. Suppose, that we wish to compress this picture slightly, Specifically, we wish to remove one pixel from each ~~row~~ of the m rows, so that the whole picture becomes one pixel narrower. To avoid disturbing visual effects, however, we require that the pixels removed in two adjacent rows be in the same or adjacent columns; the pixels removed formed a "seam" from the top row to the bottom row where successive pixels in the seam are adjacent vertically or diagonally.

a) Show that number of such possible seams grows at least exponentially in m, assuming that $n > 1$.

b) Suppose now that along with each pixel $A[i, j]$ we have calculated a real valued disruption

measure d[i,j], indicating how disruptive it would be to remove pixel A[i,j]. Intuitively, the lower a pixel's disruption measure, the more similar the pixel is to its neighbors. Suppose further that we define the disruption measure of a seam to be the sum of the disruption measures of its pixels.

Give an algorithm to find a seam with the lowest disruption measure. How efficient is your algorithm?

Answer a) If $n>1$ then for every choice of pixel at a given row, we have atleast 2 choices of pixel in the next row to add to the seam, and 3 choices of pixel if the column is other than 1 or n. Thus total number of possibilities is bounded below by 2^n.

b) We create a table $D[1..m, 1..n]$ s.t. $D[i,j]$ stores the disruption of an optimal seam ending at position $[i,j]$ which started in row 1. We also create a table $S[i,j]$ which stores the list of ordered pairs indicating which pixels were used to create the optimal seam ending at position (i,j). To find the solution to the problem,

we look for the minimum k entry in row m of table D, and use the list of pixels stored at S[m,k] to determine the optimal seam. To simplify the algorithm Seam(A), let MIN(a,b,c) be the fn which returns -1 if a is the minimum, 0 if b is the minimum, and 1 if c is the minimum value from among a, b, and c. The time complexity of the algo. is O(mn).

Algorithms

Algorithms

Greedy Algorithms

(*) Greedy Algo. another way to solve optimizatio problem

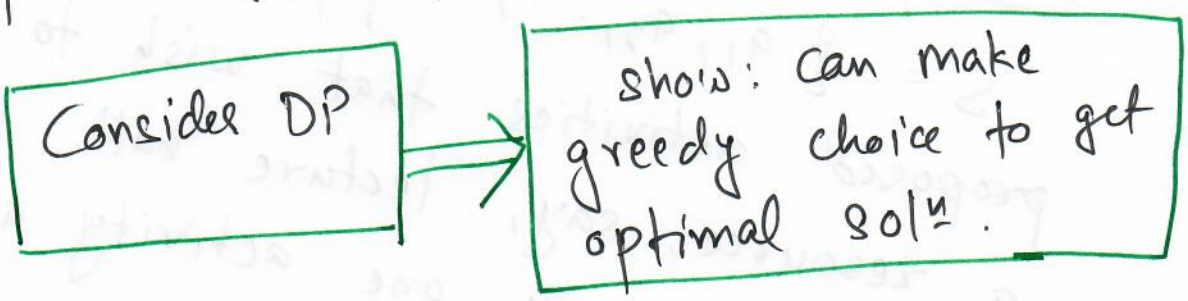
Optimization problem: problems based on involving max or min some functions.

(*) Using DP to determine the best choice is sometimes a overkill.

(*) Greedy Algo: make the choice that looks best at the moment. Don't consider all subproblems.

(*) Note: Greedy algo does not always yield optimal soln.

One possible approach:



Topics to cover:

1) Activity selection problem
2) Matroid Theory (Combinatorial str. for greedy algo.)

(*) For Matroids, greedy algo. always produces an optimal soln.

Greedy Algo in Graphs

1) Minimum Spanning Trees (Prim's & Kruskal's algo)
2) Dijkstra's algo for shortest path
3) Chvātal's greedy set-covering heuristics.

An Activity Selection Problem

$S = \{a_1, a_2, \ldots, a_n\}$, a set of n proposed activities that wish to use a resource, say, lecture hall, which can serve only one activity at a time.

For activity a_i

start time : s_i

finish time : f_i

$0 \le s_i < f_i < \infty$

(*) a_i takes place in time interval : $[s_i, f_i)$

(*) a_i and a_j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap, i.e., $[s_i, f_i) \cap [s_j, f_j) = \phi$ (empty set/interval)

(*) In other words, a_i, a_j are compatible if $s_i \ge f_j$ or $s_j \ge f_i$.

i.e., start time of one should be greater than the finish time of other.

Activity Selection Problem : select a max. size subset of mutually compatible activities.

Activities are sorted as :

$f_1 \le f_2 \le f_3 \le \dots \le f_{n-1} \le f_n$.

Example

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Q What are mutually compatible activities?

A One possibility: $\{a_3, a_9, a_{11}\}$

a_3 happens in: $[0, 6)$

a_9 " " : $[8, 12)$

a_{11} " " : $[12, 16)$

Since, the time for a_3, a_9, and a_{11} do not overlap, a_3, a_9, and a_{11} are called compatible activities.

Second possibility: $\{a_1, a_4, a_8, a_{11}\}$

a_1 happens in: $[1, 4)$

a_4 " " : $[5, 7)$

a_8 " " : $[8, 11)$

a_{11} " " : $[12, 16)$

Since time intervals of a_1, a_4, a_8, a_{11} do not overlap, they are compatible activities.

Third possibility: $\{a_2, a_4, a_9, a_{11}\}$

Solving Activity Selection Problem

Idea Sketch:

Consider DP, in which several choices (subprobs) are considered to determine which subprob. to use in an optimal soln

step-1: DP

Consider only one choice: the greedy choice => only one subprob. remains

step2: Greedy

Optimal Substr.

S_{ij}: set of activities that start after activity a_i finishes and that finishes before activity a_j starts.

A_{ij}: max. set of mutually compatible activities in S_{ij}.

For example:

$S_{2,11}$ = set of all activities that start after a_2 finishes and finishes befor a_{11} starts.

a_2 finishes at : 5

a_{11} starts at : 12

Let t_i be the time interval $[s_i, f_i)$ of activity a_i.

$A_{2,11}$ = ~~all activities~~ ~~a_i~~ max. set of mutually compatible activities in $S_{2,11}$.

List of ~~comp~~ activities in $S_{2,11}$

$t_1 = [1,4) \notin [5,12)$

$t_2 = [3,5) \notin [5,12)$

$t_3 = [0,6) \notin [5,12)$

$t_4 = [5,7) \subset [5,12)$

$t_5 = [3,9) \notin [5,12)$

$t_6 = [5,9) \subset [5,12)$

$t_7 = [6,10) \subset [5,12)$

$t_8 = [8,11) \subset [5,12)$

$t_9 = [8,12) \subset [5,12)$

$t_{10} = [2,14) \notin [5,12)$

$t_{11} = [12,16) \notin$ ~~[12,16)~~ $[5,12)$

Hence:

$S_{2,11} = \{a_4, a_6, a_7, a_8, a_9\}$

$A_{2,11}$ = set of maximally mutually compatible activities.

Mutually compatible sets are:

$\{a_4, a_8\}$, $\{a_4, a_9\}$, $\{a_4, a_9\}$,

$\{a_4, a_8\}$

$\Rightarrow A_{2,11} = \{a_4, a_9\}$ or $\{a_4, a_8\}$

Let a_k be some activity in A_{ij}

S_{ik}: activities that start after activity a_i finishes and finish before activity a_k starts.

S_{kj}: activities that start after activity a_k finishes and that finish before activity a_j starts.

Let $A_{ik} = A_{ij} \cap S_{ik}$

$A_{kj} = A_{ij} \cap S_{kj}$

$\Rightarrow A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

$\Rightarrow |A_{ij}| = |A_{ik}| + |A_{kj}| + 1$

Claim: Optimal solution A_{ij} must also include optimal sol^n to the two subproblems S_{ik} and S_{kj}.

Pf (Apply cut-paste argument)

Hypothesis (P): A_{ij} is an optimal sol^n and it is split at k^{th} activity.

Conclusion (Q): Optimal solution to subprobs A_{ik} and A_{kj} must also be part of optimal soln to A_{ij}

We need to prove : $P \Rightarrow Q$.

We prove by contradiction. We assume $\neg Q$, i.e., we assume that $\exists$ a set A'_{kj} of max. mutually compatible activities in S_{kj} with $|A'_{kj}| > |A_{kj}|$, then we can use A'_{kj} rather than A_{kj} in a sol^n to the subprob. for S_{ij}.

[That is, we can cut ~~& paste A'_{kj}~~ A_{kj} and paste A'_{kj} in A_{ij}.]

Then $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1$

$= |A_{ij}|$

This contradicts our hypothesis:
Aij is an optimal soln.

Hence, by proof by contradiction, in particular by contrapositive proof: $\neg Q$ must be false, i.e., Q must be true, i.e., an optimal soln to Aik and Akj must also be part of optimal soln to Aij.

Let
$c[i,j]$ = size of optimal soln for the set S_{ij}.

$$c[i,j] = c[i,k] + c[k,j] + 1.$$

(k is known if k is known.)

To find k:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{a_k \in S_{ij}} \{ c[i,k] + c[k,j] + 1 \} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

DP approach: 1) develop recursive algo and memoize it 2) develop bottom-up approach & fill table entries

Q Can we do better than DP?

Q What if we could choose an activity that adds to optimal sol^n without having to first solve all the subprobs.?

Several greedy choices

1) Choose activity in S that has earliest finish time

2) Choose activity in S that overlaps with least no. of activities

3) Choose an activity in S of least duration

Greedy Choice - 1

Since activities are ordered w.r.t finish time: activity that finishes first is a_1.

After making the greedy choice a_1, the next activity that is selected is among all those activities that start after a_1 finishes and finishes earliest.

Let $S_k = \{ a_i \in S : s_i \geq f_k \}$

s_i: start time of a_i

f_k: finish time of a_k.

Optimal Substr: If a_1 is in the optimal soln, then optimal soln to the original problem = activity $\{a_1\}$ ∪ all activities in optimal soln to subprob. S_1.

Q Is our greed choice of choosing the 1st activity to finish always part of optimal soln?

Thm Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some max-size subset of mutually compatible activities of S_k.

Proof: Let A_k = max. size subset of mutually compatible activities in S_k

a_j : activity in A_k with earliest finish time.

If $a_j = a_m$, we are done, since then we have that a_m is in some max-size subset of mutually compatible activities of S_k.

If $a_j \neq a_m$, let the set

$$A_k' = A_k \setminus \{a_j\} \cup \{a_m\},$$

↑ set difference.

which is same as A_k but with a_j replaced by a_m.

Activities in A_k' are disjoint, because the activities in A_k are disjoint (since A_k is a set of max. mut. compat. activities), and since $A_k' \subset S_k$, and since a_m and $a_j \in S_k$, a_m must be the 1st activity in A_k' to finish.

$\Rightarrow |A_k'| = |A_k|$

$\Rightarrow$ A_k' is a max-size subset of mut. comp. activities of S_k, and it includes a_m.

In line: $A_k' = A_k - \{a_j\} \cup \{a_m\}$

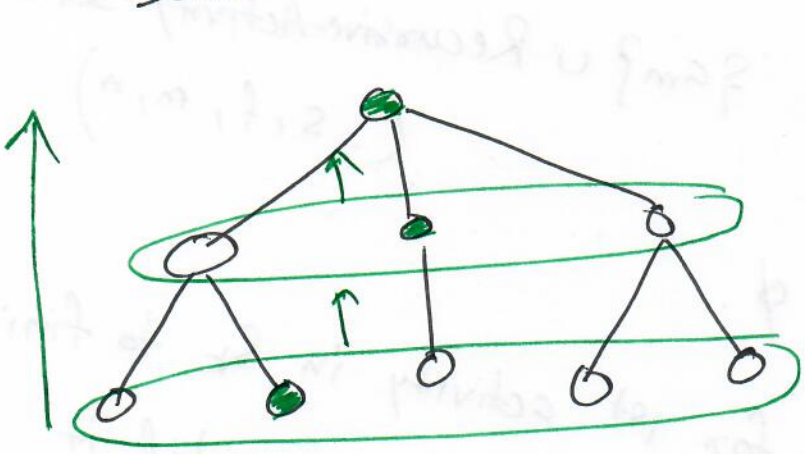
it shows that it is possible to keep max. mut. compat. set by introducing $a_m \in S_k$

the finishes first, and replacing aj which had earliest finish time in Ak.

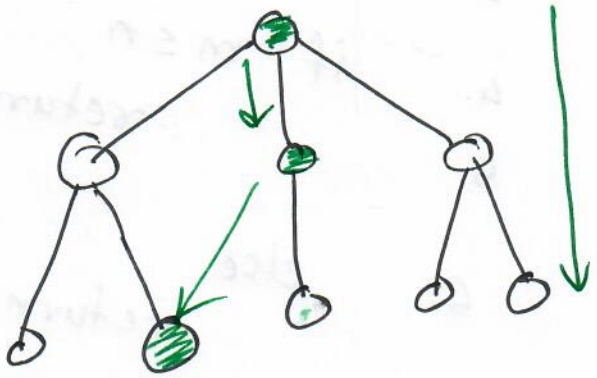
Remarks:

① Could have solved ASP with DP, but don't need. Greed Algo. works: keeps optimal soln

② While going top-down: greed algo selects solutions.



DP: bottom up: look at all subprobs to find best at level



Greedy: top down; pick best at each level while going down

Recursive Greedy Algorithm

Arrays

s: start time

f: finish time

n: no. of activities (size of the prob.)

(*) Assume that n input activities are already ordered by increasing finish time.

(*) Add an additional activity a_0 with $f_0 = 0$.

S_0 : entire set of activities

Recursive-Activity-Selector (s, f, k, n)

```
1.  m = k+1
2.  while m ≤ n and s[m] < f[k]
3.        m = m+1
4.  if m ≤ n
5.        return {am} ∪ Recursive-Activity-Selector
                        (s, f, m, n)
6.  else
        return ϕ.
```

Line 2-3: Look for 1st activity in S_k to finish
Examine $a_{k+1}, \ldots, a_n$ until it finds first activity a_m that is compatible with a_k; such an activity has $s_m \geq f_k$.

For activity-selection-Problem, in page 138 we mentioned two more greedy choices.

Q Does the two greedy choice 2) & 3) in page 138 lead to optimal sol'n ?

A. For the approach of selecting the activity of least duration from those that are compatible with previously selected activities, we give an example to show that this greedy strategy does not yield optimal solution:

activity	1	2	3
s_i	0	2	3
f_i	3	4	6
duration	3	2	3

Here the activity that finishes in least duration is the activity a_2 with start time $s_2 = 2$ and finish time $f_2 = 4$. However, optimal soln selects a_1 and a_3.

For the approach of selecting the compatible activity that overlaps the fewest other remaining activities: we consider the following counter example:

activity	1	2	3	4	5	6	7	8	9	10	11
s_i	0	1	1	1	2	3	4	5	5	5	6
f_i	2	3	3	3	4	5	6	7	7	7	8
# of overlapping activities	3	4	4	4	4	2	4	4	4	4	3

In this approach, one would select a_6, because it overlaps with the least no. of activities, after this selection, the remaining choices are atmost two other activities, i.e., one of a_1, a_2, a_3, a_4 and one of $a_9, a_{10}, a_{11}, a_{12}$. But an optimal soln contains 4 activities: $\{a_1, a_5, a_7, a_{11}\}$.

Steps for Greedy Strategy

1. Determine the optimal substr. of the problem
2. Develop a recursive soln
3. Show that if we make the greed choice, only one subprob. remains
4. Develop a recursive algo that implements the greedy strategy
5. Convert the recursive algo to an iterative algo.

Design greedy algo according to the sequence:

(1) Cast the opt. problem as one in which we make a choice and we are left with one subprob. to solve

(2) Prove that there is always an optimal sol^{n} to the original prob. that makes the greedy choice.

(3) Demonstrate optimal substr. by showing that having made the greedy choice, what remains is a subprob. with the property that if we combine an optimal sol^{n} to the subprob. with the greedy choice we have made, we arrive at an optimal sol^{n} to the orig. prob.

Greedy Choice Property: We can assemble a globally optimal sol^{n} by making locally optimal choices. Make a choice that looks best in current problem, without considering results from subprobs.

Greedy or Dynamic

0-1 knapsack problem

Thief robbing a store finds n items. The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry atmost w pounds in his knapsack, for some integer w. Which items should he take? This is called 0-1 knapsack problem because for each item, the thief must either take it or leave it behind. He cannot take fractional amount of an item or take an item more than once.

Fractional knapsack problem

Setup is same as 0-1 knapsack problem, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item

For eg: 1 item is 0-1 knapsack : gold bar is like gold dust
" Frac. " : ↑ fractional item

Fractional knapsack : Can be solved by greedy strategy

0-1 knapsack : can't solve with greedy

Greedy approach for fractional knapsack

1) Compute value per pound v_i/w_i for each item

2) Thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted, and he can still carry more, s/he takes as much as possible of the item with the next greatest value per pound and so on.

Matroids and Greedy Methods

Matroid theory: situations where greedy method yields optimal sol'n.

Matroid: 1) a combinatorial str.
2) does not cover all cases
↑ means: not all greedy algo may have Matroid str.

Matroids

$M = (S, I)$ satisfying:

1) S is a finite set
2) I, non-empty family of subsets of S called independent subsets of S, s.t., if $B \in I$ and $A \subseteq B$, then $A \in I$. In this case, I is called hereditary. $\phi \in I$.
3) If $A \in I$, $B \in I$, and $|A| < |B|$, then there exists some element $x \in B \setminus A$ s.t. $A \cup \{x\} \in I$. M satisfies exchange property.

Example (Linear Matroid)

As a simple example, consider the following:

$$S = \left\{ \begin{pmatrix}1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\end{pmatrix} \right\}$$

$$I = \left\{ \{\}, \left\{\begin{pmatrix}1\\0\\0\end{pmatrix}\right\}, \left\{\begin{pmatrix}0\\1\\0\end{pmatrix}\right\}, \left\{\begin{pmatrix}0\\0\\1\end{pmatrix}\right\}, \left\{\begin{pmatrix}1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\end{pmatrix}\right\}, \left\{\begin{pmatrix}1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\end{pmatrix}\right\}, \left\{\begin{pmatrix}0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\end{pmatrix}\right\} \right\}$$

1) S is a finite set: $|S| = 3$

2) Define independence of elements of S to be "independence of vectors". (Recall linear independence of vectors from Lin. alg.)

I: set of all lin. ind. set of S.

For any $B \in I$, for example

$$B = \left\{ \begin{pmatrix}1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\end{pmatrix} \right\}$$

and $A = \left\{ \begin{pmatrix}1\\0\\0\end{pmatrix} \right\} \subseteq B$, we do have $A \in I$.

Consider another $B \in \mathcal{I}$,

$$B = \left\{ \begin{Bmatrix} 1 \\ 0 \end{Bmatrix}, \begin{Bmatrix} 0 \\ 1 \end{Bmatrix}, \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \right\}.$$

and any $A \subseteq B$, say,

$$A = \left\{ \begin{Bmatrix} 1 \\ 0 \end{Bmatrix}, \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \right\}$$

then indeed $A \in \mathcal{I}$.

So, hereditary property holds. $\emptyset \in \mathcal{I}$.

3) Let $A = \left\{ \begin{Bmatrix} 1 \\ 0 \end{Bmatrix}, \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \right\}$

$$B = \left\{ \begin{Bmatrix} 1 \\ 0 \end{Bmatrix}, \begin{Bmatrix} 0 \\ 1 \end{Bmatrix}, \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \right\}$$

we have: $|A| = 2 < 3 = |B|$.

we have $B \setminus A = \left\{ \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \right\}$, Choose $x \in B/A$.

Clearly, $A \cup x = A \cup \left\{ \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \right\}$

$$= \left\{ \begin{Bmatrix} 1 \\ 0 \end{Bmatrix}, \begin{Bmatrix} 0 \\ 1 \end{Bmatrix}, \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \right\} \in \mathcal{I}.$$

One can check this for all such $A \& B$

$\Rightarrow$ the ~~matroid~~ $M = (S, \mathcal{I})$ is a matroid.

As another example: Graphic Matroid.

$M_G = (S_G, I_G)$ defined in terms of a given undirected graph $G = (V, E)$.

- S_G = set of edges E of G.
- If $A \subseteq E$, then $A \in I_G$ if and only if A is acyclic. That is, a set of edges A is independent if and only if the subgraph $G_A = (V, A)$ forms a forest.

Th. If $G = (V, E)$ is an undirected graph, then $M_G = (S_G, I_G)$ is a matroid.

Pf. Clearly $S_G = E$ is a finite set.

Claim 1: I_G is hereditary since a subset of a forest is a forest, removing edges from acyclic set of edges cannot create cycles.

Claim 2: M_G satisfies the exchange property

Let $G_A = (V, A)$, $G_B = (V, B)$ be forests of G & that $|B| > |A|$. That is, B contains more edges than A does.

Fact: Any $F = (V_F, E_F)$ ← forest contains exactly $|V_F| - |E_F|$ trees.

$$|E_F| = \sum_{i=1}^{t} e_i$$

v_i : no. of vertices of the i^{th} tree

e_i : no. of edges of the i^{th} tree

$$|E_F| = \sum_{i=1}^{t} (v_i - 1) = \sum_{i=1}^{t} v_i - t$$

$$= |V_F| - t$$

$$\Rightarrow t = |V_F| - |E_F|$$

$\Rightarrow$ G_A contains $|V| - |A|$ trees

& G_B " $|V| - |B|$ trees.

But since $|B| > |A|$, forest G_B has fewer trees than forest G_A does, forest G_B must contain some tree T whose vertices are in two different trees in forest G_A.

Since T is connected, it must contain an edge (u, v) s.t. u & v are in different trees in G_A. Since (u, v) connects vertices in two different trees in G_A, we can add the edge (u, v) to G_A without creating a cycle.

$\Rightarrow$ M_G satisfies the exchange property.

$\Rightarrow$ M_G is a Matroid. ∎

Def: (Extension) Given $M = (S, I)$, we call an element $x \notin A$ an extension of $A \in I$, if we can add x to A while preserving independence, i.e., x is an extension of A if $A \cup \{x\} \in I$.

Example Consider graphic Matroid M_G. If A is an independent set of edges, then edge e is an extension of A if and only if e is not in A and the addition of e to A does not create a cycle.

Def: (Maximal) If A is an independent subset in a Matroid M, we say A is maximal if it has no extensions.

Thm All maximal independent subsets in a Matroid have the same size.

Pf: Spse to the contrary that A is a maximal independent subset of M and there exists another larger ind. subset B of M. From exchange prop.: For some $x \in B \setminus A$, we can extend A to a larger ind. set

$A \cup \{x\}$, contradicting the assumption that A is maximal.

Example M_G, for connected, undirected graph G.

Every maximal ind. subset of M_G must be a tree with $|V|-1$ edges that connects all vertices of G. Such a tree is called spanning tree.

Def: (weighted Matroid)

$M = (S, I)$ is weighted if $\exists$ a weight fn $w(x) > 0$ to each $x \in S$. For $A \subseteq S$

$$w(A) = \sum_{x \in A} w(x)$$

Greedy Algo on a weighted Matroid

Idea:

Find a greedy Algo $\Leftrightarrow$ Find a maximum-weight independent subset in a weighted matroid. Given $M = (S, I)$, find ind. set $A \in I$, s.t. $w(A)$ is maximized.

Defn (optimal subset)

A subset of a Matroid is called optimal subset if it is independent and has maximum possible weight.

optimal subset (=) Maximal ind. subset

Example MST problem

- Given $G = (V, E)$
- Length fn w s.t. $w(e)$ is length of edge e.

[Note: reserve "weight" for Matroid, and "length" for original edge weight of graph]

Goal for MST: Find a subset of the edges that connects all of the vertices and has minimum total length (actually: weight of edges)

Formulate MST interms of problem of finding an optimal subset of a Matroid

It may happen that weights are negative. But for Matroid, we need a positive weight fn. Consider M_G, a matroid with wt. fn w' s.t. $w'(e) = w_0 - w(e)$, where w_0 is larger than the maximum length of any edge.

Hence:

- We ensure that wt. fn is > 0
- optimal subset is a spanning tree of min. total length in original graph.
- Each max. ind. subset A corresp. to a spanning tree with $|V|-1$ edges.

Since $$w'(A) = \sum_{e \in A} w'(e)$$

$$= \sum_{e \in A} w_0 - w(e)$$

$$= (|V|-1)\, w_0 - \sum_{e \in A} w(e)$$

$$= (|V|-1)\, w_0 - w(A),$$ for any ind. subset A, an independent subset that maximizes $w'(A)$ must ~~maximize~~ minimize $w(A)$.

Greedy (M, w)

1) $A = \phi$
2) sort M.S into monotonically decreasing order by weight w
3) for each $x \in M.S$, taken in monotonically decreasing order by weight $w(x)$ if
4) $A \cup \{x\} \in M.I$
5) $A = A \cup \{x\}$
6) return A

Note: A is always ind. by induct.

Let $n = |S|$, then sort time: $O(n \lg n)$

Line 4 executes n times: each execution of line 4 requires a check whether $A \cup \{x\}$ is independent. If each check takes $O(f(n))$,

Total Cost: $O(n \lg n + nf(n))$

Lemma 16.7 (Matroids Exihibit the Greedy Choice Property)

Given $M = (S, I)$, a weighted matroid with weight fn w and that S is sorted into monotonically decreasing order by weight. Let x be the 1st element

of S s.t. $\{x\}$ is independent, if any such x exists. If x exists, then there exists an optimal subset A of S that contains x.

Proof: [Recall similar result for A S P]

- If no such x exists, then the only independent subset is the empty set and lemma is vacously true.
- Otherwise, let B be any nonempty optimal subset. Assume $x \notin B$; otherwise, letting $A = B$ gives an optimal subset of S that contains x.
- No element of B has weight greater than $w(x)$. To see why, observe that $y \in B$ implies that $\{y\}$ is independent, since $B \in I$ and I is hereditary. Our choice of x therefore ensures that $w(x) \geq w(y)$ for any $y \in B$.

[Note: Above paragraph is from Cormen (CLRS book, page 441, 3rd ed. We should pause here and think the following: Given that all elements in S are sorted in monotonically decreasing order by weight, why didn't we directly conclude that $w(x) \geq w(y)$. Why use hereditary property? Answer on next page]

A fine detail that we need to understand here is the following:

$x \in S$ & $\{x\}$ is independent. Moreover, x is the 1st element, meaning x has the max. weight among all $z \in S$ s.t. $\{z\}$ is independent!

So, it may happen that there are many elements $z \in S$ s.t. $\{z\}$ is **not** independent & $w(z) \geq w(x)$.

So, to prove the claim $y \in B$, then $w(x) \geq w(y)$ we must first show using hereditary property that $\{y\}$ is independent. Then among all indepent sets of form $\{z\}$, we know by hypothesis that $w(x) \geq w(y)$.

Now, going back to the proof in CLRS: Our idea here will be similar to the one in activity selection problem: Recall that in activity selection problem we made a greedy choice to select an activity the finishes first among the set of mutually compatible activities. To show this we started with an optimal soln and proved that we can include an activity that finishes first in optimal soln (by replacing other activity). We do same here:

Construct the set A as follows. Begin with $A = \{x\}$. By the choice of x, set A is indep. Using the exchange property, repeatedly find a new element of B that we can add to A until $|A| = |B|$, while preserving the independence of A. At that point, A and B are same except that A has x and B has some other element y. That is,

$$A = B - \{y\} \cup \{x\} \quad \text{for some } y \in B$$

$$\Rightarrow \quad w(A) = w(B) - w(y) + w(x)$$

$$\geq w(B) \quad \left(\begin{array}{l} \text{since } w(x) \geq w(y) \\ \text{i.e } w(x) - w(y) \geq 0 \end{array} \right)$$

Because set B is optimal, set A with $|A| = |B|$, i.e., largest possible ind. set with $w(A) \geq w(B)$ must also be optimal.

Lemma 16.8 $M = (S, I)$ be any Matroid. If x is an element of S that is an extension of some independent subset A of S, then x is also an extension of ϕ.

Pf Since x is an extension of A, $A \cup \{x\}$ is independent (by the def^n of extension). Since $\mathcal{I}$ is hereditary, $\{x\}$ must be independent. Thus, x is an extension of ϕ.

Cor. 16.9 (CLRS) $M = (S, \mathcal{I})$. If $x \in S$ s.t. x is not an extension of ϕ, then x is not an extension of any independent subset A of S.

Pf Let us write this statement as

$$P \Rightarrow Q$$

P: $x \in S$ s.t. x is not an extension of ϕ

Q: x is not an extension of any indepen-dent subset A of S

Taking the contrapositive:

$$\neg Q \Rightarrow \neg P$$

$\neg Q$: x is an extension of some ind. subset A of S

$\neg P$: x is an extension of ϕ

But this statement was proved in previous Lemma 16.8.

Lemma 16.10 (Matroids exhibit the Optimal Substr. property)

Let x be the 1st element of S chosen by GREEDY for the weighted matroid $M = (S, \mathcal{I})$. The remaining problem of finding a maximum-weight independent subset containing x reduces to finding a maximum-weight independent subset of the weighted matroid $M' = (S', \mathcal{I}')$, where

$$S' = \{ y \in S : \{x, y\} \in \mathcal{I} \}$$

$$\mathcal{I}' = \{ B \subseteq S - \{x\} : B \cup \{x\} \in \mathcal{I} \},$$

and the weight fn for M' is the weight function for M, restricted to S'.

Proof: If A is any max.-weight independent subset of M containing x, then $A' = A - \{x\}$ is an independent subset of M'. [Because $A - \{x\} = A' \subseteq S - \{x\}$ & $A' \cup \{x\} = A \in \mathcal{I}$, assumed]

Conversly, any independent subset A' of M' is of the form $A' = A - \{x\}$ s.t. $A' \cup \{x\} = A \in \mathcal{I}$, i.e an ind. subset of M,

In both cases

$$w(A) = w(A') + w(x),$$

a max. weight sol'n in M containing x yields a max. wt. sol'n in M', and vice-versa. ☒

Th. 16.11 (Correctness of the greedy Algo on matroids)

If $M = (S, I)$ is a weighted matroid with weight fn w, then Greedy(M, w) returns an optimal subset.

Proof:

- By Cor. 16.9, any elements that are not extensions of ϕ initially because they can never be useful. Greedy passes over since they can be forgotten about,
- If x is selected by Greedy as 1st element, then Lemma 16.7 ensures that such x will be part of the optimal subset.
- Lemma 16.10 tells us that once an $\{x\}$ is chosen, the remaining problem is one of finding an optimal subset in M' (see def'n of M' above)

This process is then applied repeatedly. The Greedy then finds the first element x' in matroid M'. Clearly, if $\{x'\}$ is independent in M' then $\{x'\} \cup \{x\}$ is independent in M. This way Greedy builds a max. wt. independent subset for M.

Solved Problems

1. Show that (S, I_k) is a matroid, where S is any finite set and I_k is the set of all subsets of S of size atmost k, where $k \leq |S|$.

A. First condition that S is a finite set is given.

Let $k > 0 \Rightarrow I_k$ is nonempty.

Proof of Hereditary prop.: Let $A \in I_k$ $\Rightarrow |A| \leq k$. Let $B \subseteq A \Rightarrow |B| \leq |A|$ $\Rightarrow |B| \leq k \Rightarrow B \in I_k$. This proves her. prop.

Proof of Exchange prop.: Let $A, B \in I_k$ s.t $|A| < |B|$. Choose $x \in B \setminus A$, then

$$|A \cup \{x\}| = |A| + 1 \leq |B| \leq k$$

$$\Rightarrow A \cup \{x\} \in I_k.$$

Q Given an $m \times n$ matrix T over some field, show that (S, I) is a matroid, where S is the set of columns of T and $A \in I$ if and only if the columns in A are Lineasly independent.

A. See example done before.

Q Show how to transform the wt. fn of a weighted matroid problem, where the desired optimal soln is a minimum-wt. maximal independent subset, to make it a standard weighted-matroid problem. Argue.

A. Suppose that W is the largest weight that any one element takes. Then, define the new weight fn

$$w_2(x) = 1 + W - w(x).$$

This assigns a strictly positive weight. Moreoves, any independent set that has maximum wt. w.r.t w_2 will have minimum wt. w.r.t w.

Algorithms

Algorithms

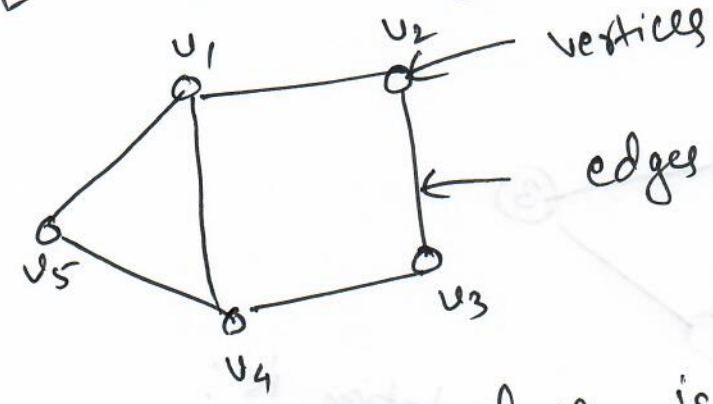
Graph Algorithms

Representation of graphs

$G = (V, E)$

V : set of vertices

E : set of edges



An edge bet^n v_2 and v_3 is denoted by (v_2, v_3).

Two ways to represent a graph:

1) Adjacency lists
2) Adjacency matrix

1) or 2) applies to both directed and undirected graphs

For sparse graphs: adjacency list is a compact representation. $|E| \ll |V|^2$ ($\ll$: much less)

For dense graph: consider adjacency matrix

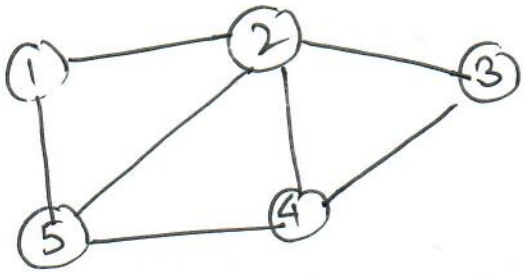
$|E| \approx |V|^2$ ($\approx$: close to)

Adjacency List Representation

For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v s.t. there is an edge $(u, v) \in E$.

$Adj[u]$ = all the vertices adjacent to u in G.

For example:



The adjacency list representation is:

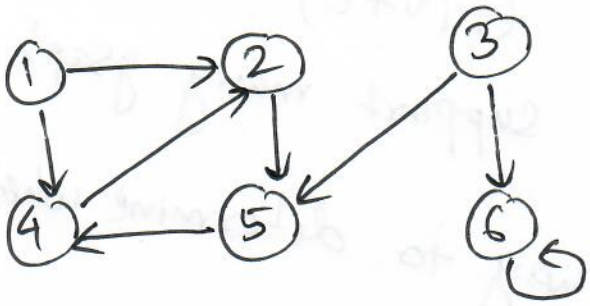
1 2 5
2 1 5 3 4
3 2 4
4 2 5 3

The adjacency matrix representation is:

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Algorithms

Representation of directed graphs:



Adjacency list representation:

1	2	4
2	5	
3	5	6
4	2	
5	4	
6	6	

Adjacency Matrix representation

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Note: Adjacency matrix storage requires more space than Adjacency list.

Memory for Adj list: $\Theta(V+E)$

(++) Adj list can support many graph variants

(--) No quicker way to determine whether (u,v) is present

Adjacency Matrix Repr.

$G = (V, E)$, vertices numbered: $1, 2, \ldots, |V|$

$A = (a_{ij}) \in \mathbb{R}^{|V| \times |V|}$ s.t.

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Memory: $\Theta(V^2)$

Undirected graph $\Rightarrow$ adj. mat. symmetric.
[ok to store entries only above diagonal]

For weighted graph:

$$a_{ij} = \begin{cases} w(i,j) & \text{if } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Breadth First Search

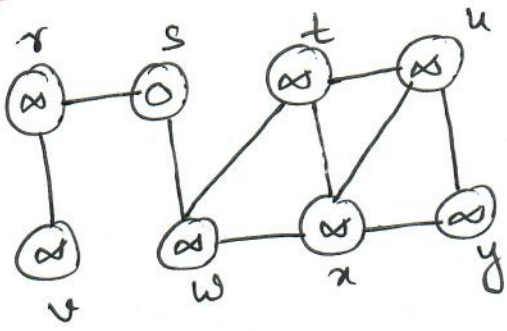
Given $G = (V, E)$, a source vertex s, explore the edges of G to discover every vertex that is reachable from s to every reachable vertex. Also produces breadth-first tree with root s that contains all reachable vertices.

Algorithm:

BFS(G, s)

```
1.  for each vertex u ∈ G.V - {s}
2.      u.color = WHITE
3.      u.d     = ∞
4.      u.π     = NIL
5.  s.color = GRAY
6.  s.d = 0
7.  s.π = NIL
8.  Q = ϕ
9.  ENQUEUE(Q, s)
10. while Q ≠ ϕ
11.     u = Dequeue(Q)
12.     for each v ∈ G.Adj[u]
13.         if v.color == WHITE
14.             v.color = GRAY
15.             v.d = u.d + 1
16.             v.π = u
17.             ENQUEUE(Q, v)
18.     u.color = BLACK
```

Example of Breadth-First Search (BFS)

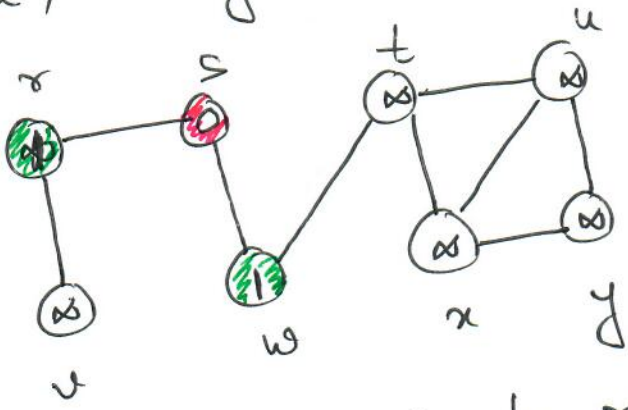


(1) Starting from source s, we find the adjacent vertices.

The adj(s) = r, w

The Queue is Q = | s |
                 | 0 |

(2) After finding the adjacent vertices of s, we color r & w by green (in algo: GREY) and update the distance, meaning distance of r and w from s is 1.



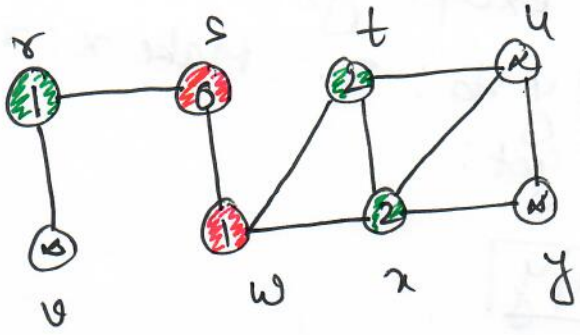
We also pait s to be red (in algo: BLACK) we had dequeued s, then in queue, we enqueue r & w

Q = | w | r |
    | 1 | 1 |

(3) We now dequeue w, and find adjacent vertices of w, adj(w) = t, x, they are now painted green and distance of t & x is now distance of w & + 1. i.e t, x are at distance 2 from source s. Also, vertex w is now painted red (in algo: BLACK).

The status of Queue is:

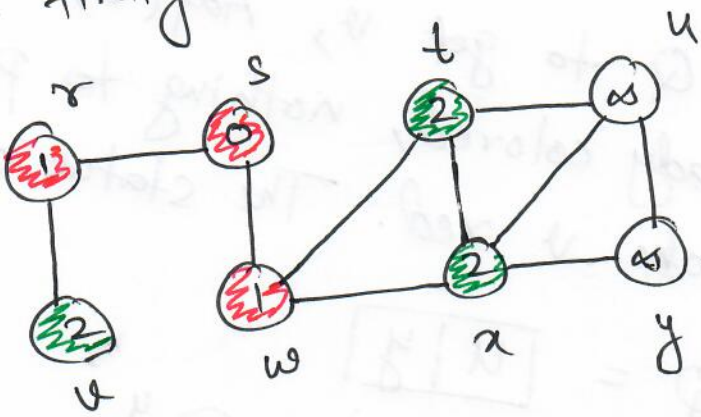
Q = | r | t | x |



(4) we then dequeue to get r. Adj(r) = v, we paint v green, push v in queue, update

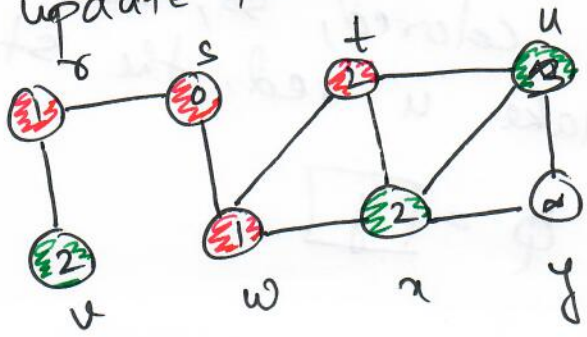
Q = | t | x | v |

the distance of v, which is 2 (= distance of (r) + 1), and finally make r red.



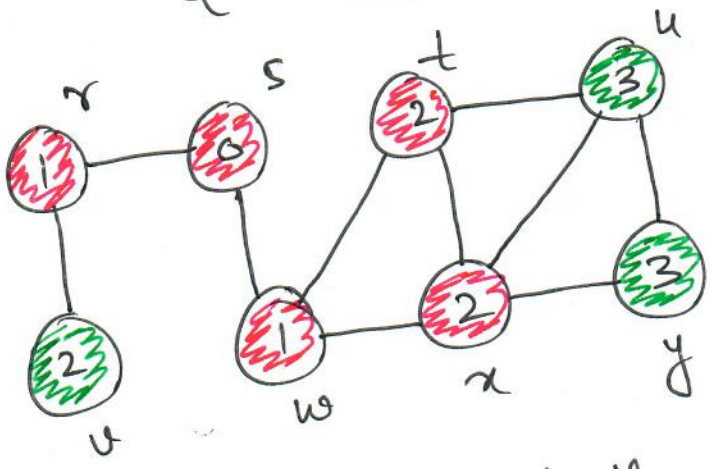
(5) we now dequeue the queue Q to get t. Adj(t) = w, x, u, but w and x are already colored green or red. we enqueue u in queue to get Q = | x | v | u |

we update the distance of u to: 3, & make t red.

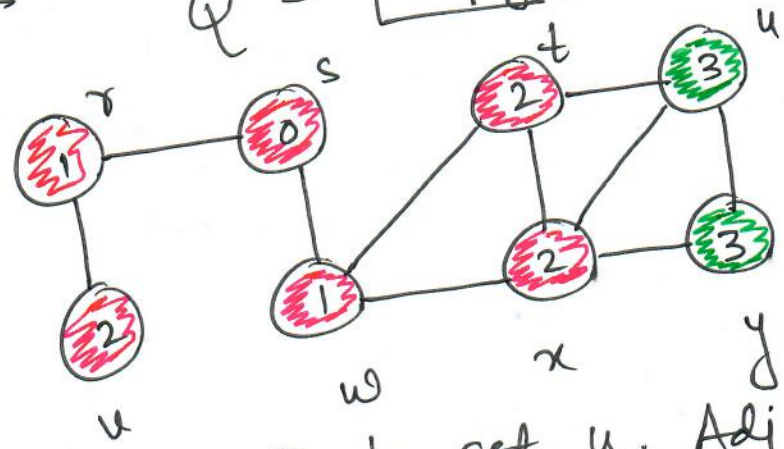


(6) Dequeue Q to get x, Adj(x) = w, t, y, u but all are colored, except y, make it green, update the distance of y to : 3. Make x red, and enqueue y to get:

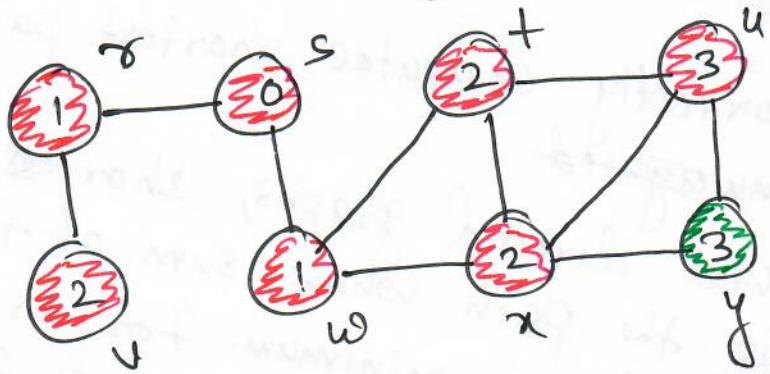
$Q = \boxed{v \mid u \mid y}$



(7) Dequeue Q to get v, Adj(v) = r, but r is already colored, nothing to push in Queue. Make v red. The status of Queue is $Q = \boxed{u \mid y}$

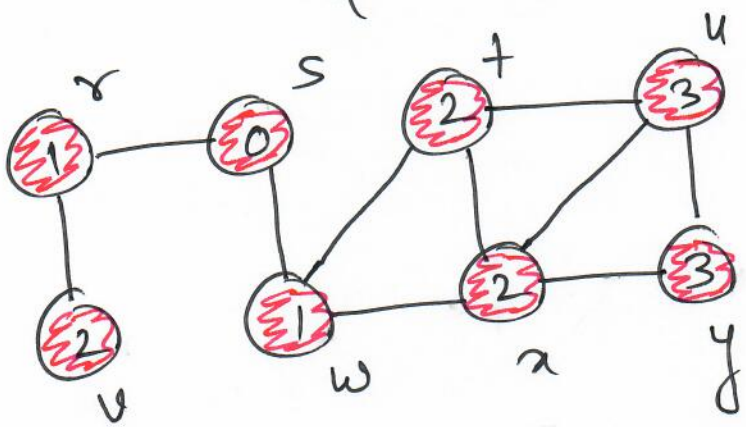


(8) Dequeue Q to get u, Adj(u) = t, x, y but all are colored, so, nothing to enqueue in queue. Make u red, the status of queue is $Q = \boxed{y}$



(9) Finally dequeue Q to get y, Adj(y) = x, u but all are colored, so, nothing to enqueue. Make y red. The status of Q is

$Q = \phi$



Analysis

- Each vertex is enqueued atmost once, and hence dequeued at most once. Enqueuing and dequeuing takes $O(1)$ time. Total time for queue op.: $O(V)$.
- Adjacency list is scanned atmost once for each vertex. Sum of lengths of all adj. lists is $\Theta(E)$. Total time in scanning adj. list $= O(E)$.
- Overhead of initialization: $O(V)$

Total Running time of BFS = $O(V+E)$.

Note: ① BFS correctly computes shortest path distances ~~for~~ ~~unweighted~~

② For weighted directed graphs, shortest path is defined as the path whose sum of the weights of the edges is minimum. For this, we need another algorithm such as Bellman-Ford or Dijkstra's algo.

Algorithms

(5186)

Algorithms

Depth-First Search

- It explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, the search backtracks to explore edges leaving the vertex from which v was discovered. This process continues until all the vertices that are reachable from source have been discovered.
- If any undiscovered vertices remain, then DFS selects one of them as a new source, and it repeats the search from that source.
- As in BFS, DFS colors vertices during the search to indicate their state.
  - Each vertex is initially white
  - A vertex is greyed when it is discovered
  - A vertex is blackened when its adjacency list has been examined completely.
- DFS also time stamps each vertex
  - $v.d$ records when v is first discovered
  - $v.f$ records when the search finishes examining v's adjacency list.

Algorithm for DFS

DFS(G)

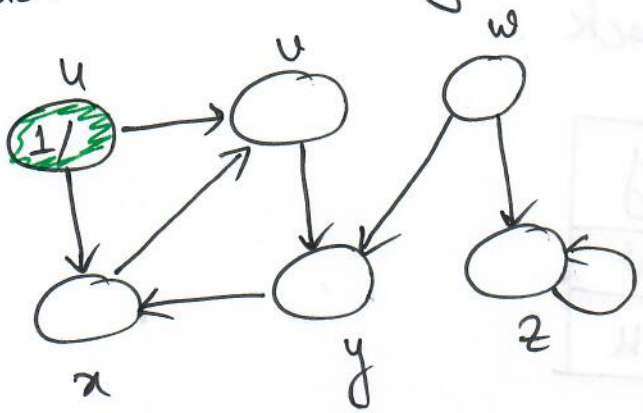
```
1.  for each vertex u ∈ G.V
2.      u.color = WHITE
3.      u.π = NIL
4.  time = 0
5.  for each vertex u ∈ G.V
6.      if u.color == WHITE
7.          DFS-Visit (G, u)
```

DFS-Visit(G, u)

```
1.  time = time + 1
2.  u.d = time
3.  u.color = GRAY
4.  for each v ∈ G.Adj[u]
5.      if v.color == WHITE
6.          v.π = u
7.          DFS-Visit (G, v)
8.  u.color = BLACK
9.  time = time + 1
10. u.f = time
```

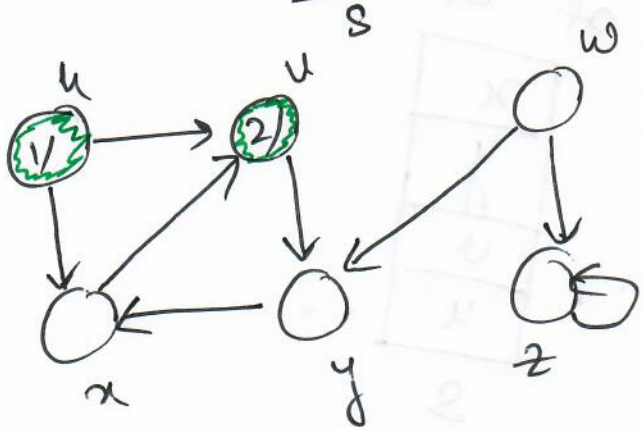
Worked out example of DFS

Consider the following graph:

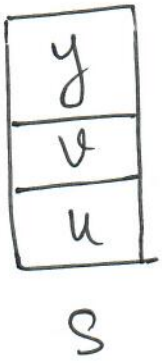


① We start with vertex u, and make it green, we also make a time stamp, i.e., $u.d = 1$. We push u in the stack:

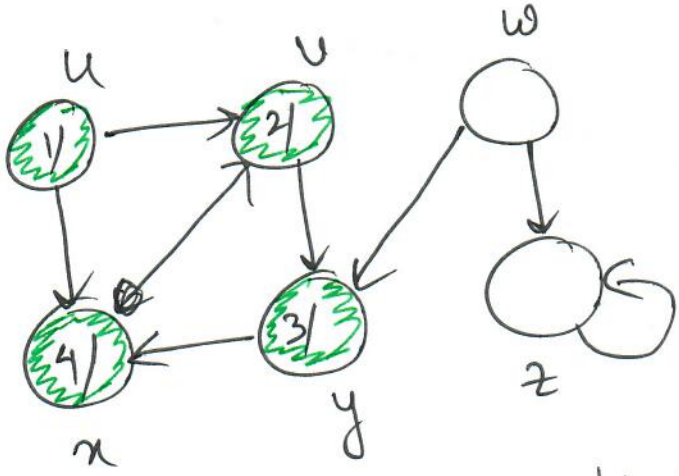
② Explore adjacent vertices of u, $adj(u) = v, x$. ~~Since~~ visit v, make it green, update time $v.d = 2$, push v in stack:



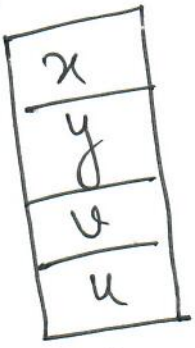
(3) Explore adjacent vertices of v.
Adj(v) = y. Visit y, make it green, update time stamp of y: y.d = 3.
Push y in stack:



S

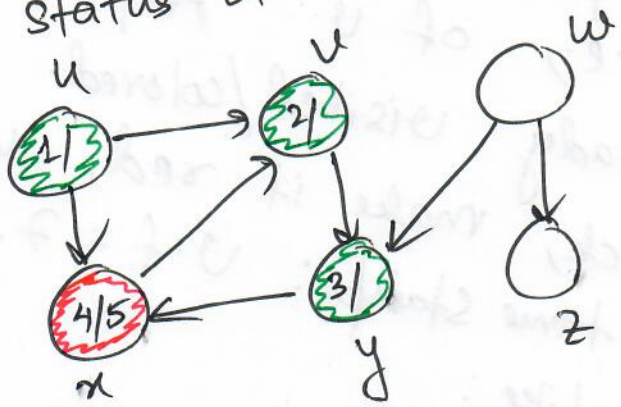


(4) Explore adjacent vertices of top element of stack, i.e of y. Adj(y) = x.
Visit x, make it green, update time stamp of x: x.d = 4. Push x in stack.
The status of stack is:



S

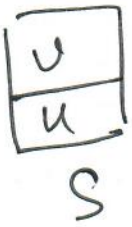
(5) Explore adjacent vertices of top element of stack, i.e., of x. Adj(x) = {v}
But v is already colored, so v cannot be visited, and can't be put in stack (v is already in stack). Also, time stamp of v can't be changed. Since, there is no more adjacent vertices to visit, start backtracking.
Make x red, and pop x out of stack.
The status of stack is:

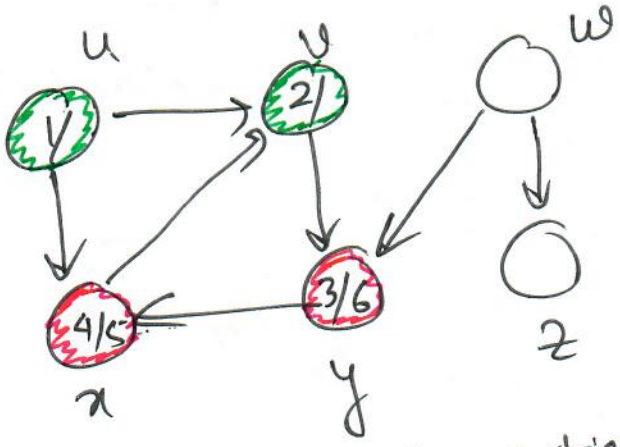


Also, final stamp of x is updated as x.f = 5. The stack is:

(6) Explore the adj. list of top element of stack, which is y. Adj(y) = x, but x is already visited/colored, so nothing to be done. Pop out y from stack, make y red, make final timestamp y.f = 6

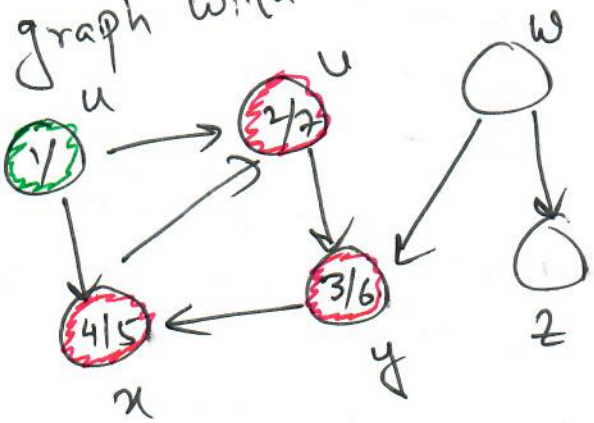
The stack looks like:





(7) Explore adjacent vertices of top element of stack, i.e., of v. Adj(v) = y, but is already visited/colored. Pop out v from stack, make it red, update the final time stamp: v.f = 7. The stack looks like:

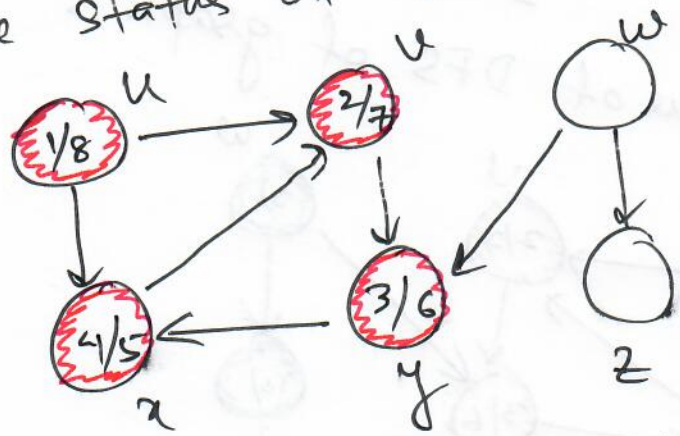
The graph with DFS status:



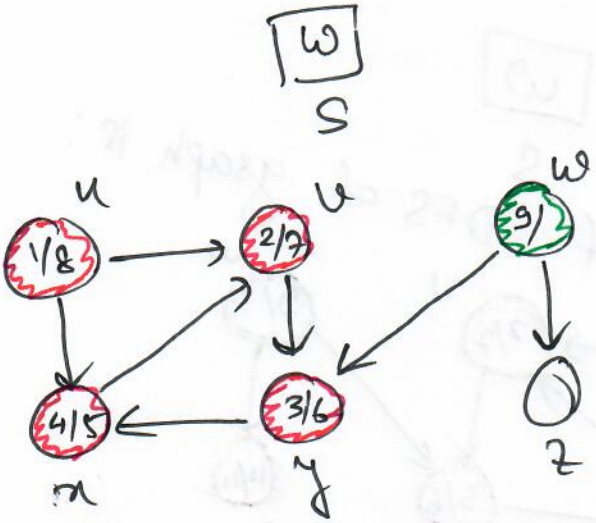
(8) Explore the adj. vertices of top element of stack, which is u. Adj(u) = v, x. But both v and x have been visited/colored. Pop out u from stack, make u red, make final timestamp: u.f = 8.

The status of stack S = ϕ (empty!).

The status of BFS:

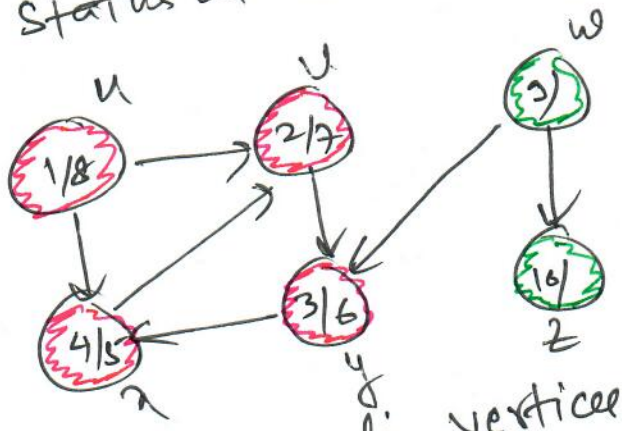


(9) Start DFS with remaining vertices, i.e, either with w or z. Let us start with w. Push w in stack (new stack), make it green, update timestamp of w: w.d = 9.



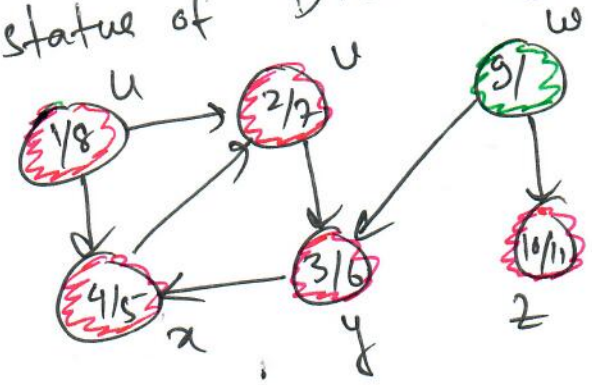
(10) Explore the ~~edges~~ vertices of the top element of stack, which is w. Adj(w) = z, here z is not colored, so visit z, make it green, make timestamp: z.d = 10. push z in stack. The stack is

The status of DFS of graph is



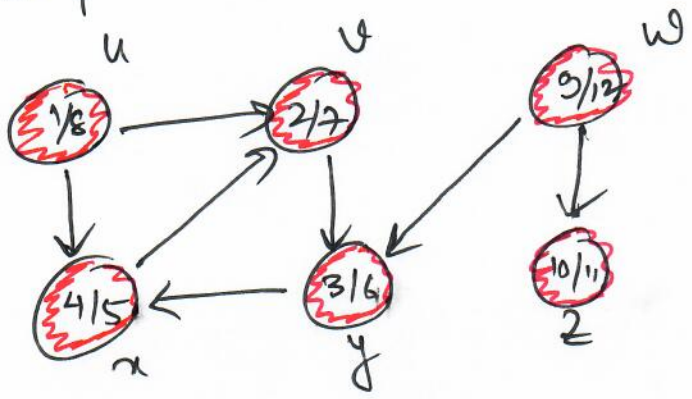
(11) Explore the adj. vertices of the top element of stack. The top element is z. Adj(z) = ϕ. Popout z, make it red, make final timestamp z.f = 11. The stack is:

The status of DFS of graph is:



(12) Explore the adjacency vertices of top element of stack. The top element of stack S : w. Adj(w) = y, z. But both y & z are visited/ colored. Pop out w, make it red, make final timestamp : w.f = 12.

(13)



(13) These are no more vertices left. The DFS stops here.

P138

Algorithms

[2001]

Minimum Spanning Trees

24.09.17 (P1)

Motivation: To interconnect a set of n pins, we can use arrangement of $n-1$ wires, each connecting two pins. The one that uses least amount of wire is usually most desirable.

$G = (V, E)$, V: set of pins, E: set of possible interconnections betn pairs of pins. For each edge $(u,v) \in E$, we have weight $w(u,v)$ specifying the cost to connect u & v.

Goal: Find a acyclic subset $T \subseteq E$ that connects all of the vertices and total weight

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

is minimized.

* Two algorithms: Kruskal's and Prim's
* Each of them are greedy

Kruskal's algorithm

```
MST-KRUSKAL(G, w)
1. A = ∅
2. for each vertex v ∈ G.V
3.     MAKE-SET(v)
4. sort the edges of G.E into nondecreasing order
   by weight w
5. for each edge (u,v) ∈ G.E, taken in nondecreasing
   order by weight.
```

if FIND-SET(u) ≠ FIND-SET(v)
A = A ∪ {(u,v)}
UNION(u,v)

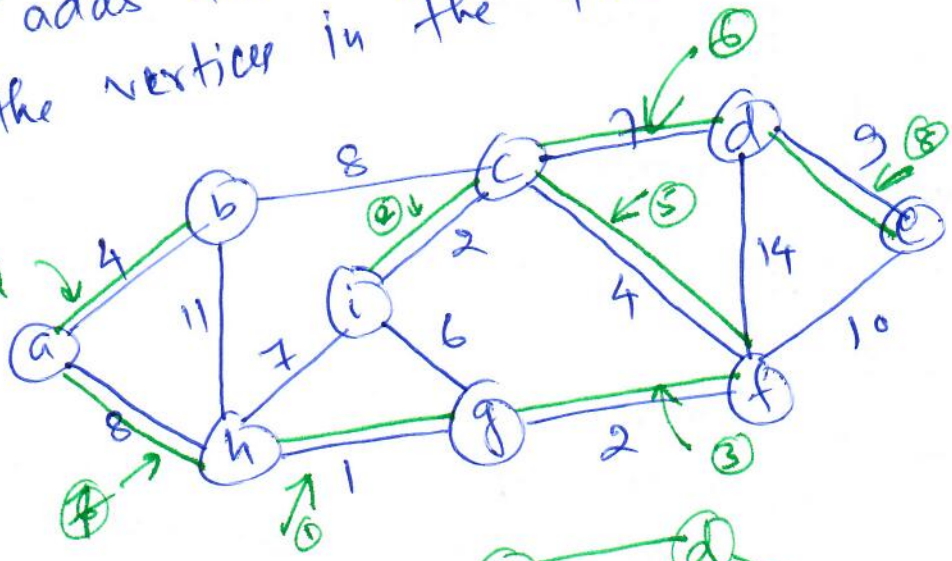
return A

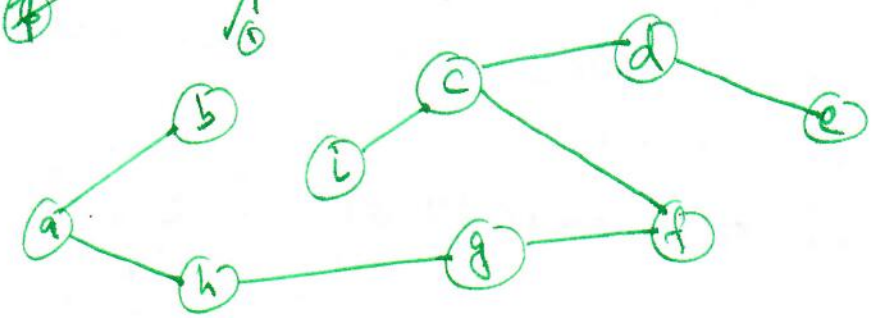
Key steps

(1) Line 1-3 in algo initialize the set A to empty set and create |V| trees, one containing each vertex.

(2) The for loops in line 5-8 examines edges in increasing order of weight, from lowest to highest. The loop cheks for each edge (u,v), whether the endpoints u and v belong to the same tree. If they do, then the edge (u,v) cannot be added the forest without creating a cycle, and edge is discarded. otherwise, line 7 adds the edge (u,v) to A, and line 8 merges the vertices in the two trees.

Example





Running Time of Kruskal's Algorithm

* Depends on how disjoint-set data str. is implemented.
* Assume disjoint-set-forest implementation with union-by-rank and path-compression.

- Line 1: $O(1)$ time
- Line 4: time to sort edges: $O(E \lg E)$.
- Lines 5-8: $O(E)$ Find-Set and Union operations
- Line 3: $|V|$ make-set ops.
- Total: $O((V+E)\alpha(V))$, α is a slowly growing fn.

Since $|E| \geq |V|-1$ (G is connected)

disjoint set ops take: $O(E\alpha(V))$

$\alpha(|V|) = O(\lg V) = O(\lg E)$

Total run time: $O(E \lg E)$.

Since $|E| < |V|^2$, $\lg|E| = O(\lg V)$.

Total run time: $O(E \lg V)$

Minimum Spanning Trees

Prim's Algorithm

* Edges in set A always form a single tree.

Key steps.

1) Tree starts from arbr. root vertex r and grows until the tree spans all the vertices in V.

2) Each step adds to the tree A a light edge that connects A to an isolated vertex - one on which no edge of A is incident.

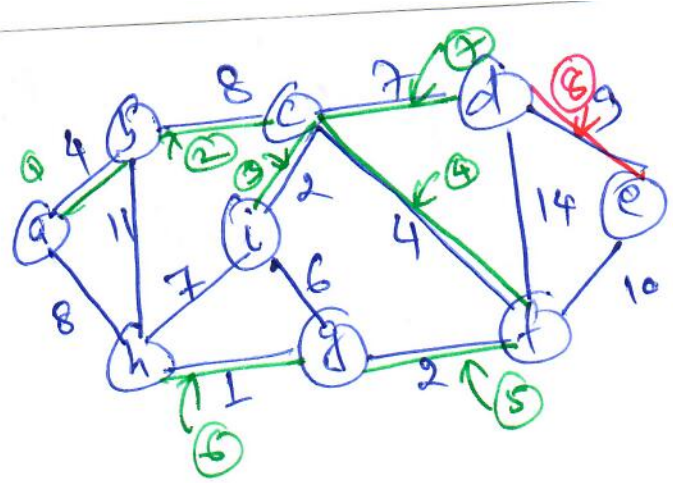
* Need a fast way to select a new edge to add to the tree formed by the edges in A.

```
MST-PRIM(G, w, r)
1.  for each u ∈ G.V
2.      u.key = ∞
3.      u.π   = NIL
4.  r.key = 0
5.  Q = G.V
6.  while Q ≠ ϕ
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π   = u
11.             v.key = w(u,v)
```

Q: Min. priority Queue. based on a key attribute

v.key: min weight of any edge connecting v to a vertex in the tree.

v.key = ∞, if there is no edge.



Running Time

* Depends on how min-priority queue Q is implemented.
* Assum Q is a binary min-heap.

Lines 1-5: $O(V)$ using Build-Min-Heap

Line 6: it executes $|V|$ times, extract-min takes $O(\lg V)$ time, total time: $O(V \lg V)$.

Line 8-11: for loop executes $O(E)$ times

test for membership in Q in line 9 can be implemented in constant time by keeping a bit for each vertex that tells whether or not it is in Q, and updating the bit when the vertex is removed from Q.

Line 11: assignment involves an implicit Decrease-key operation on the min-heap.

A bin-min-heap takes $O(\lg V)$ time for this.

Total time: $O(V \lg V + E \lg V) = O(E \lg V)$

Rk. Asymptotic run time of Prim's algo can be improved by using Fibonacci heaps.

Define min-heap here.

Single-Source Shortest Paths

Problem: Given a weighted, directed graph $G = (V, E)$, with weight fn: $w : E \to \mathbb{R}$ mapping edges to real-valued weights.

The weight $w(p)$ of path $p = \langle v_0, v_1, \ldots v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

shortest-path weight $\delta(u, v)$ from u to v.

$$\delta(u, v) = \begin{cases} \min\{ w(p) : u \overset{p}{\rightsquigarrow} v \} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

shortest path: Any path p with weight $w(p) = \delta(u, v)$

Variants of Shortest path Problem

1) **Single-destination shortest-path problem**: find a shortest path to a given destination vertex t from each vertex v.
Reverse edges to reduce it to a single source problem.

(2) Single-pair shortest-path problem:
Find a shortest path from u to v for given vertices u and v. If we solve the single-source problem with source vertex u, we solve this problem.

(3) All-pairs shortest-paths problem: Find a shortest path from u to v for every pair of vertices u and v.
- Can solve this by running a single source alg. once from each vertex, but can solve it faster.

Optimal substr. of a shortest path

* shortest path betn two vertices contains other shortest paths within it.
* Hence, DP or greedy method might apply.
* Dijkstra's algo is a greedy algo
* Floyd-Warshall algo is a DP algo.

Lemma 24.1 (Subpaths of shortest paths are shortest paths)
Given weighted, directed graph $G = (V, E)$ with weight fn $w : E \to \mathbb{R}$, let

Single-Source Shortest Paths (Graph Algorithms)

$p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and for any i and j s.t. $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then p_{ij} is a shortest path from v_i to v_j.

Proof: Decompose path p into:

$$v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k \longrightarrow (SP1)$$

we have,

$$w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk}).$$

Let there be a path p_{ij}' from v_i to v_j that is shorter than the path p_{ij}, meaning

$$w(p_{ij}') < w(p_{ij}). \longrightarrow (SP2)$$

If we cut-paste this path p_{ij}' in (SP1) then we have:

$$v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}'}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k \longrightarrow (SP2)$$

Now,

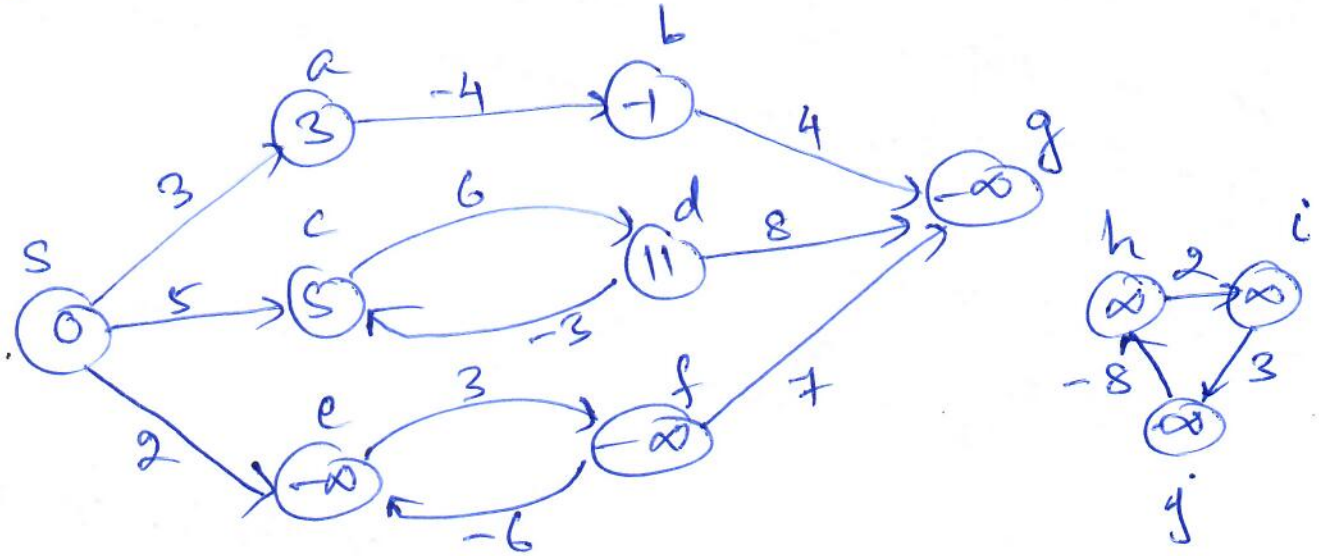
$$w(p') = w(p_{0i}) + w(p_{ij}') + w(p_{jk})$$

$$\overset{(SP2)}{<} w(p_{0i}) + w(p_{ij}) + w(p_{jk})$$

That is, we found a path p' which is shorter than the shortest path p, a contradiction to our hypothesis.

Negative weight Edges and Shortest paths

Consider



1) $\delta(s,a) = w(s,a) = 3$

2) $\delta(s,b) = w(s,a) + w(a,b) = 3 + (-4) = -1$

Paths from s to c: $\langle s,c\rangle$, $\langle s,c,d,c\rangle$, $\langle s,c,d,c,d,c\rangle$ and so on.

cycle $\langle c,d,c\rangle$ has weight $= 6 + (-3) = 3 > 0$

3) $\delta(s,c) = w(s,c) = 5$.

Similarly

4) $\delta(s,d) = w(s,c) + w(c,d) = 11$.

Single-Source Shortest Path

Paths from s to e : $\langle s,e\rangle$, $\langle s,e,f,e\rangle$, $\langle s,e,f,e,f,e\rangle$, and so on.

weight of $\langle s,e\rangle = 2$

" $\langle s,e,f,e\rangle = -1$

" $\langle s,e,f,e,f,e\rangle = -4$.

⋮ ⋮

5) Hence $\delta(s,e) \xrightarrow{\pi} -\infty$. (tends to / equal to.)

6) Similarly $\delta(s,f) = -\infty$

7) Since $\delta(s,f) = -\infty$, $\delta(s,g) = -\infty + 7 = -\infty$.

(optimal substr. property).

8) $\delta(s,h) = \delta(s,i) = \delta(s,j) = \infty$, because they are not reachable from source s.

Cycles

Remarks

i) Dijkstra's algo assume edge weights are nonnegative

2) Bellman-Ford allow negative-weights and produce correct answer as long as no negative weight

cycles reachable from source. Typically algo. can detect -ve weight cycle. (P14)

Cycles and shortest path

1) Non-zero weight cycles lead to ~~-ve~~ $-\infty$ weight or they are useless.

2) This leaves 0-weight cycles. We can repeatedly remove these cycles from the path until we have shortest path that is cycle-free

3) Restrict attention to shortest paths of $|V|-1$ edges.

Representing Shortest Paths

* Wish to compute vertices on shortest-paths.
* Given $G = (V, E)$, maintain for each $v \in V$ a predecessor $v.\pi$ that is either another vertex or NIL.
* Consider predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ induced by π values.

$$V_\pi = \{ v \in V : v.\pi \neq \text{NIL} \} \cup \{ s \}$$

$$E_\pi = \{ (v.\pi, v) \in E : V_\pi - \{ s \} \}.$$

Single Source Shortest Path

* π values (Graph Algo) produced by algos have the property that at termination G_π is a shortest paths tree. A rooted tree containing a shortest path from source s to every vertex that is reachable from s.

Let the shortest path tree rooted at s be $G' = (V', E')$. It is directed.

$V' \subseteq V$, $E' \subseteq E$.

1) V' is the set of vertices reachable from s in G.
2) G' forms a rooted tree with root s, and
3) for all $v \in V'$, the unique simple path from s to v in G' is the shortest path from s to v in G.

RK Shortest paths are not unique.

Relaxation

For each vertex $v \in V$, maintain an attribute $v.d$, an upper bound on the weight of a shortest path from source s to v. Here $v.d$ = shortest path estimate

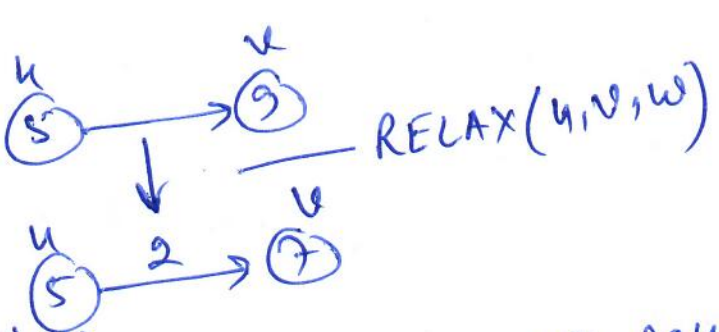
INITIALIZE-SINGLE-SOURCE (G, s)

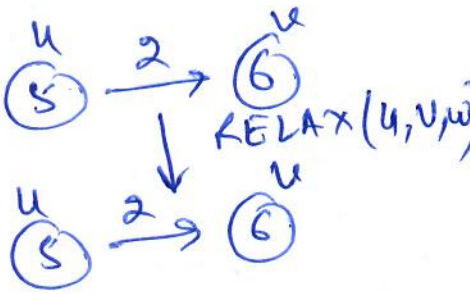
1. for each vertex $v \in G.V$
2. $\quad v.d = \infty$
3. $\quad v.\pi = NIL$
4. $s.d = 0$

Relaxing an edge (u, v) means testing whether we can improve the shortest path to v found so far by going through u and if so updating v.d and v.π.

RELAX (u, v, w)

1) if $v.d > u.d + w(u,v)$
2) $\quad v.d = u.d + w(u,v)$
3) $\quad v.\pi = u$





Each Algo

*) calls INITIALIZE-SINGE-SOURCE
2) Repeatedly relax edges.

Single-Source Shortest (Graph Algos)

Rk :
1) Dijkstra's Algo & shortest paths algo for directed acyclic graphs relax each edge exactly once.
2) Bellman-Ford algo relaxes each edge $|V|-1$ times.

Properties of shortest paths and Relaxation

1) **Triangle inequality** (Lemma 24.10)
For any edge $(u,v) \in E$,
$$\delta(s,v) \leq \delta(s,u) + w(u,v)$$

2) **Upper-bound property** (Lemma 24.11)
We always have
$$\delta(s,v) \leq v.d \quad \forall v \in V,$$
and once $v.d = \delta(s,v)$, $v.d$ never changes.

3) **No-path property** (Cor 24.12)
If there is no path from s to v, then we always have $v.d = \delta(s,v) = \infty$.

Convergence Property: (Lemma 24.14)

If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s,u)$ at any time prior to relaxing edge (u,v), then $v.d = \delta(s,v)$ at all times afterwards.

Path-relaxation property: (Lemma 24.15)

If $p = \langle v_0, v_1, \ldots v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order

$(v_0,v_1), (v_1,v_2), \ldots (v_{k-1}, v_k)$, then

$v_k.d = \delta(s, v_k)$.

This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxation of the edges of p.

Predecessor-subgraph property (Lemma 24.17)

Once $v.d = \delta(s,v)$ $\forall v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Algos for shortest path

Single-Source Shortest-(Graph Algo)

Assume the following conventions

1) for any real $a \neq -\infty$, $a + \infty = \infty + a = \infty$
2) for any real $a \neq \infty$, $a + (-\infty) = (-\infty) + a = -\infty$
3) G is stored in adjacency list representation

Bellman-Ford Algorithm

It solves single-source shortest-paths problem in case in which edge weights may be negative.

Given a weighted, directed graph $G = (V, E)$ with source s & wt. fn $w : E \to \mathbb{R}$, the BF algo. returns:

1) boolean indicating whether or not there is a negative-weight cycle that is reachable from source.

2) if there is no negative-wt. cycle reachable, the algo produces shortest paths and their weights.

BELLMAN-FORD Algo

BELLMAN-FORD (G, w, s)

```
1. INITIALIZE-SINGLE-SOURCE (G, s)
2.  for i = 1 to |G.V| - 1.
3.      for each edge (u, v) ∈ G.E
4.          RELAX (u, v, w)
5.
6.  for each edge (u, v) ∈ G.E
7.      if v.d > u.d + w(u, v)
8.          return FALSE
9.
10. return TRUE
```

Algo relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$.

Steps

(1) Line 1: initialize the d & π values of all vertices.

(2) Line 2: Make $|V| - 1$ passes over the edges of the graph. Each pass consists of relaxing each edge of the graph once.

(3) Lines 5-8: check for a negative weight cycle.

Single-Source Shortest (Graph Algorithms)

Cost: $O(VE)$

Pf Initialization: $\Theta(V)$ (Line 1)

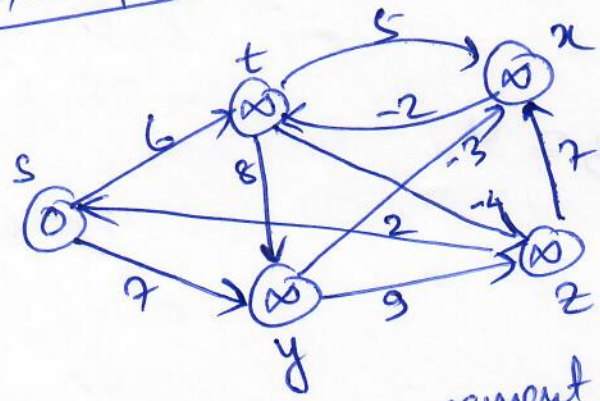
Lines 2-4: $|V|-1$ passes over the edges = ~~$\Theta(VE)$~~ $(|V|-1)\,\Theta(E)$ $= \Theta(VE)$

Lines 5-7: $O(E)$.

Total: $\Theta(V) + \Theta(VE) + O(E)$

$= O(VE)$

Example: Bellman-Ford Algo



Consides the arrangement of edges in the following order:

(t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)

1st pass

(t,x) ~~x~~ $x.d \not> t.d + w(t,x)$ $\infty \not> \infty + 5$

(t,y) $y.d > t.d + w(t,y)$? $\infty \not> \infty + 8$

(t,z) $z.d = \infty \not> \infty + 4$

(x,t) ✗

(y,x) ✗

(y,z) ✗

(z,x) ✗

(z,s) ✗

(s,t) ✓ $t.d = s.d + w(s,t) = 6$

(s,y) ✓ $y.d = 0 + 7 = 7$

2nd pass

(t,x) ✓ $x.d = 11$

(t,y) ✗

(t,z) ✓ $z.d = 2$

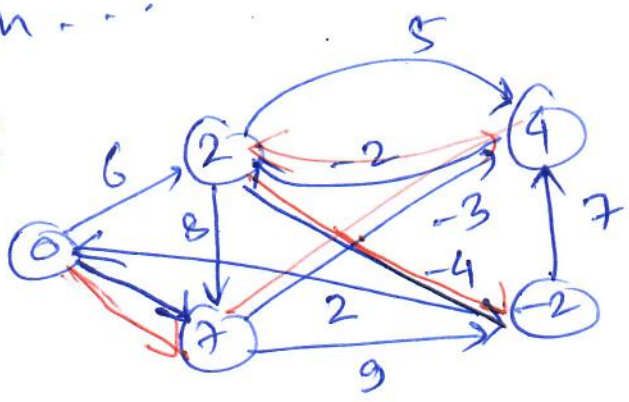
(x,t) ✗

(y,x) ✓ $x.d = 7 - 3 = 4.$

(y,z) ✓ ~~$z.d$~~ =

and so on...

Final Answer



Single-Source Shortest (Graph Algo)

Lemma 24.2

Let $G=(V,E)$ be a weighted, directed graph with source s and weight fn $w: E \to \mathbb{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, after the $|V|-1$ iterations of the for loop of lines 2-4 of BELLMAN-FORD, we have

$$v.d = \delta(s,v)$$

for all vertices v that are reachable from s.

Path-relaxation Property: Consider any shortest path $p = \langle v_0, v_1, \ldots, v_k \rangle$ from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0,v_1), (v_1,v_2), \ldots (v_{k-1},v_k)$, then $v_k.d = \delta(s,v_k)$. Other edge relaxation can be "intermixed".

Pf We recall the path relaxation property:

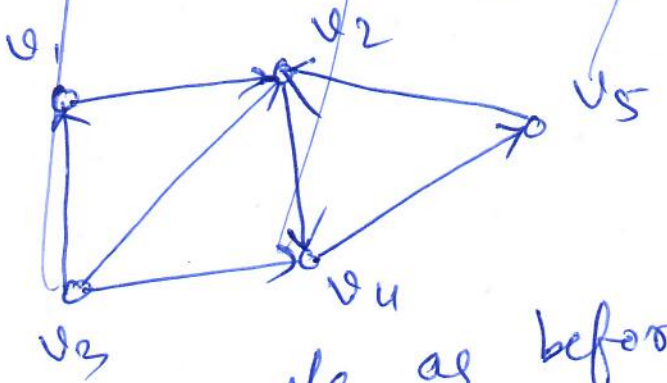
Consider any vertex v that is reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v. Because shortest paths are simple, p has at most $|V|-1$ edges, so $k \le |V|-1$. Each of the $|V|-1$ iterations of the for loop relaxes all $|E|$ edges. Among the edges relaxed in the ith iteration is (v_{i-1}, v_i), for $i = 1, 2, \ldots, k$. By path-relaxation property, $v.d = v_k.d = \delta(s,v)$. ∎

~~CS 2413~~

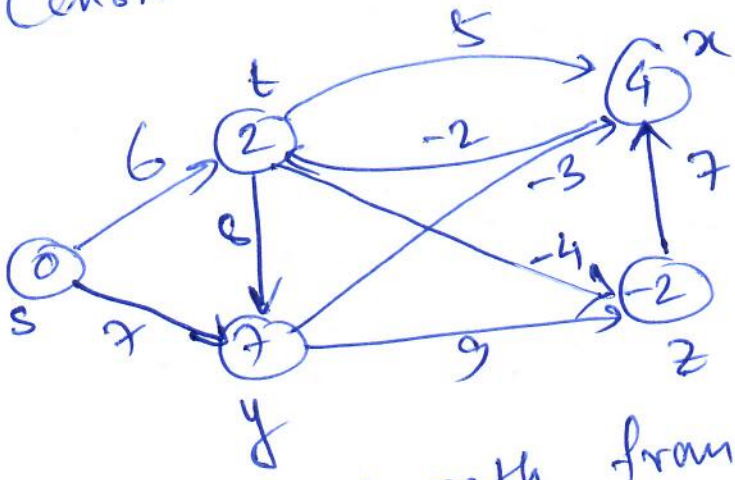
Example.

Let $V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{(v_3, v_1), (v_3, v_4), (v_3, v_2), (v_5, v_2)\}$



Consider the example as before:



The shortest path from s to t is:

(s, y), (y, x), (x, t), i.e, $\langle s, y, x, t \rangle$

In the Bellman-Ford algo, we decide to relax the edges in the following order:

(t, x), (t, y), (t, z), (x, t), (y, x), (y, z),

(z, x), (z, s), (s, t), (s, y).

Since Shortest paths are simple paths, shortest path ~~cannot contain~~ ~~|V|-1~~ can contain atmost |V|-1 edges.

Here, for this example |V|-1 = 5-1 = 4.

Hence, the Bellman-ford algo for loop (Lines 2-4) runs 4 times, ~~then~~ that is,

```
for i = 1 to 4
    RELAX (t,x,w)
    RELAX (t,y,w)
    .
    .
    RELAX (s,y,w)
end
```

Hence all relaxation happens ~~as~~ in following order.

~~RELAX (t,x,w) RE~~

(t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)

(t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)

(t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)

(t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)

So, (s,y), (y,x), (x,t) were relaxed in this order. By relaxation property,

~~$v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$~~

$t.d = \delta(s, t) = 2$, which is the shortest path from s to t.

Further relaxation of (s,y), (y,x), and (x,t) does not change $t.d$ (Convergence property) ∎

Cor. 24.3

Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight fn $w: E \to \mathbb{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, for each vertex $v \in V$, there is a path from s to v if and only if BELLMAN-FORD terminates with $v.d < \infty$ when it is run on G.

Th 24.4 (Correctness of the Bellman-Ford algorithm)

Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source s and weight fn $w: E \to \mathbb{R}$. If G contains no negative-weight cycles that are reachable from s, then the algo returns TRUE, we have $v.d = \delta(s, v)$ $\forall v \in V$, and the predecessor subgraph G_π is a shortest-paths tree rooted at s. If G contains a negative weight cycle reachable from s, then the algo returns FALS

Pf. Spse that G contains no negative weight cycles that are reachable from the source s.

Claim 1): $v.d = \delta(s,v) \quad \forall v \in V$.

1) If v is reachable from s, then Lemma 24.2 proves this claim.

2) If v is not reachable from s, then the claim follows from the no-path property.

Predecessor-subgraph property implies that G_π is a shortest-paths tree.

Claim 2: BELLMAN-FORD returns TRUE.

At termination, we have for all edges $(u,v) \in E$,

$$\begin{aligned} v.d &= \delta(s,v) \\ &\le \delta(s,u) + w(u,v) \\ &= u.d + w(u,v) \end{aligned}$$

$\Rightarrow$ None of the tests in line 6 causes BELLMAN-FORD to return FALSE. Therefore, it returns TRUE.

Now, suppose that G contains a -ve weight cycle that is reachable from source s; let this cycle be

$$c = \langle v_0, v_1, \ldots, v_k \rangle, \text{ where}$$

$v_0 = v_k$. Then,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$

→ SSS

Assume for the purpose of contradiction that the Bellman-Ford algo returns TRUE.

$$\Rightarrow v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$$

$$i = 1, 2, \ldots, k.$$

Summing the inequalities around cycle c gives

$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Since $v_0 = v_k$, each vertex in cycle c appears exactly once in each of the summations $\sum_{i=1}^{k} v_i.d$ and $\sum_{i=1}^{k} v_{i-1}.d$

(SS4)

and so

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d$$

Moreover, by Cor. 24.3, $v_i.d$ is finite for $i = 1, 2, \ldots k$, because since the cycle is reachable, each vertex are reachable. Since $v_i.d$ is finite, we can cancel from the inequality

$$\sum v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

(SS4)

$$= \sum_{i=1}^{k} v_i.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

$$\Rightarrow \sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$$

But it contradicts (SS3). Hence, BELLMAN-FORD algo returns TRUE if graph G contains no negative-weight cycles reachable from the source, and FALSE otherwise.

∎

Dijkstra's Algo

* Solves single-source shortest path problem on a weighted directed graph $G = (V, E)$ in which all edge weights are nonnegative. $w(u,v) \geq 0 \; \forall (u,v) \in E$.

* Run time of Dijkstra's algo is lower than that of Bellman-Ford algo.

key idea

*) Maintain a set S of vertices whose final shortest-path weight from the source s have already been determined.

2) Repeatedly select the vertex $u \in V-S$ with the minimum shortest path estimate, adds u to S, and relaxes all edges leaving u.

```
DIJKSTRA (G, w, s)
1) INITIALIZE-SINGLE-SOURCE (G, s)
2)    S = ∅
3)    Q = G.V
4)    while Q ≠ ∅
5)          u = EXTRACT-MIN(Q)
6)          S = S ∪ {u}
7)          for each vertex v ∈ G.Adj[u]
8)                RELAX (u, v, w).
```

Line 1: initialize d & π value
Line 2: initialize the set S to empty set.
Line 3: initialize the min-priority queue Q to contain all the vertices in V.

Lines 4-8: Line 5 extracts a vertex u from $Q = V - S$ and adds it to set S.
* Vertex u has smallest shortest path estimate of any vertex in $V-S$.

Line 7-8: Relax each edge (u,v) leaving u, thus updating $v.d$ and the predecessor $v.\pi$ if we can improve the shortest path to v found so far by going through u.

Th 24.6 (Correctness of Dijkstra's Algorithm)
Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative wt. fn w and source s, terminates with $u.d = \delta(s,u)$ $\forall u \in V$.

Pf Assignment, CLRS, page 660

Run Time : 1) $O((V+E) \lg V)$ Analysis: p 66 (p 32)

- Min-priority queue with a binary min-heap

2) $O(V \lg V + E)$

- Min-priority queue with a Fibonacci heap.

(Details: CLRS, page 662).

Difference constraints and shortest paths

1) Linear programming that reduce to finding shortest paths from a single source.
2) Solve single-source shortest paths problem by running Bellman Ford algo, thereby solving Linear prog pb.

Linear prog

Given $m \times n$ matrix A, m-vector b, n-vector c. We wish to find a vector x of n elements that maximize the objective fn $\sum_{i=1}^{n} c_i x_i$ subject to the m constraints given by

$$Ax \leq b.$$

Graph Algo

* Simplex does not run in poly time in the size of input.
* these are algo that run in poly tim.
* Single-pair shortest path & max-flow are special cases of LP.

Def^n (feasible sol^n)
Any vector x that satisfies $Ax \leq b$.

System of difference constraints

$Ax \leq b$ represent diff. constraints.

representing $x_j - x_i \leq b_k$.

$1 \leq i, j \leq n$, $i \neq j$ & $1 \leq k \leq m$.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & & & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \leq \begin{bmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{bmatrix}$$

Equivalent to finding values of the unknowns x_1, x_2, x_3, x_4, x_5 satisfying

$$x_1 - x_2 \leq 0$$
$$x_1 - x_5 \leq -1$$
$$\vdots$$
$$x_5 - x_4 \leq -3$$

One sol'n $x = (-5, -3, 0, -1, -4)$.

$x' = (0, 2, 5, 4, 1)$

each component of x' is 5 larger.

Lemma 24.8

Let $x = (x_1, x_2, \ldots x_n)$ be a sol'n to a system of $Ax \leq b$ of diff. constraints and let d be any constant. Then $x + d = (x_1 + d, x_2 + d, \ldots x_n + d)$ is a sol'n. to $Ax \leq b$ as well.

Pf for each x_i & x_j

$$x_j + d - (x_i + d) = x_j - x_i$$

∎

Constraint graphs

Given $Ax \leq b$

Constraint graph: weighted, directed graph

$G = (V, E)$,

$V = \{v_0, v_1, \ldots, v_n\}$

$E = \{(v_i, v_j) : x_j - x_i \leq b_k$ is a constraint$\}$

$\cup \{(v_0, v_1), \ldots, (v_0, v_n)\}$.

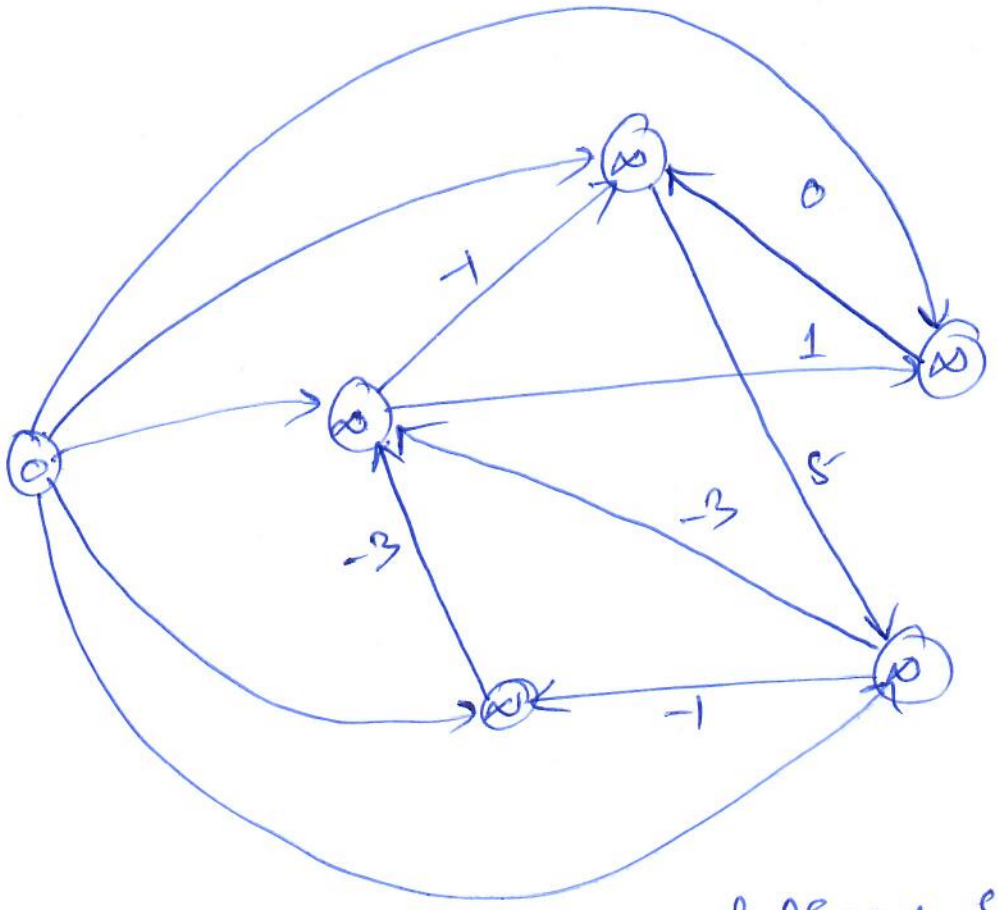
Additional vertex: v_0, to guarantee that the graph has some vertex which can reach all other vertices.

$V = v_i$ for each unknown $x_i + v_0$.

$E =$ an edge for each difference constraint + (v_0, v_i) for each x_i.

*) If $x_j - x_i \leq b_k$ then $w(v_i, v_j) = b_k$.

*) Weight of each edge leaving v_0 is 0.



Th. Given $Ax \le b$ of difference constraints, let $G = (V, E)$ be the corresponding constraint graph. If G contains no negative-weight cycles, then

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \ldots, \delta(v_0, v_n)) \quad (*)$$

is a feasible sol^n. If G contains a negative weight cycle, then there is no feasible sol^n of $Ax \le b$.

Pf. Claim: If the constraint graph contains no negative-weight cycles, then (*) gives a feasible sol^n.

Graph Algo

Consider any edge $(v_i, v_j) \in E$.

By Δle ineq.

$$\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$$

$$\Rightarrow \delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j)$$

By choosing $x_i = \delta(v_0, v_i)$ & $x_j = \delta(v_0, v_j)$

the diff. constraint

$$x_j - x_i \leq w(v_i, v_j)$$

is satisfied.

Claim: If the constraint graph contains a negative weight cycle, then the system of diff. constraints has no feasible sol^n.

W.L.O.G, Let the negative-wt. cycle be

$$c = \langle v_1, v_2, \ldots, v_k \rangle$$

where $v_1 = v_k$. [v_0 not in cycle, because no entering edge to v_0]

$$x_2 - x_1 \leq w(v_1, v_2)$$

$$x_3 - x_2 \leq w(v_2, v_3)$$

$$\vdots$$

$$x_k - x_{k-1} \leq w(v_{k-1}, v_k).$$

Assum x has a sol^n satisfying each of these k inequalities.

If we sum:

$$0 \leq w(c) \qquad \left[\begin{matrix}\text{Note} \\ x_1 = x_k\end{matrix}\right]$$

But since c is a -ve wt. cycle, $w(c) < 0$,
hence, a contradiction.

Complexity: m constraint + n unknowns

$\Rightarrow$ graph with $n+1$ vertices & $n+m$ edges

$$O((n+1)(n+m)) = O(n^2 + nm)$$

Graph Algorithms

Maximum Flow

Motivation

Imagine material coursing through a system from a source, where the material is produced, to a sink, where it is consumed. Source produces the material at some steady rate, and sink consumes the material at the same rate.

flow ≡ rate at which the material moves.

Flow Networks

1) Liquid flowing through pipes
2) parts through assembly lines,
3) current through electrical networks
4) Information through communication networks.

Flow conservation: rate at which material enters a vertex must equal the rate at which it leaves the vertex.

"Kirchoff's current Law".

Max flow Problem

Greatest rate at which we can ship material from the source to the sink without violating any capacity constraints.

Two general Methods

1) Max-flow problem (formalize the notion of flow networks)
2) Classical method of Ford & Fulkerson

Flow Networks

Flow networks and flows

flow network $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has nonnegative capacity $c(u, v) \geq 0$.

if E contains an edge (u, v), then there is no edge (v, u) in the reverse direction. If $(u, v) \notin E$, then define $c(u, v) = 0$. Self-loops not allowed.

~~Source~~

Distinguish two vertices source s and sink t.

Each vertex lies on some path from the source to the sink.

For each vertex $v \in V$, the flow network contains a path $s \rightsquigarrow v \rightsquigarrow t$.

$|E| \geq |V| - 1$ (each vertex other than s has at least one entering edge).

Let $G = (V, E)$ be a flow network with a capacity function c. Let s be the source of the network, and let t be the sink. A flow in G is a real-valued fn $f: V \times V \to \mathbb{R}$ that satisfies the following two properties:

Capacity constraint: $\forall u, v \in V$, we require

$0 \leq f(u,v) \leq c(u,v)$.

Flow conservation: $\forall u \in V - \{s, t\}$, we require

$$\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v)$$

When $(u,v) \notin E$, there can be no flow from u to v, & $f(u,v) = 0$

The value $|f|$ of a flow f is defined as

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$$

↑ total flow out of the source

↑ total flow into the source.

Typically, a flow network will not have any edges into the source, and, therefore, the flow into the source, i.e.,

$$\sum_{v \in V} f(v,s) = 0.$$

Maximum-flow problem: given a flow network G with source s and sink t, find a flow of max^m value.

Capacity constraint: flow from one vertex to another must be nonnegative and must not exceed the given capacity.

flow-conservation property: total flow into a vertex other than the source or sink must equal the total flow out of that vertex — "flow in equals flow out".

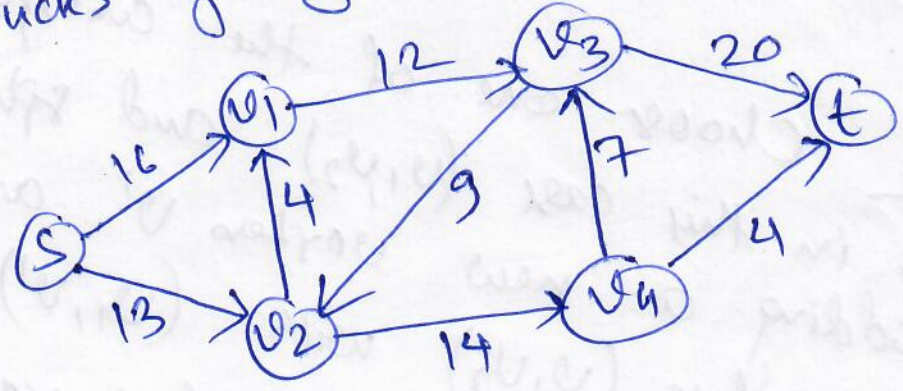
Example of flow

Trucking problem

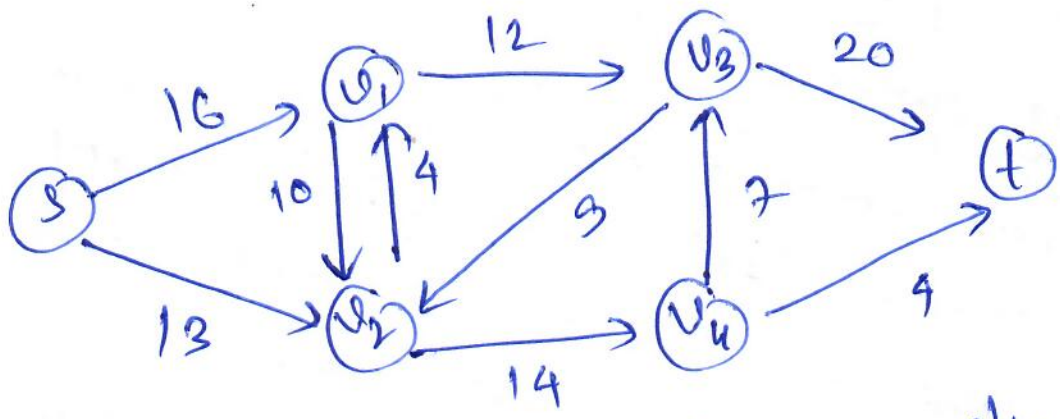
* Lucky Puck company has a factory (source s) in Vancouver that manufactures hockey pucks, and it has a warehouse (sink t) in Winnipeg that stocks them.
* Lucky Puck can ship at most $c(u,v)$ crates per day bet^n cities u & v.
* Need to determine the largest no. p of crates per day that they can ship so that they only produce p crates per day.

Modeling problems with antiparallel edges

If trucking firm offered Lucky Puck the opportunity to lease space for 10 crates in trucks going from Edmonton to Calgary.

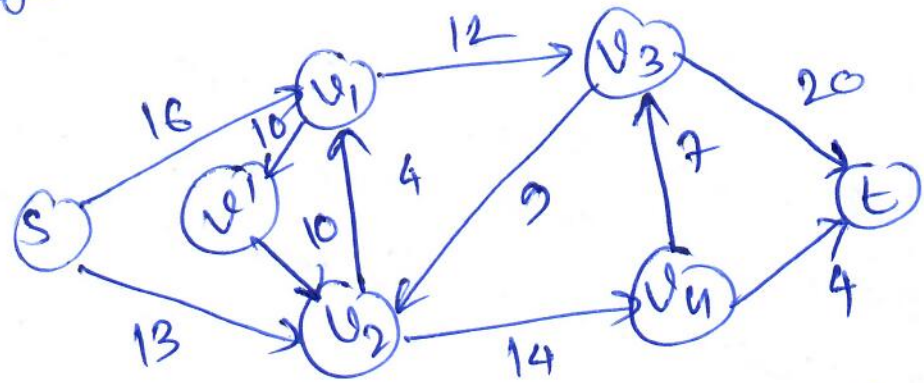


Form a network.



* violates our original assumption that if $(v_1, v_2) \in E$, then $(v_2, v_1) \notin E$. (v_1, v_2) & (v_2, v_1) are called antiparallel.

Tranform a network with antiparallel edges into one that does not contain one.



Trick: Choose one of the antiparallel edges, in this case (v_1, v_2) and split it by adding a new vertex v' and replace edge (v_1, v_2) with (v_1, v') & (v', v_2). The capacities of both edges remain same.

Prove: resulting network is equivalent to original one. (Exercise)

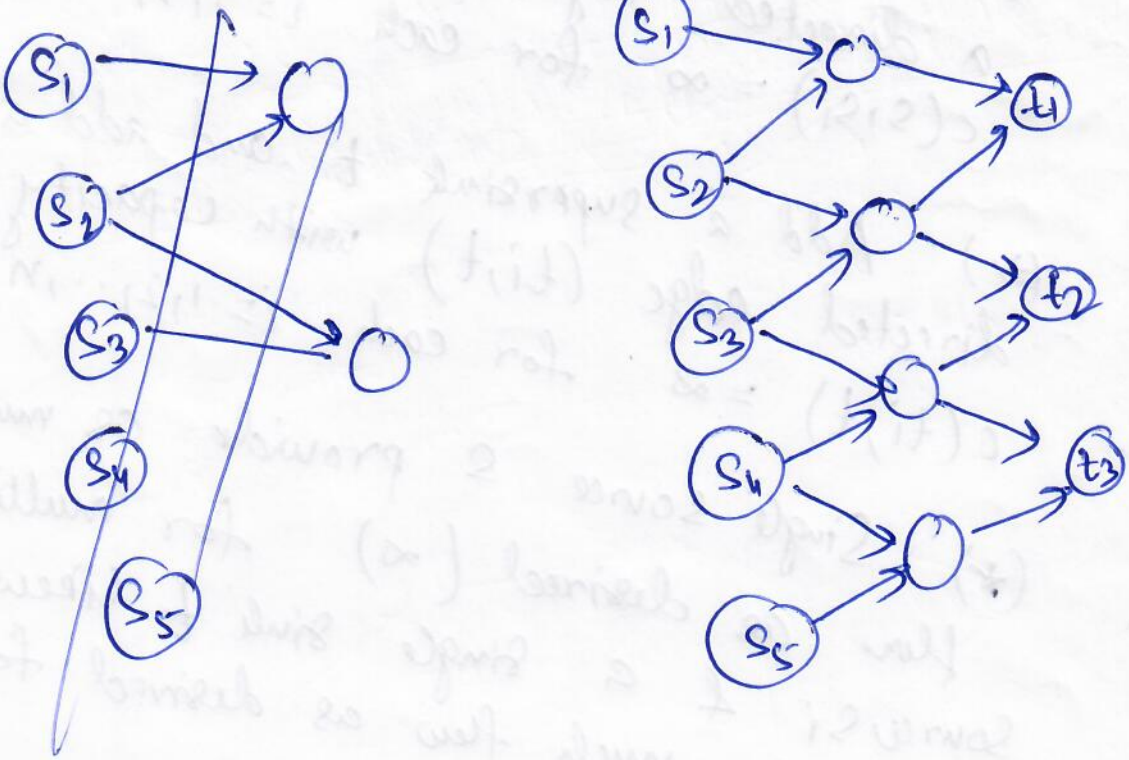
Networks with multiple sources and sinks.

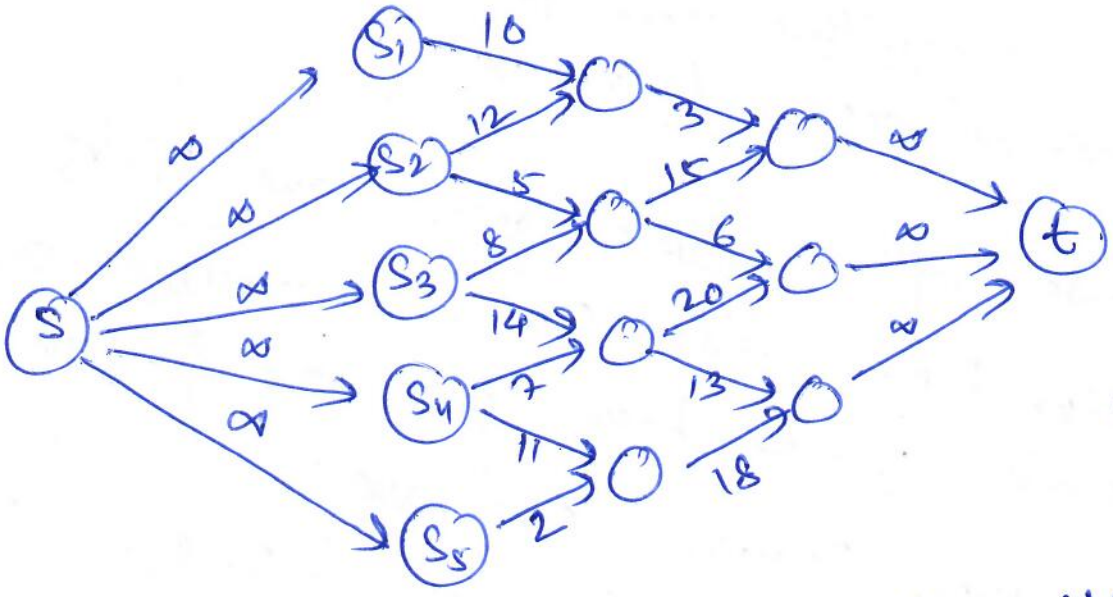
A Max-Flow problem can have multiple sources and sinks, rather than just one of each.

Luck Luck Company can have:

1) set of m factories $\{s_1, s_2, \cdots, s_m\}$
2) set of n warehouses $\{t_1, t_2, \cdots, t_n\}$

we have following.





(*) Can reduce a max-flow network with multiple source & multiple sinks to an ordinary max-flow problem.

(*) Add a supersource s and add a directed edge (s, s_i) with capacity $c(s, s_i) = \infty$ for each $i = 1, 2, \ldots, m$.

(*) Add a supersink t and add a directed edge (t_i, t) with capacity $c(t_i, t) = \infty$ for each $i = 1, 2, \ldots, n$.

(*) Single source s provides as much flow as desired (∞) for multiple sources s_i & a single sink t likewise can use as much flow as desired for the multiple sinks t_i.

Exercise: Prove that two problems are equivalent.

Graph Algorithm (Max Flow)

The Ford-Fulkerson method

Depends on 3 imp. ideas

1) residual networks
2) augmenting paths
3) cuts.

(Max-flow min-cut theorem).

1) Start with $f(u,v) = 0 \quad \forall u, v \in V$.
2) At each iteration, we increase the flow value in G by finding an "augmenting path" in an associated "residual network" G_f.
3) Once the edges of an augmenting path in G_f is known, we can identify specific edges in G for which we can change the flow so that we increase the value of the flow.
4) Repeatedly augment the flow until the residual network has no more augmenting paths.

FORD-FULKERSON-METHOD (G, s, t)

1) initialize flow f to 0
2) while there exists an augmenting path p in the residual network G_f augment flow f along p
3) return f

Residual Networks

Give a flow network G and a flow f, the residual network G_f consists of edges with capacities that represent how we can change the flow on the edges of G.

~~Residual capacity : $c_f(u,v) = c(u,v) - f(u,v)$~~

An edge of the flow network can admit an amount of additional flow equal to the edge's capacity minus the flow on that edge. If that value is positive, we place that edge in G_f with residual capacity of $c_f(u,v) = c(u,v) - f(u,v)$.

* Only edges of G that are in G_f are those that can admit more flow.
* Edges (u,v) whose flow equals their capacity, naturally, have $c_f(u,v) = 0$, and they are not in G_f.
* G_f may also contain edges that are not in G.

(*) To represent a possible decrease of a positive flow $f(u,v)$ on an edge in G, we place an edge (v,u) into G_f with residual capacity

$$c_f(v,u) = f(u,v).$$

an edge that can accept flow in opposite direction to (u,v) at most canceling out the flow on (u,v).

(*) Sending the flow back along an edge is equivalent to decreasing the flow on the edge.

Formal definition of Residual Capacity

Given flow network $G = (V,E)$ with source s and sink t. Let f be a flow in G, and consider a pair of vertices $u,v \in V$. Define residual capacity $c_f(u,v)$ by

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise,} \end{cases}$$

Example if $c(u,v) = 16$, $f(u,v) = 11$

then $f(u,v)$ can be increased upto $c_f(u,v)$, i.e., by 5 units before $c_f(u,v)$ becomes negative. Also, an algorithm should return upto 11 units of flow from v to u, here $c_f(v,u) = 11$.

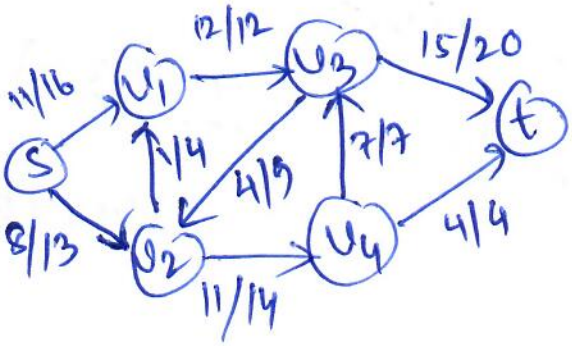
Definition (Residual Network)

Given $G = (V,E)$, flow f, residual network of G induced by f is $G_f = (V, E_f)$

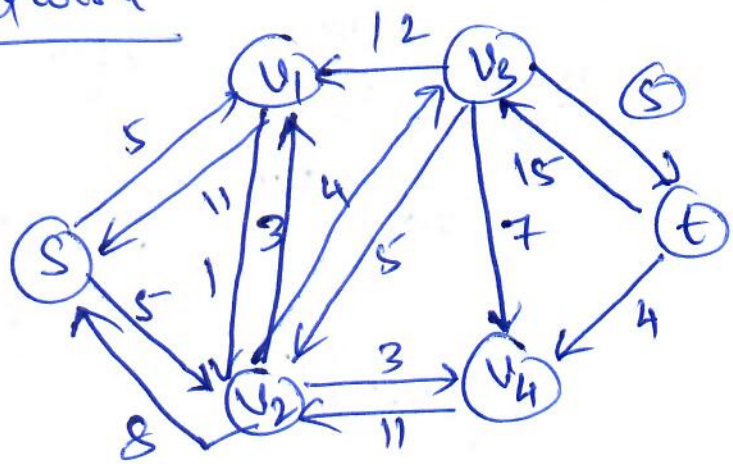
$$E_f = \{ (u,v) \in V \times V : c_f(u,v) > 0 \}.$$

Each edge of residual network, or residual edge can admit a flow > 0.

Example of Residual network



Flow Network



Residual Network

Max Flow

Edges in E_f are either edges in E or their reversals, thus

$$|E_f| \leq 2|E|.$$

(*) Residual network G_f does not satisfy the def^n of flow network, because it may contain both an edge (u,v) & (v,u). Apart from this, residual network has the same properties as flow network.

(*) Residual network is a flow w.r.t capacities c_f in the network G_f.

Augmentation of flow

(*) Flow in residual network provides a roadmap for adding flow to original flow network.

(*) If f is a flow in G and f' is a flow in G_f, define

$f \uparrow f'$, the augmentation of f by f', to be a fn from $V \times V$ to $\mathbb{R}$, defined by

Max Flow

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E, \\ 0, & \text{otherwise} \end{cases} \quad (\star)$$

Intuition: increase the flow on (u,v) by $f'(u,v)$, but decrease it by $f'(u,v)$ because pushing flow on the reverse edge in the residual network signifies decreasing the flow in the original network.

Example: If we sent 5 crates of cricket balls from u to v and send 2 crates from v to u, we could equivalently just sent 3 crates from u to v and none from v to u.

Lemma Let $G = (V, E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residual network of G induced by f, and f' be a flow in G_f. Then the function $f \uparrow f'$ defined in eq^n $(\star)$ above is a flow in G with value

$$|f \uparrow f'| = |f| + |f'|$$

Max Flow

Proof. We first verify that $f \uparrow f'$ obeys the capacity constraint for each edge in E and flow conservation at each vertex in $V - \{s,t\}$.

Recall the properties of flow

1) Capacity constraint: $\forall u,v \in V$, require $0 \le f(u,v) \le c(u,v)$

2) Flow conservation:

$\forall u \in V - \{s,t\}$, require

$$\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v)$$

$(u,v) \notin E$, then can be no flow from u to v, and $f(u,v) = 0$.

Claim 1: Capacity constraint is satisfied.

If $(u,v) \in E$, then $c_f(v,u) = f(u,v)$. $\longrightarrow (**)$

~~$f(v,u)$~~

Since f' is a flow in G_f, it must satisfy the capacity constraint, i.e,

$f'(v,u) \le c_f(v,u)$ ~~$f(u,v)$~~.

From $(**)$ $f'(v,u) \le f(u,v)$

$$\begin{aligned}(f \uparrow f')(u,v) &= f(u,v) \\ &\quad + f'(u,v) - f'(v,u) \\ &\quad f(u,v) + f'(u,v) - f(u,v) \\ &\geq \\ &= f'(u,v). \\ &\geq 0\end{aligned}$$

In addition,

$$\begin{aligned}(f \uparrow f')(u,v) &= f(u,v) + f'(u,v) - f'(v,u) \\ &\leq f(u,v) + f'(u,v) \quad (\text{flows are nonneg.}) \\ &\leq f(u,v) + c_f(u,v) \quad (\text{capacity constraint}) \\ &= \cancel{f(u,v)} + c(u,v) - \cancel{f(u,v)} \quad (\text{def}^n \text{ of } c_f) \\ &= c(u,v)\end{aligned}$$

For flow conservation, because both f and f' obey flow conservation, we have that for all $u \in V - \{s,t\}$,

$$\sum_{v \in V} (f \uparrow f')(u,v) = \sum_{v \in V} \left(f(u,v) + f'(u,v) - f'(v,u) \right)$$

Max Flow

$$= \sum_{v \in V} f(u,v) + \sum_{v \in V} f'(u,v) - \sum_{v \in V} f'(v,u)$$

(flow conservation)

$$= \sum_{v \in V} f(\cancel{u,v}) + \sum_{v \in V} f'(v,u) + \sum_{v \in V} f'(u,v)$$

$$= \cancel{\sum_{v \in V} f(u,v) + f'(v,u) - f'(u,v)}$$

$$= \cancel{\sum_{v \in V} (f \uparrow f')(v,u).}$$

$$= \sum_{v \in V} \left(f(v,u) + f'(v,u) - f'(u,v) \right)$$

$$= \sum_{v \in V} (f \uparrow f')(v,u).$$

Since we disallow antiparallel edges in G, and hence for each vertex $v \in V$, we know that there can be an edge (s,v) or (v,s) but never both.

$v_1 = \{v : (s,v) \in E\}$, a set of vertices with edges from s,

$v_2 = \{v : (v,s) \in E\}$, a set of vertices with edges to s.

We have $V_1 \cup V_2 \subseteq V$, and because we disallow antiparallel edges $V_1 \cap V_2 = \phi$.

we compute

$$|f \uparrow f'| = \sum_{v \in V} (f \uparrow f')(s,v) - \sum_{v \in V} (f \uparrow f')(v,s)$$

$$= \sum_{v \in V_1} (f \uparrow f')(s,v) - \sum_{v \in V_2} (f \uparrow f')(v,s) \longrightarrow (\star)$$

second line follows because

$$(f \uparrow f')(w,x) = 0 \text{ if } (w,x) \notin E.$$

Applying definition of $f \uparrow f'$ to $(\star)$

$$|f \uparrow f'| = \sum_{v \in V_1} (f(s,v) + f'(s,v) - f'(v,s)) -$$

$$\sum_{v \in V_2} (f(v,s) + f'(v,s) - f'(s,v))$$

$$= \sum_{v \in V_1} f(s,v) + \sum_{v \in V_1} f'(s,v) - \sum_{v \in V_1} f'(v,s)$$

$$- \sum_{v \in V_2} f(v,s) + \sum_{v \in V_2} f'(v,s) + \sum_{v \in V_2} f'(s,v)$$

$$= \sum_{v \in V_1} f(s,v) - \sum_{v \in V_2} f(v,s) + \sum_{v \in V_1} f'(s,v) + \sum_{v \in V_2} f'(s,v)$$

$$- \sum_{v \in V_1} f'(v,s) - \sum_{v \in V_2} f'(v,s)$$

$$= \sum_{v \in V_1} f(s,v) - \sum_{v \in V_2} f(v,s) + \sum_{v \in V_1 \cup V_2} f'(s,v) - \sum_{v \in V_1 \cup V_2} f'(v,s).$$

We can extend all four summations to sum over V, since each additional term has value 0. We have

$$|f \uparrow f'| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{v \in V} f'(s,v) - \sum_{v \in V} f'(v,s)$$

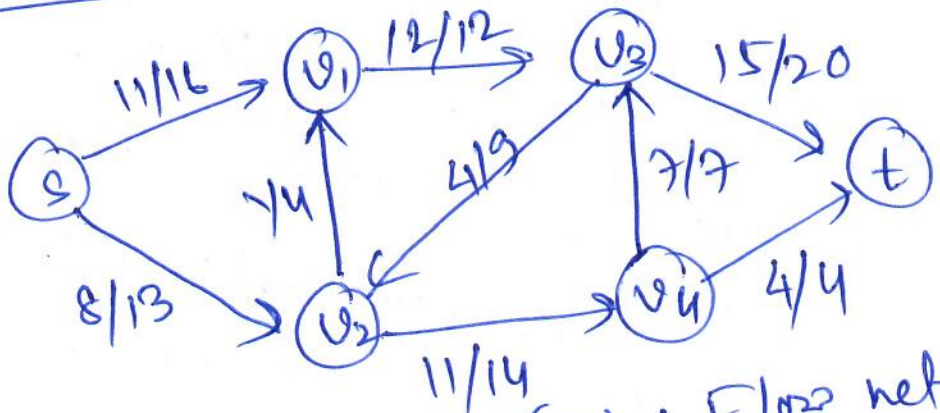
$$= |f| + |f'|.$$

Augmenting Paths

Given a flow network $G_f = (V, E)$ and a flow f, an augmenting path p is a simple path from s to t in the residual network G_f.

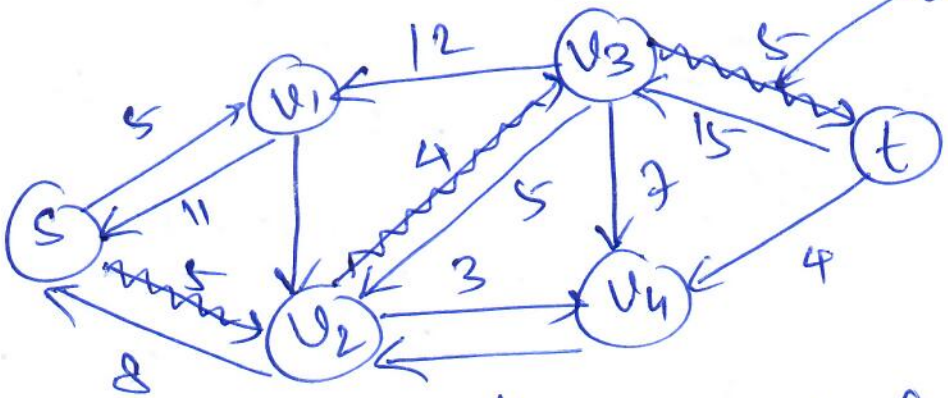
Recall: we may increase the flow on an edge (u,v) of an augmenting path by up to $c_f(u,v)$ without violating the capacity constraint.

Example: Augmenting path



(a): Flow network G.

augmenting path p.



(b): Residual network G_f with augmenting path p.

Treating the residual network G_f in (b) as a flow network, we can increase the flow through each edge of this path by up to 4 units, without

violating capicity constraints, since the smallest residue capacity on this path is $c_f(v_2, v_3) = 4$.

Residual Capacity: Max. amount by which we can increase the flow on each edge in an augmenting path p.

$$c_f(p) = \min\{ c_f(u,v) : (u,v) \text{ is on } p \}.$$

Lemma 26.2

Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Define a fn

$$f_p : V \times V \to \mathbb{R} \text{ by}$$

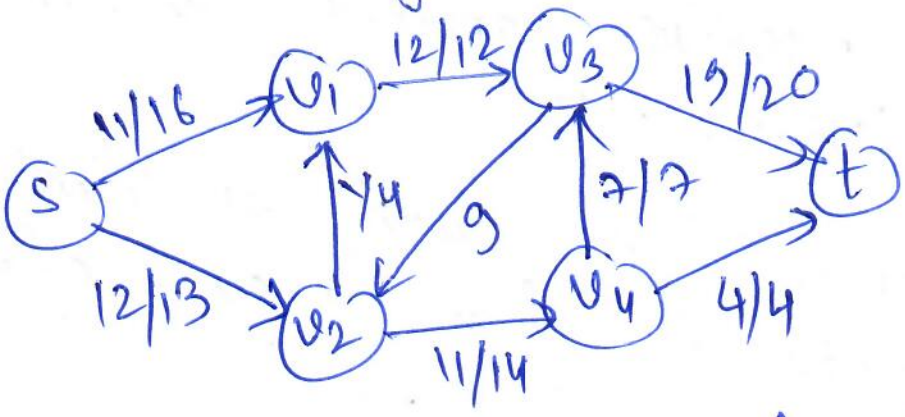
$$f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \text{ is on } p \\ 0 & \text{otherwise.} \end{cases} \quad (**)$$

Then f_p is a flow in G_f with value

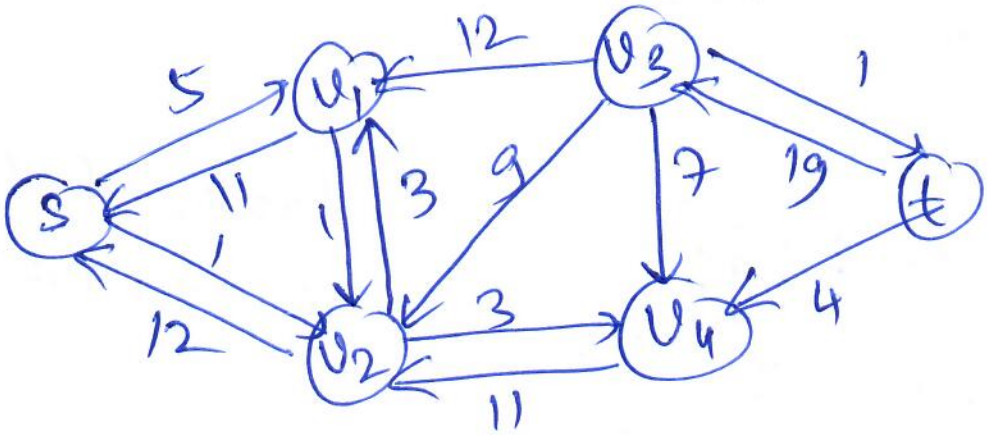
$$|f_p| = c_f(p) > 0.$$

Intuition: If we augment f by f_p we get another flow in G whose value is closer to the maximum.

When we augment the path p,



Corresp. residual network:



Cor: Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Let f_p be defd as in eqn (22), and suppose that we augment f by f_p. Then the fn

$f \uparrow f_p$ is a flow in G with

value $|f \uparrow f_p| = |f| + |f_p| > |f|$.

From prev. Lemma.

$$|f \uparrow f_p| = |f| + |f_p|$$

and from prev. Lemma

$$|f| + |f_p| > |f| \quad (\text{Since } |f_p| > 0)$$

Cuts of flow networks

Ford-Fulkerson method repeatedly augments the flow along augmenting paths until it has found a maximum flow.

Q How to know when the algo terminates we have found max flow?

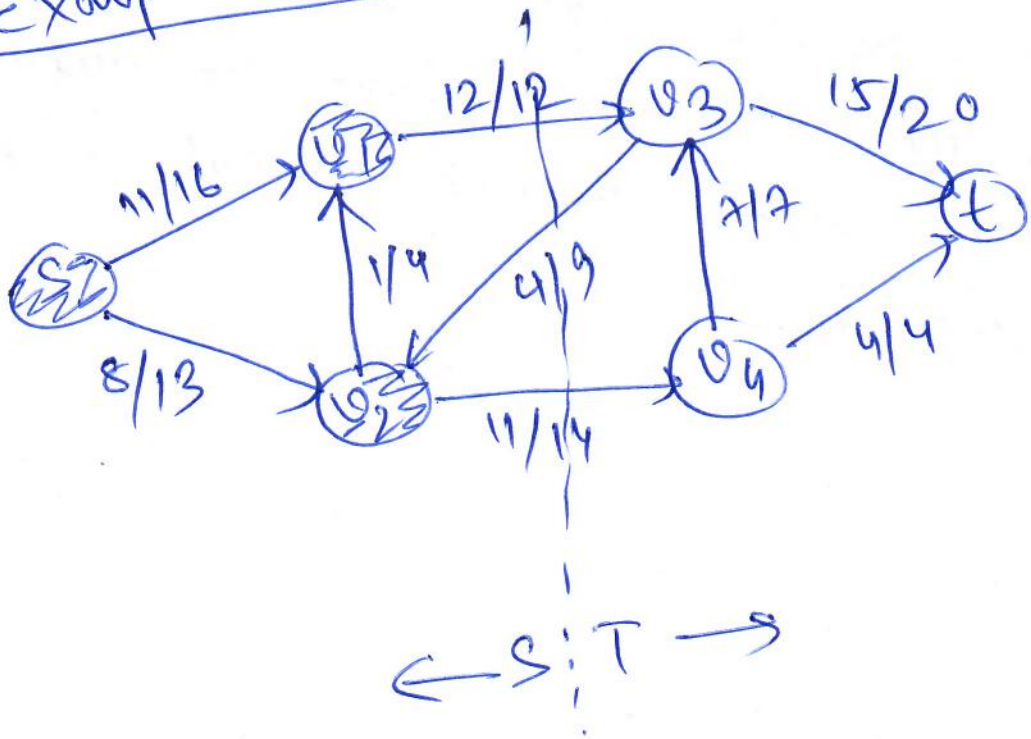
A. Max-flow-min cut theorem, tells us that a flow is max. if and only if its residual network contains no augmenting path. We need notion of a cut of a flow network.

A cut (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ s.t. $s \in S$ and $t \in T$. (Similar to cut used for min. spanning trees, except that here we are cutting directed graph rather than undirected graph)

* If f is a flow, then the net flow $f(S, T)$ across cut (S, T) is defined to be

$$f(S, T) = \sum_{v \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$$

Example of cut (S, T) in the flow network



Max Flow

Here $S = \{s, v_1, v_2\}$, $T = \{v_3, v_4, t\}$. vertices in S are like (S) and vertices in T are (T). The net flow across (S,T) is $f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$.

$\sum_{u \in S} \sum_{v \in T}$ includes ~~vertices~~ edges:

$\{(s,v_3), (s,v_4), (s,t), (v_1,v_3), (v_1,v_4), (v_1,t), (v_2,v_3), (v_2,v_4), (v_2,t)\}$

We have

$\sum_{u \in S} \sum_{v \in T} f(u,v) = 0+0+0+12+0+0+0+11+0$

and

~~Also,~~

$\sum_{u \in S} \sum_{v \in T} f(v,u) = 0+0+0+4+0+0$

Hence $f(S,T) = 23-4 = 19$

and the capacity of the cut is

$$c(v_1, v_3) + c(v_2, v_4) = 12 + 14 = 26.$$

For a given flow f, the net flow across any cut is the same, and it equals $|f|$, the value of the flow.

Lemma Let f be a flow in a network G with source s and sink t, and let (S, T) be any cut of G. Then the net flow across (S, T) is $f(S, T) = |f|$.

Proof Flow conservation condition for any node $u \in V - \{s, t\}$ is:

$$\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0 \longrightarrow (MT-1)$$

We have

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

Summing (MT-1) over all vertices in $S - \{s\}$ gives:

Max Flow

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{u \in S-\{s\}} \left(\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) \right)$$

is 0

is 0.

Expanding the right hand side summation and regrouping term yield

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{u \in S-\{s\}} \sum_{v \in V} f(u,v)$$

$$- \sum_{u \in S-\{s\}} \sum_{v \in V} f(v,u)$$

Fabiy $\sum_{v \in V}$ comment!

$$\sum_{v \in V} \left(f(s,v) + \sum_{u \in S-\{s\}} f(u,v) \right) -$$

$$\sum_{v \in V} \left(f(v,s) + \sum_{u \in S-\{s\}} f(v,u) \right)$$

$$= \sum_{v \in V} \sum_{u \in S} f(u,v) - \sum_{v \in V} \sum_{u \in S} f(v,u).$$

Because $V = S \cup T$ and $S \cap T = \phi$, we can split each summation over V into sums over S and T to obtain.

$$|f| = \sum_{v \in S} \sum_{u \in S} f(u,v) + \sum_{v \in T} \sum_{u \in S} f(u,v) - \sum_{v \in S} \sum_{u \in S} f(v,u)$$

$$- \sum_{v \in T} \sum_{u \in S} f(v,u)$$

$$= \sum_{v \in T} \sum_{u \in S} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u) +$$

$$\left(\sum_{v \in S} \sum_{u \in S} f(u,v) - \sum_{v \in S} \sum_{u \in S} f(v,u) \right).$$

(same!)

$$\Rightarrow |f| = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$

$$= f(S,T)$$

∎

Defn (Min. Cut): A cut whose capacity is minimum over all cuts of the network.

Cor. The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.

Proof: Let (S,T) be any cut of G and let f be any flow. By Lemma 26.4 and the capacity constraint,

$$|f| = f(S,T)$$

$$= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$

$$\leq \sum_{u \in S} \sum_{v \in T} f(u,v) \qquad \left[f(v,u) \geq 0 \right].$$

$$\leq \sum_{u \in S} \sum_{v \in T} c(u,v)$$

$$= c(S,T).$$

$\Rightarrow$ the value of a max. flow in a network is bounded from above by the capacity of a minimum cut of the network.

Intuition: Value of a max. flow is in fact equal to the capacity of a minimum cut.

Th. (Max-flow min-cut thm).
If f is a flow in a network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1) f is a max. flow in G.
2) The residual network G_f contains no aug. paths.
3) $|f| = c(S, \bar{T})$ for some cut $(S, \bar{T})$ of G.

(1) $\Rightarrow$ (2).

Pf.
Assume on contrary that f is a max. flow in G but that G_f has an augmenting path p. By Cor. 26.3, the flow formed by aug. f by f_p where f_p is given by — is a flow in G with value strictly greater than $|f|$, contradicting the assumption that f is max flow.

(2) ⇒ (3)

Suppose that G_f has no augmenting path, that is, that G_f contains no path from s to t. Define

$$S = \{v \in V : \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$$

$$T = V - S.$$

The partition (S, T) is a cut: we have $s \in S$ trivially and $t \notin S$, because there is no path from s to t in G_f.

Consider a pair of vertices $u \in S$ & $v \in T$. If $(u, v) \in E$, we must have

$$f(u, v) = c(u, v)$$

since otherwise $(u, v) \in E_f$, which would place v in set S. If $(v, u) \in E$, we must have $f(v, u) = 0$, because otherwise $c_f(u, v) = f(v, u)$ would be positive and we would have $(u, v) \in E_f$, which would place v in S.

If neither (u,v) nor (v,u) is in E, then $f(u,v) = f(v,u) = 0$.

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u)$$

$$= \sum_{u \in S} \sum_{v \in T} c(u,v) - \sum_{v \in T} \sum_{u \in S} 0$$

$$= c(S,T).$$

By Lemma 26.4,

$$|f| = f(S,T) = c(S,T).$$

(3) $\Rightarrow$ (1): By Cor. 26.5

~~$|f| = f(S,T) = c(S,T)$~~

$|f| \le c(S,T)$ for all cuts (S,T). $\rightarrow$ (MF4)

The cond^n $|f| = c(S,T)$

$\Rightarrow$ that f is max flow because flow cannot exceed $c(S,T)$ from (MF4)

□.

Max Flow

Ford-Fulkerson Algorithm

Idea:

1) In each iteration: find some augmenting path p and use p to modify the flow f.
2) From Lemma 26.2 & Cor. 26.3, replace f by $f \uparrow f_p$, obtaining a new flow whose value is $|f| + |f_p|$.

In the Algo:

1) Given $G = (V, E)$, update flow attribute $(u,v).f$ for each edge $(u,v) \in E$. if $(u,v) \notin E$, assm $(u,v).f = 0$.
2) Assm that capacities $c(u,v)$ are given, $c(u,v) = 0$ if $(u,v) \notin E$.
3) Compute residual capacity:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

FORD-FULKERSON (G, s, t)

for each edge $(u,v) \in G.E$

$(u,v).f = 0$

while there exists a path p from s to t in the residual network G_f

$c_f(p) = \min \{ c_f(u,v) : (u,v) \text{ is in } p \}$

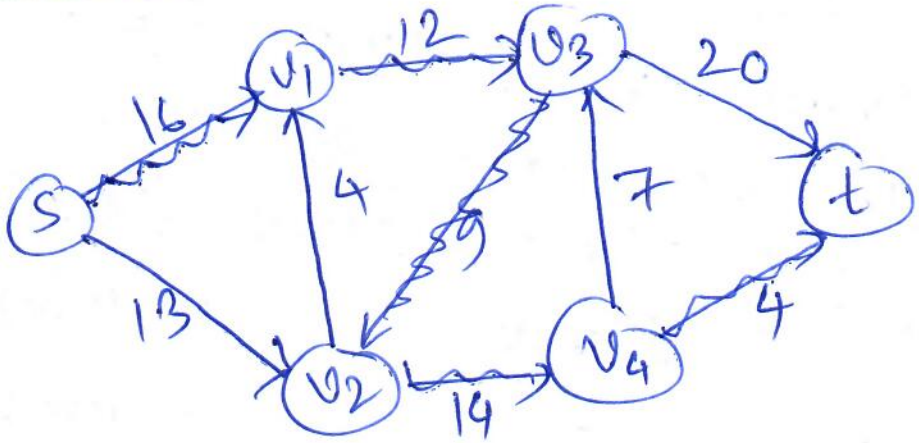
for each edge (u,v) in p

if $(u,v) \in E$

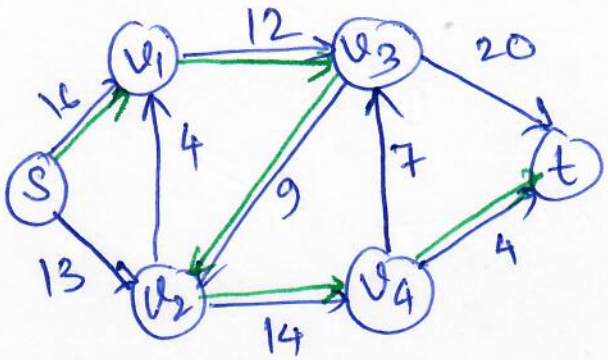
$(u,v).f = (u,v).f + c_f(p)$

else $(v,u).f = (v,u).f - c_f(p)$.

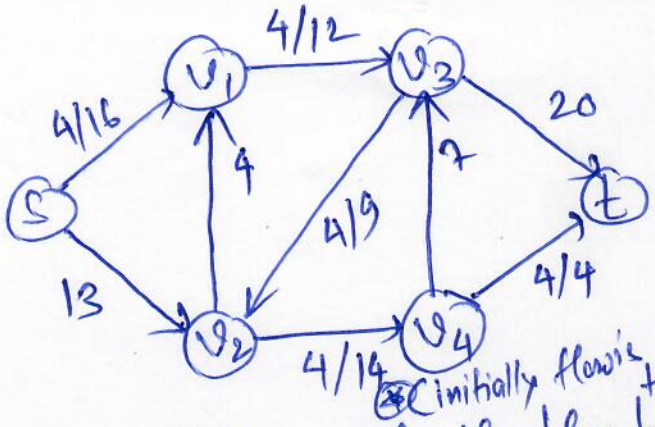
Example of Ford-Fulkerson



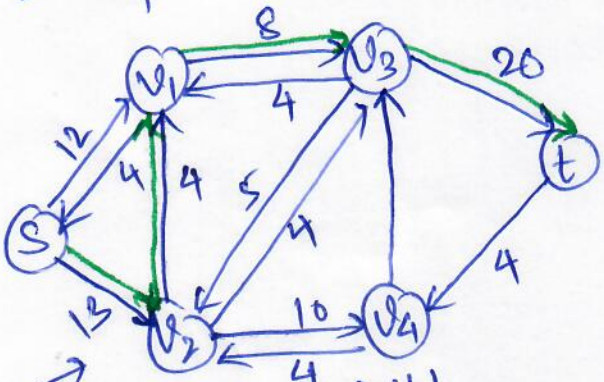
Example: Max-Flow



(a) Augmenting path indicated in green. The algo chooses some path from (s) to (t).

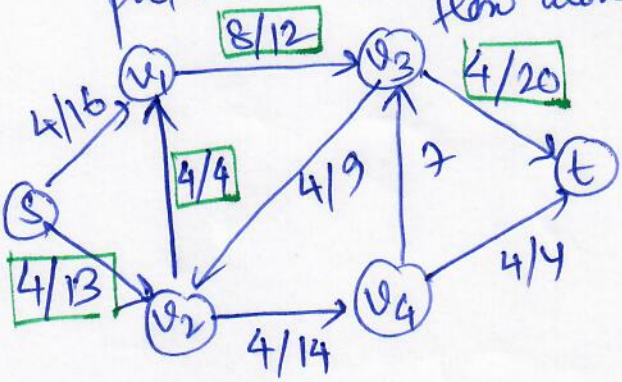


(b) We increased the flow by 4 along the augmenting path in (a). 4 is max. possible flow along this path. (initially flow is assumed to be 0)

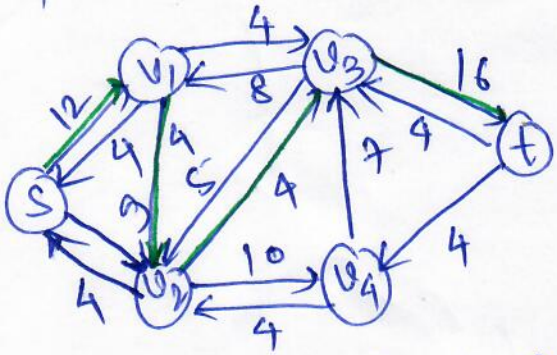


Residue network of (b).

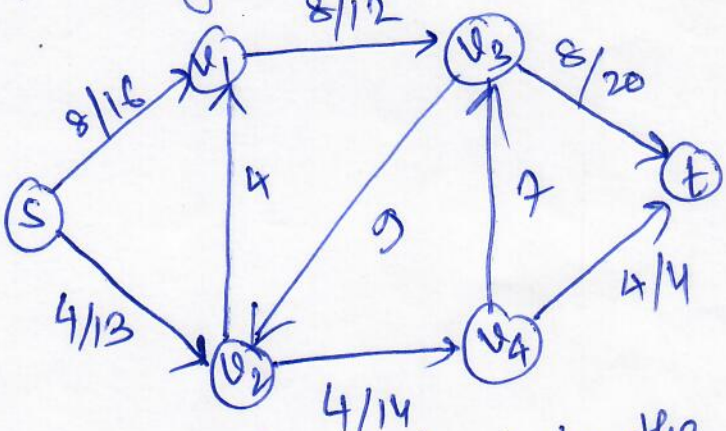
(c) Now look at augmenting path: $\langle s, v_2, v_1, v_3, t \rangle$



(d) We can increase the flow along augm. path by 4.



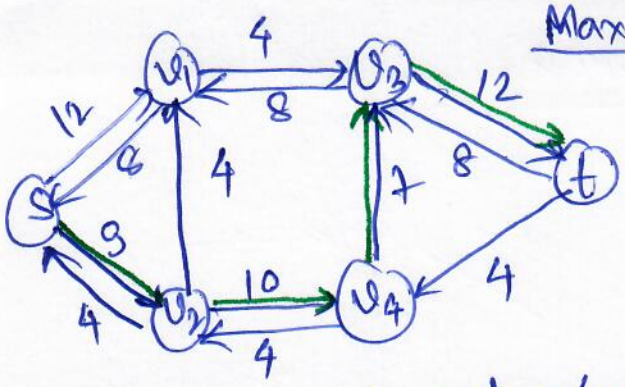
(e) Residue network of (d) and augmenting path



(f) The flow $(v_1, v_2) \cdot f = 4$ in the augmenting path in (e), but the flow previously along $v_2 \to v_1$ i.e., $(v_2, v_1) \cdot f$ in (d) was 4, these two flows in reverse directions cancel out, and we keep the original edge

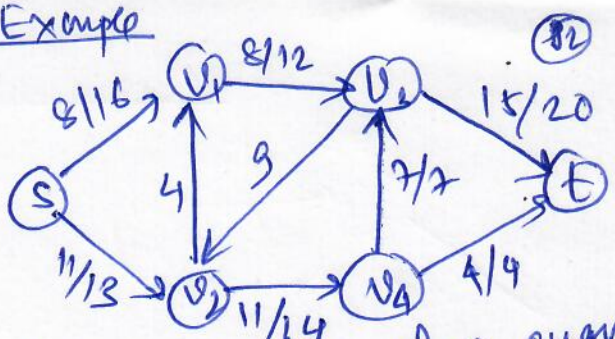
Similarly for edge (v_2, v_3). The flow in augm. path in (e) is $(v_2, v_3) \cdot f = 4$, but originally (v_3, v_2) flow is 4, so they cancel out & original edge remains

Max-flow Example

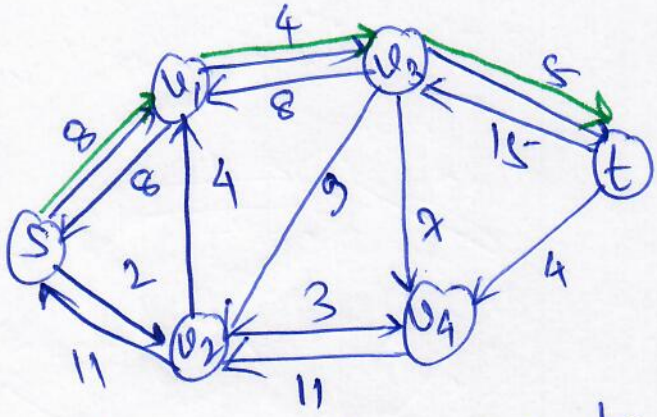


(g): residue network of (f)

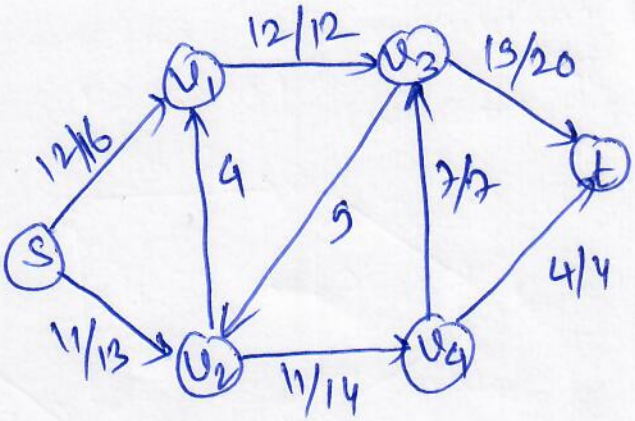
The augmenting path we choose this time is $\langle s, v_2, v_4, v_3, t \rangle$



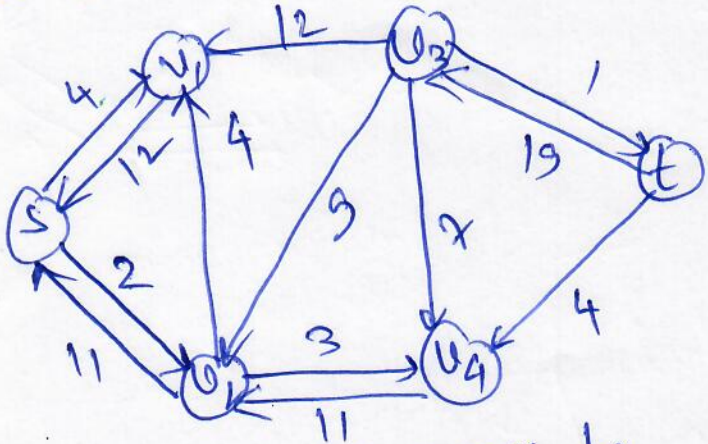
(h): Increasing flow along augm. path in (g).



(i) Residue graph corresp. to (h). There are two paths remaining from vertex (s) to vertex (t). These are: $\langle s, v_1, v_3, t \rangle$ and $\langle s, v_2, v_1, v_3, t \rangle$. We choose the shortest one.



(j) The max flow along the aug. path indicated in (i) is 4. We increase the flow along this path by 4.



(k) Residue graph corres. to (j). Note that now there are no more paths from (s) to (t), hence we stop!

The max flow is computed from (j) as:

$$\sum_{v \in V} (s, v) \cdot f$$

$$= (s, v_1) \cdot f + (s, v_2) \cdot f$$

$$= 12 + 11 = 23.$$

Algorithms
(Selected Topics)

Number Theory

Divisibility: b is divisible by a if there is an integer x s.t. $b = ax$, and write $a|b$.

Th. (1)
1) $a|b \Rightarrow a|bc$ for any integer c
2) $a|b$ & $b|c \Rightarrow a|c$
3) $a|b$ and $a|c \Rightarrow a|bx+cy$ for any integers x & y.
4) $a|b$ and $b|a \Rightarrow a = \pm b$
5) $a|b$, $a>0, b>0 \Rightarrow a \le b$.
6) $m \neq 0$ $a|b \Leftrightarrow ma|mb$.

Division Algo: Given any integers a and b, with $a>0$, there exist unique integers q & r s.t $b = qa+r$, $0 \le r < a$. If $a \nmid b$, then r satisfies the strange ineq. $0 < r < a$.

Euclidean Algorithm Given integers b and $c>0$, we make a repeated application of the division algorithm, above to obtain a series of equation

$$b = cq_1 + r_1 \qquad 0 < r_1 < c,$$
$$c = r_1 q_2 + r_2, \qquad 0 < r_2 < r_1,$$
$$r_1 = r_2 q_3 + r_3, \qquad 0 < r_3 < r_2,$$
$$\vdots$$
$$r_{j-2} = r_{j-1} q_j + r_j \qquad 0 < r_j < r_{j-1}$$
$$r_{j-1} = r_j q_{j+1}.$$

The greatest common divisor (b,c) of b and c is r_j, the last nonzero remainder in the division process. Values of x_0 & y_0 in $(b,c) = bx_0 + cy_0$ can be obtd. by writing each r_i as a linear comb. of b and c.

Example Find the G.C.D. of 42823 and 6409.

Solution: Choose $b = 42823$
$c = 6409$

By division algorithm:

$$\begin{array}{r|l} & 6 \\ 6409 & 42823 \\ & 38454 \\ \hline & 4369 \end{array}$$

Hence: $42823 = 6 \cdot 6409 + 4369$ $\begin{pmatrix} 42823, \\ 6409 \end{pmatrix}$

Dividing the quotient 6409 by the remainder 4369, we have:

$$\begin{array}{r|l} & 1 \\ 4369 & 6409 \\ & 4369 \\ \hline & 2040 \end{array}$$

Continuing the process of dividing the quotient by the remainder.

$$\begin{array}{r|l} & 2 \\ 2040 & 4369 \\ & 4080 \\ \hline & 289 \end{array}$$

Dividing the quotient & 2040 by remainder 289, Putting all together.

$$\begin{array}{r|l} & 7 \\ 289 & 2040 \\ & 2023 \\ \hline & 17 \end{array}$$

Dividing 289 by 17:

$$\begin{array}{r|l} & 17 \\ 17 & 289 \\ & 289 \\ \hline & 0 \end{array}$$

Since, we get remainder 0, the previous non-zero remainder is the G.C.D.

Example Find integers x and y to satisfy

$$42823x + 6409y = 17.$$

From previous G.C.D. computation:

$$42823 = 6 \cdot 6409 + 4369 \quad \rightarrow (1)$$

$$6409 = 1 \cdot 4369 + 2040 \quad \rightarrow (2)$$

$$4369 = 2 \cdot 2040 + 289 \quad \rightarrow (3)$$

$$2040 = 7 \cdot 289 + 17 \quad \rightarrow (4)$$

$$289 = 17 \cdot 17$$

From (4), we have

$$17 = 1 \cdot 2040 - 7 \cdot 289 \longrightarrow (5)$$

From (3) we have:

$$289 = 4369 - 2 \cdot 2040 \longrightarrow (6)$$

Substituting 289 from (6) in (5), we get

$$17 = 1 \cdot 2040 - 7 \cdot (4369 - 2 \cdot 2040)$$

$$= 1 \cdot 2040 - 7 \cdot 4369 + 14 \cdot 2040$$

$$= 15 \cdot 2040 - 7 \cdot 4369 \longrightarrow (7)$$

Substituting 2040 from (2) in (7) above,

$$17 = 15(1 \cdot 6409 - 1 \cdot 4369) - 7 \cdot 4369$$

$$= 15 \times 6409 - 15 \times 4369 - 7 \times 4369$$

$$= 15 \times 6409 - 22 \times 4369 \longrightarrow (8)$$

Substituting 4369 from (8) in (1), we get

$$17 = 15 \times 6409 - 22 \times (1 \times 42823 - 6 \times 6409)$$

$$= 15 \times 6409 - 22 \times 42823 + \underline{22 \times 6 \times 6409}$$

$$= 147 \times 6409 - 22 \times 42823 .$$

Primes

Any integer $p > 1$ is called a prime number or a prime, in case these is no divisor d of p satisfying $1 < d < p$. If an integer $a > 1$ is not a prime, it is called a composite number.

Example 2, 3, 5, and 7 are prime.
4, 6, 8, and 9 are composite.

Th. (2) Every integer n greater than 1 can be expressed as a product of prime

$$n = p_1^{d_1} p_2^{d_2} \cdots p_r^{d_r}$$

* Canonical factoring of n as a product of prime.
* Called fundamental theorem of arithmetic.

Th. (3) If $p \mid ab$, p being a prime, then $p \mid a$ or $p \mid b$. More generally, $p \mid a_1 a_2 \cdots a_n$ then $p \mid a_i$ for atleast on i.

Th. (4) The factoring of any integer $n > 1$ into prime is unique apart from the ordering of the prime factors.

Def^n. If an integers m, not zero, divides the difference $a-b$, we say that a is congruent to b modulo m and write

$$a \equiv b \pmod{m}$$

Th 5. Let a, b, c, d denote integers. Then:

(1) $a \equiv b \pmod{m}$, $b \equiv a \pmod{m}$ and $a - b \equiv 0 \pmod{m}$ are equivalent statements

(2) if $a \equiv b \pmod{m}$ and $b \equiv c \pmod{m}$, then $a \equiv c \pmod{m}$

(3) If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$ then $a + c \equiv b + d \pmod{m}$

(4) If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$ then $ac \equiv bd \pmod{m}$

(5) If $a \equiv b \pmod{m}$ and $d \mid m$, $d > 0$, then $a \equiv b \pmod{d}$

(6) If $a \equiv b \pmod{m}$ then $ac \equiv bc \pmod{mc}$, for $c > 0$.

Example (1): $3 \mid n$ if and only if sum of its digits is divisible by 3.

Example (1) $11 \mid n$ if and only if ~~alternate~~ sum ~~of digits with alternate sign~~ is divisible by 11. Eg: 2728 is divisible by 11 because $11 \mid (2 - 7 + 2 - 8)$.

Number Theory Alg.

Th. 6 Let f denote a polynomial with integral coeff. If $a \equiv b \pmod m$, then

$$f(a) \equiv f(b) \pmod m.$$

Th. 7 $ax \equiv ay \pmod m$ if and only if

$$x \equiv y \left(\mod \frac{m}{(a,m)}\right)$$

Th. 8 If $ax \equiv ay \pmod m$ and $(a,m) = 1$, then $x \equiv y \pmod m$.

Th. 9 $x \equiv y \pmod{m_i}$ for $i = 1, 2, \ldots, r$ if and only if $x \equiv y \left(\mod [m_1, m_2, \ldots, m_r]\right)$.

Q: What is m_i are relatively prime?

Pf ① If $ax \equiv ay \pmod m$ then m divides $ay - ax \Rightarrow ay - ax = mz$ for some integer z.

Dividing by the g.c.d of a & m:

$$\frac{a}{(a,m)}(y-x) = \frac{m}{(a,m)} z.$$

$$\Rightarrow \frac{m}{(a,m)} \Big| \frac{a}{(a,m)}(y-x)$$

Since $\left(\frac{a}{(a,m)}, \frac{m}{(a,m)}\right) = 1$

$\Rightarrow \frac{m}{(a,m)} \mid (y-x)$

$\Rightarrow x \equiv y \pmod{\frac{m}{(a,m)}}$.

Th. If $c \mid ab$ and $(b,c) = 1$, then $c \mid a$.

Converse: try.

(2) Special case of (1).

Th. (8) If $(a,m) = 1$ then there is an x s.t $ax \equiv 1 \pmod{m}$. [Any two such x are congruent $\pmod{m}$. If $(a,m) > 1$ then there is no such x.]

Pf. If $(a,m) = 1$, then there exist x & y s.t. $ax + my = 1$. That is $ax \equiv 1 \pmod{m}$, Conversely, if $ax \equiv 1 \pmod{m}$, then $m \mid ax - 1$

$\Rightarrow ax - 1 = my$ for some y. That is.

$ax + m(-y) = 1$

$ax + my' = 1$

$\Rightarrow (a,m) = 1$.

Thus if $ax_1 \equiv ax_2 \equiv 1 \pmod{m}$ if $(a,m) = 1 \Rightarrow x_1 \equiv x_2 \pmod{m}$

I. The Chinese Remainder Theorem

Solving simultaneous congruences.

$$x \equiv a_1 \pmod{m_1}$$
$$x \equiv a_2 \pmod{m_2} \quad (*)$$
$$\vdots$$
$$x \equiv a_r \pmod{m_r}.$$

II. The Chinese Remainder Theorem. Let $m_1, m_2, \ldots, m_r$ denote r positive integers that are relatively prime in pairs, and let $a_1, a_2, \ldots a_r$ denote any r integers. Then the congruence (*) has common solutions. If x_0 is one such solution, then an integer x satisfies the congruence (*) if and only if x is of the form $x = x_0 + km$ for some integer k. Here $m = m_1 m_2 \cdots m_r$.

Pf. Writing $m = m_1 m_2 \cdots m_r$, we see that m/m_j is an integer and that

$$\left(\frac{m}{m_j}, m_j\right) = 1$$

(because m_j is relatively prime with $m_1, m_2, \ldots m_{j-1}, m_{j+1} \ldots m_r$)

Hence, by theorem (8) for each j there is an integer b_j s.t.

$$\left(\frac{m}{m_j}\right) b_j \equiv 1 \pmod{m_j}.$$

Clearly $\left(\frac{m}{m_j}\right) b_j \equiv 0 \pmod{m_i}$ if $i \neq j$.

Put $$x_0 = \sum_{j=1}^{r} \frac{m}{m_j} b_j a_j.$$

we consider this number module m_i, and find that

$$x_0 \equiv \frac{m}{m_i} b_i a_i \equiv a_i \pmod{m_i}.$$

$\left(\frac{m}{m_i}\right) b_i \equiv 1 \pmod{m_i}$ & $a_i \equiv a_i \pmod{m_i}$

$\Rightarrow$ x_0 is a solution of the system (*).

If x_0 & x_1 are two solutions of the system (*). then. $x_0 \equiv x_1 \pmod{m_i}$ for $i = 1, 2, \ldots r$

$\Rightarrow x_0 \equiv x_1 \pmod{m}$ [By multiplying all congruns. & using.

- $x \equiv y \pmod{m_i}$, $i = 1, 2, \ldots r$ if & only if $x \equiv y \pmod{[m_1, m_2, \ldots m_r]}$

Example of Chinese remainder Theorem

Ex: Find the least positive integer x s.t.

$x \equiv 5 \pmod 7$, $x \equiv 7 \pmod{11}$, $x \equiv 3 \pmod{13}$

Sol^n: We have

$a_1 = 5$, $a_2 = 7$, $a_3 = 3$.

$m_1 = 7$, $m_2 = 11$, $m_3 = 13$.

$m = m_1 m_2 m_3 = 7 \cdot 11 \cdot 13 = 1001$.

(1) We have:

$$\left(\frac{m}{m_1}, m_1\right) = \left(m_2 m_3, m_1\right) = 1$$ (Since m_1, m_2 and m_3 are relat. prime).

Hence, by Euclidean algo, we have

$(-2) \cdot m_2 m_3 + 21 \cdot m_1 = 1$

Here, we take $\boxed{b_1 = -2.}$

(2) Similarly, we consider

$$\left(\frac{m}{m_2}, m_2\right) = \left(m_1 m_3, m_2\right) = 1.$$ By Euclidean algo, we have.

$4 \cdot m_1 m_3 + (-33) \cdot m_2 = 1$

So, we take $\boxed{b_2 = 4.}$

(3) ~~Again considering~~

we have $\left(\frac{m}{m_3}, m_3\right) = \left(m_1 m_2, m_3\right) = 1$

By Euclidean algo, we have:

$(-1) \cdot m_1 m_2 + 6 \cdot m_3 = 1$, so we may take $\boxed{b_3 = -1}$

Then we construct

$$x_0 = \sum_{j=1}^{r} \frac{m}{m_j} b_j a_j$$ (Chinese remainder thm)

(Here $r = 3$)

$$= \frac{m}{m_1} b_1 a_1 + \frac{m}{m_2} b_2 a_2 + \frac{m}{m_3} b_3 a_3$$

$$= 11 \cdot 13 \cdot (-2) \cdot 5 + 7 \cdot 13 \cdot 4 \cdot 7 + 7 \cdot 11 \cdot (-1) \cdot 3 = 887$$

Since this sol^n is unique modulo $m = 7 \cdot 11 \cdot 13 = 1001$.

This is the least positive sol^n.

Note: If $x_0 > 1001$, then take modulo m to get the least number.