Latest Advances in Algorithm Analysis and Design

General Queries

- Any one who has taken the one semester Algorithms course at UG or PG Level
- Anyone who has gone through any book on algorithms from start to end
- Anyone who says I am at level 3 with regard to expertise on algorithms if we have 1, 2 and 3 level (level 3 being the highest)

History & Background

- As old as human civilization
- Five Important persons who have direct relevance to Algorithms
- Al Khwarizmi
- Allan Turing
- Von Neumann
- Donald E Knuth
- John Bentley

Algorithm Basics

- Difference between man and animal
- Abstraction (real life problems to be converted to mathematical problems)
- Using tools (How much effective)

Towards Algorithm definition

Components of an Algorithm

Input

Output

Logic to convert from Input to output

How we solve a General Problem

- Specify Problem
- Specify input
- Output Properties
- Generic relationship between input and output

Computational problems

- Works on values: inputs
- Produces values : outputs

- Only computational problems not others can be solved using computer algorithms
- others like whether God exists, whether you love me or not, whether he fears or not

Computatinal Problems

- Computation procedure
- Sequence of computation steps - finite & unambiguous
- Manipulation of mathematical entities

Formal Definition

- An Algorithm is a correct solution for a problem in finite sequence of steps where each step is unambiguous and which terminates for all possible inputs in a finite amount of time and memory.

What is more important in algorithm

- Is something more important then performance

- Modularity
- Scalability
- Graceful Degradation for size and number of inputs
- Correctness
- Maintainability
- Simple

Algorithm Characteristics

- User friendly
- Extensible
- Programmer Time
- Concurrency
- Distributed
- Upload performance
- Security
- Power Efficiency
- Hardware/OS compliant
- Memory

Some proverbs

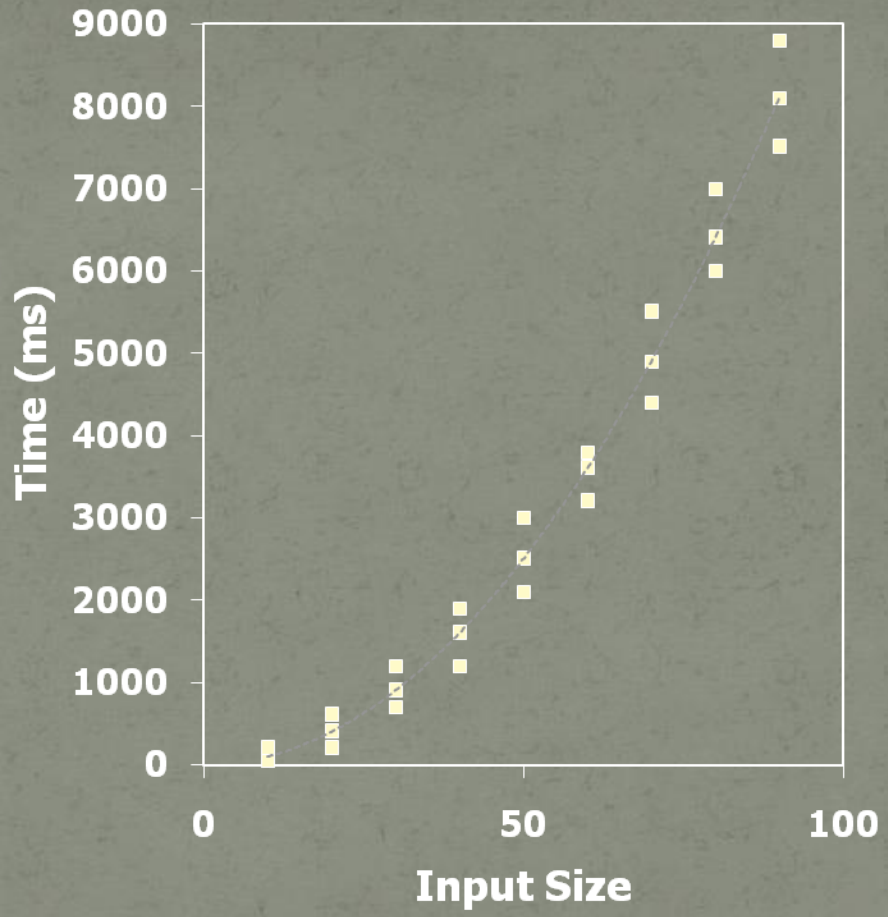
- It is art more then a science
- It is fun
- Fluidity of thinking
- New Idea generation is required for a new algorithm
- Crude solution to refined solution e.g. sitting place
- Performance is the currency of computing

Memory Issue

- complexity of an algorithm is resources required to run that algorithm
- memory now mainly concerned with bandwidth, small gadgets, smart cards. Vista requires 1 GB of RAM.
- Running in RAM or Cache is still a big issue
- Small programs more efficient (Concept of time space trade off does not hold good always)
- advertisement in byte in 1960 unimaginable 32 KB of RAM whooping 10 MB of Hard Disk

Empirical Study

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a function, like the built-in clock() function, to get an accurate measure of the actual running time
- Plot the results


9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Time (ms)
0
50
100
Input Size

Limitations of Empirical study

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Even in same hardware and software environments the results may vary depending upon the processor load, sharing of resources, No. of background processes, Actual status of Primary and Secondary Memory at the time of running the program, Compiler, Network Architecture, programming language

Apriori and Posterior Analysis

- Apriori – Designing then making
- Posterior - Making then waking up after the problem crops up
- Apriori is always better.
- There is corresponding guarantee that any algorithm who is better in performance in its apriori analysis will be better in performance in its posterior analysis

Instruction Count as a measure

- Is not related to type of input
- Different instructions may be very differently loaded in terms of resource requirements
- Can be taken as a first preliminary indication to compare the size of two algorithms but should not cheat you
- Only 10% of the instructions may be actually responsible for the 90% of resource usage

Micro Analysis

- To count each and every operation of the program.
- Detailed
- Takes more time and is complex and tedious

  (average lines of codes are in the range of 3 to 5 million lines)
- Those operations which are not dependent upon the size or number of the input will take constant time and will not participate in the growth of the time or space function, So they need not be part of our analysis

Macro Analysis

- Concerned with selective instructions which are dominant and costliest
- Selection of the right instructions is very important
- Comparisons and Swapping are basic operations in sorting algorithms
- Arithmetic operations are basic operations in math algorithms
- Comparisons are basic operations in searching algorithms
- Multiplication and Addition are basic operations in matrix multiplication

Input Size-two types

- No. of Inputs items
- No. of bits value of Input does not varies (GCD only 2 numbers but will vary on the large and small value of numbers) what increases in the number of bits to represent the information
- No. if input items can be through single parameter - sorting
- Multiple parameters in graphics (vertex & Edges)
- Matrix multiplication pXq and qXr depends on p,q,r three independent matrices. If square matrices then only a single parameter.

Type of input

- Even No. of inputs and bits are same

- Worst Case
- goodness of an algorithm is most often expressed in terms of its worst-case running time.
- Need for a bound on one's pessimism, Every Body needs a guarantee. This is the maximum time an algorithm will take on a given input size
- ease of calculation of worst-case times
- In case of critical systems we can not rely on average or best case times
- Worst Case for all sorting problems is when the inputs are in the reverse order

Average Case

- Very Difficult to compute
- Average-case running times are calculated by first arriving at an understanding of the average nature of the input, and then performing a running-time analysis of the algorithm for this configuration
- Needs assumption of statistical distribution of input
- It is supposed that all inputs are equally likely to occur
- If we have the same worst case time for two algorithms then we can go for average case analysis
- If the average case is also same then we can go for micro analysis or empirical analysis

Worst Case

- Let IC(i) is the instruction count for an input i In the set of all inputs of size n
- Worst case complexity WCC(n) = Max $IC(i_n)$ $I \varepsilon i_n$
- Average Case Complexity will be (IC(i1)+IC(i2)+……IC(in))/n

- 990,991,994,990,999,1000 (A genuine example for average case)
- 1,1,1,1,1,1,1….1000 (A bad example of average case)

Best Case

- Not used in general
- Best case may never occur
- Can be a bogus or cheat algorithm that is otherwise very slow but works well on a particular input
- A particular input is likely to occur more then 90% of the time then we can go for a customized algorithm for that input
- Best Case for all sorting problems is that sequence is already in the sorted sequence

Asymptotic Analysis

- Asymptotic analysis means studying the behavior of the function when n→ infinity or very large
- Problems size will keep on increasing so asymptotic analysis is very important
- Limiting behavior
- We are more concerned with large input size because small size inputs will not very much in running time

Time complexity

- Something should relate instruction count and input size
- Big Oh Notation is the most suitable solution for this

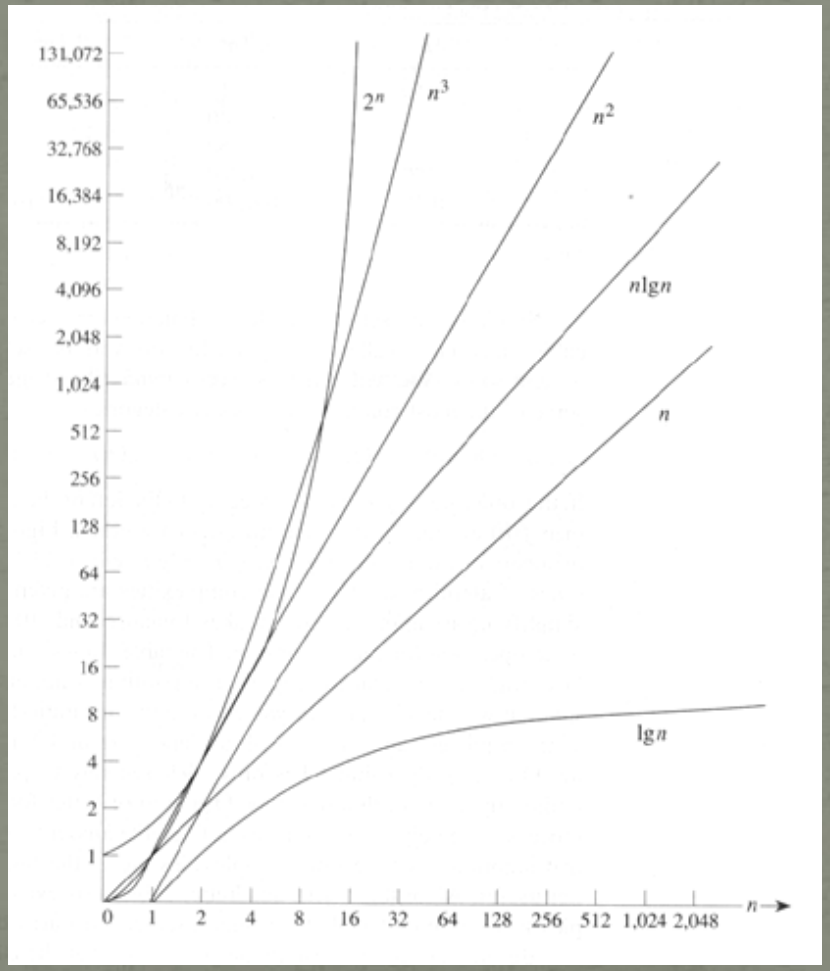
Big Oh Notation

- In most of the algorithms it is difficult to analyze the exact number of operations
- To simplify the analysis:
- Identify the fastest growing term
- Neglect the slow growing terms
- Neglect the constant factor in the fastest growing term

Comparison of growth of common functions

n	logn	nlogn	n^2	n^3	2^n	n!
1	0	0	1	1	2	1
2	1	2	4	8	4	2
4	2	8	16	64	16	24
8	3	24	64	512	256	40320
16	4	64	256	4096	65536	20764283904000
32	5	160	1024	32768	4294967296	~infinity
64	6	384	4096	262144	~infinity	~infinity

Growth of functions



Max size of a problem that can be solved in a given time

Running time	1 sec	1min	1hour	With a 256 times faster processor in an hour
400n	2500	150000	9,000,000	2304000000
20nlogn	4096	166,666	7,826,087	2003478272
$2n^2$	707	5477	42,436	678976
N^4	31	88	244	1952
2^n	19	25	31	39

We get only 2 times faster processor after 18 months

F(n)=n	lg(n)	n	nlgn	n^2	n^3	2^n
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	8 μs	1 ms
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	27 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	64 μs	18.3 min
50	0.006 μs	0.04 μs	0.282 μs	2.5 μs	125 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10 μs	1 ms	4 X 10^{13} yrs
10^3	0.010 μs	1.0 μs	9.966 μs	1 ms	1 s	infinity
10^4	0.013 μs	10 μs	130 μs	100 ms	16.7 min	Infinity
10^5	0.017 μs	0.10 ms	1.67 ms	10 s	11.6 days	Infinity
10^6	0.020 μs	1 ms	19.93 ms	16.7 min	31.7 years	Infinity
10^7	0.023 μs	0.01 s	0.23 s	1.16 days	31,709 yrs	Infinity
10^8	0.027 μs	0.1 s	2.66 s	115.7 days	3.17 X 10^7 yrs	Infinity
10^9	0.030 μs	1 sec	29.90 s	31.7 years	infinity	infinity

Uselessness of lower order terms and constant factors in the growth rate

n	n^2	$0.1n^2 + n + 100$	$n^2 + 2n + 5$
10	100	120	125
20	400	160	445
50	2500	400	2605
100	10000	12000	10205
1000	1000000	101100	1000105
10000	100000000	100100100	100001005
100000	10000000000	10000100100	10000010005

500n and $n^2/10$ will meet at 5000 (threshold)
Unless the threshold is very high we take a lower growth

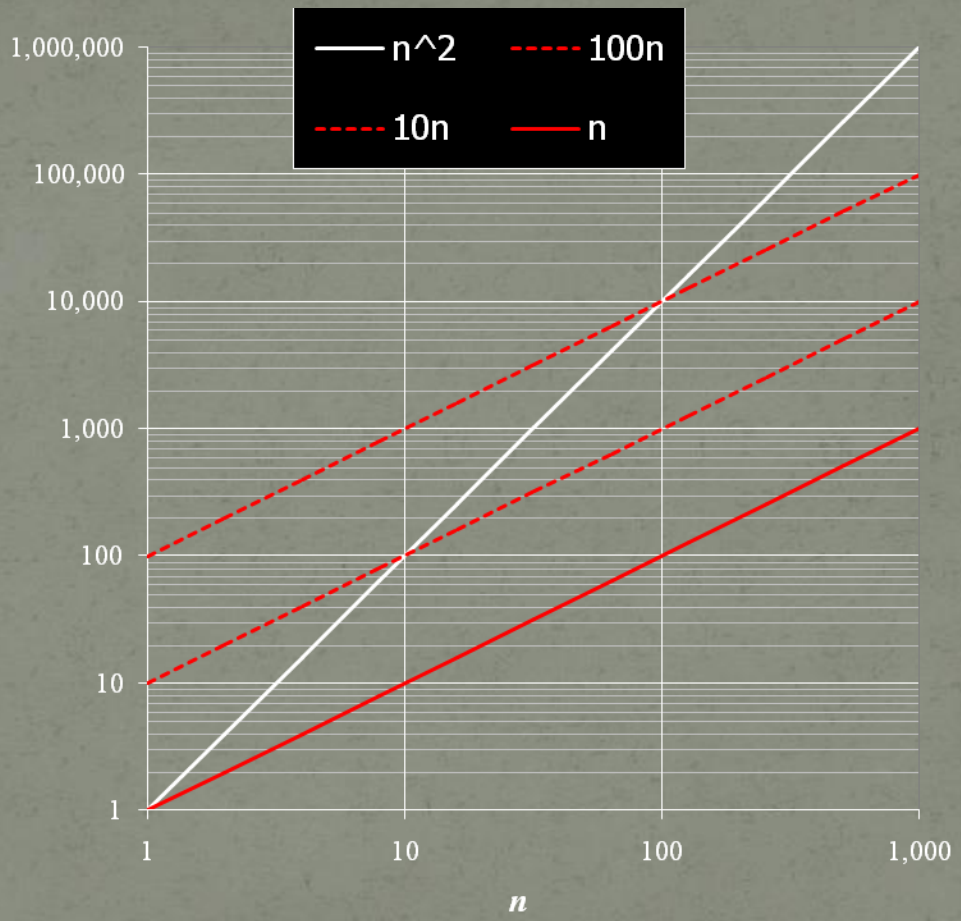
Big oh notation

- resultant of such a simplification is called the algorithm's time complexity.
- It focuses on the growth rate of the algorithm with respect to the problem size
- it makes sense (both mathematically and logically) to ignore the slow growing terms and also the co-efficient of the fastest growing term.

Big Oh Notation

- g(n) = O(f(n)) if there are positive constants c and n0 such that g(n) <= cf(n) for all n >= n_0 This notation is known as Big-Oh notation
- f(n) = $2n^2+7n+9$
- g(n)= $20n^2+6n+42$ belongs to O(f(n))
- c=10 n_o=1

- g(n) = n^3 f(n) = $2n^2+7n+9$
- $n^3/2n^2+7n+9$ → infinity as n-> infinity so you cannot find any c
- so n^3 does not belong to O(f(n)) because order of growth is more then f(n)


n^2
100n
10n
n
1,000,000
100,000
10,000
1,000
100
10
1
1
10
100
1,000
n

Big Oh Notation

- how to find the constant for the sake of argument
- $a_0n^0+a_1n^1+a_2n^2+\ldots\ldots\ldots a_kn^K$ belongs to $O(n^k)$
- $2n^2+7n+9$ is big oh of n^2
- So ignore the lower order terms and ignore the constants associated with the higher order terms
- g(n) is big oh of f(n)
- If g(n)/f(n) ->c(finite) for n → infinity
- c can be zero also
- cf(n) is the upper bound for $n>n_0$ and growth rate is equal or more

Theta Notation

- f(n) = Θ(g(n)) if there are positive constants c_1, c_2 and n_0 such that $c_1 f(n) <= g(n) <= c_2 f(n)$, for all $n >= n_0$. This notation is known as Big-Theta notation
- $20n^2+17n+9$ belongs to $\Theta(n^2)$
- $8n+2$ does not belong to $\Theta(n^2)$
- n^3 does not belong to the $\Theta(n^2)$
- If g(n)/f(n) ->c(finite) for n tending to infinity

c can not be zero

also

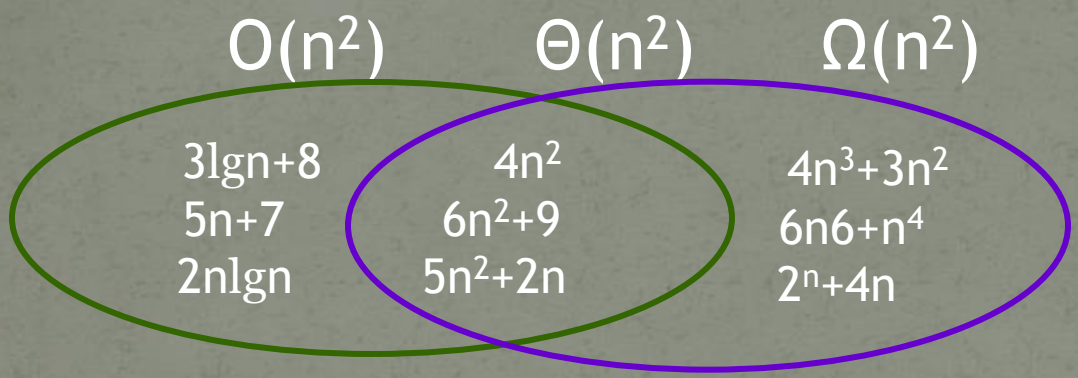
- If f(n)/g(n) ->c(finite) for n tending to infinity

  c can not be zero

Omega Notation

- f(n) = Ω(g(n)) if there are positive constants c and n0 such that f(n) >= cg(n) for all n >= n_0. This notation is known as Big-Omega notation
- The Big-Omega notation can be considered as a lower bound for the f(n) which is the actual running time of an algorithm.
- Informally Ω(g(n)) denotes the set of all functions with a larger or same order of growth as g(n). For example, n^2 belongs Ω(n)

Venn Diagram

-
- 

Lower Bounds

- consider the set of problems to find the maximum of an ordered set of n integers. Clearly every integer must be examined at least once. So Ω(n) is a lower bound for that. For matrix multiplication we have $2n^2$ inputs so the lower bound can be $\Omega(n^{2)}$
- For all sorting & searching we use comparison trees for finding the lower bound.
- For an unordered set the searching algorithm will take Ω(n) as the lower bound. For an ordered set it will take

Ω(logn) as the lower bound. Similarly all the sorting algorithms can not sort in less then Ω(nlogn) time so Ω(nlogn) is the lower bound for sorting algorithms

Amortized Analysis

- An amortized analysis is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.
- Even though we're taking averages, probability is not involved
- •An amortized analysis guarantees the average performance of each operation in the worst case

Aggregate function

- The sum of the amortized complexities of all operations in any sequence of operations be greater than or equal to their sum of the actual complexities

- $\sum_{1 \le i \le n} \text{amortized}(i) \quad \ge \quad \sum_{1 \le i \le n} \text{actual}(i)$

Accounting Method

- Charge ith operation a fictitious amortized cost ci, where we pay 10 Rs. for 1 unit of work (i.e., time).
- •This fee is consumed to perform the operation.
- •Any amount not immediately consumed is stored in the bank for use by subsequent operations.
- •The bank balance must not go negative!
- Thus, the total amortized costs provide an upper bound on the total true costs

Potential Method

- Idea: View the bank account as the potential energy of the dynamic set.
- Framework:

  Start with an initial data structure D0.
- Operation i transforms Di–1to Di.

  The cost of operation i is ci.
- Define a potential functionΦ: {Di} →R,

  such that Φ(D0) = 0 and Φ(Di) ≥0for all i.

  The amortized cost ĉi with respect to Φ is defined to be ĉi= ci+ Φ(Di) –Φ(Di–1).

Potential Method

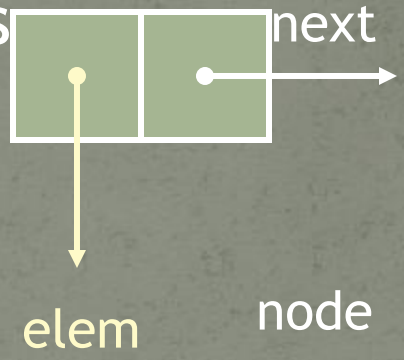
- Amortized costs can provide a clean abstraction of data-structure performance.
- •Any of the analysis methods can be used when an amortized analysis is called for, but each method has some situations where it is arguably the simplest or most precise.
- •Different schemes may work for assigning amortized costs in the accounting method, or potentials in the potential method, sometimes yielding radically different bounds.

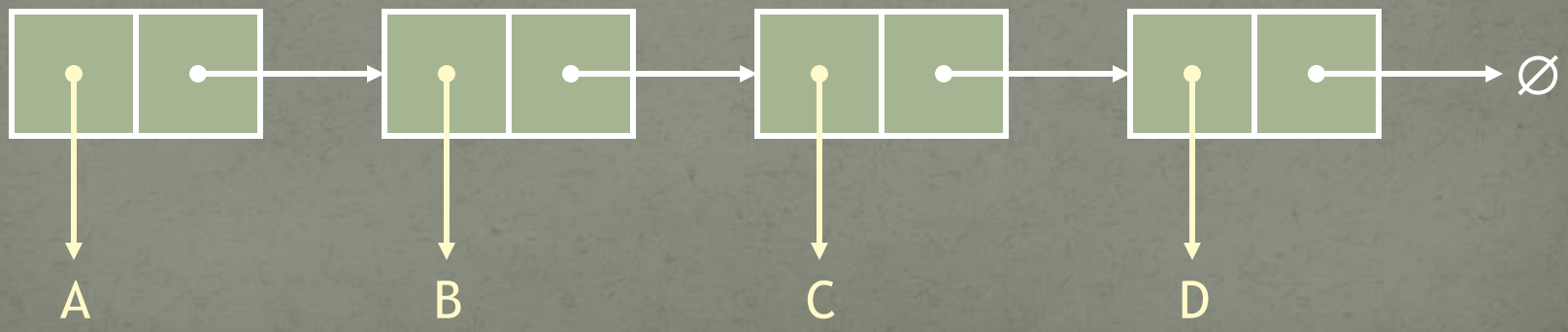
Primitive Data Structures

- Variable – int , char, float, boolean, double, long, short, byte
- comparison of Arrays and Link Lists in terms of access, space , reliability and simplicity
- Single, Double and circular Link List Comparison

Singly Linked List

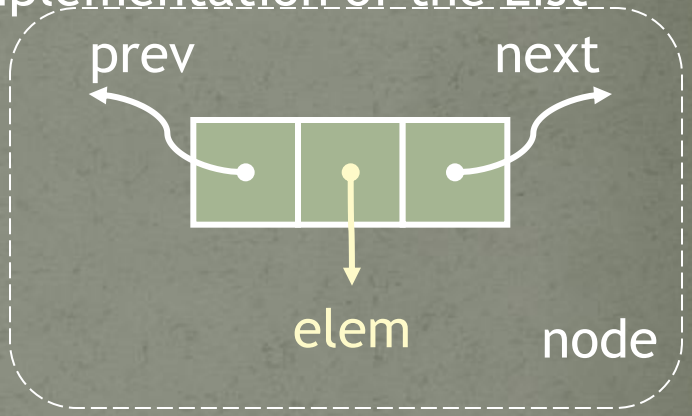
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node

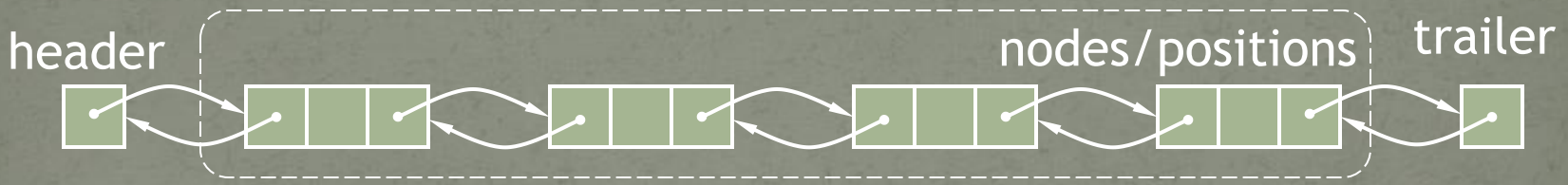




Doubly Linked List

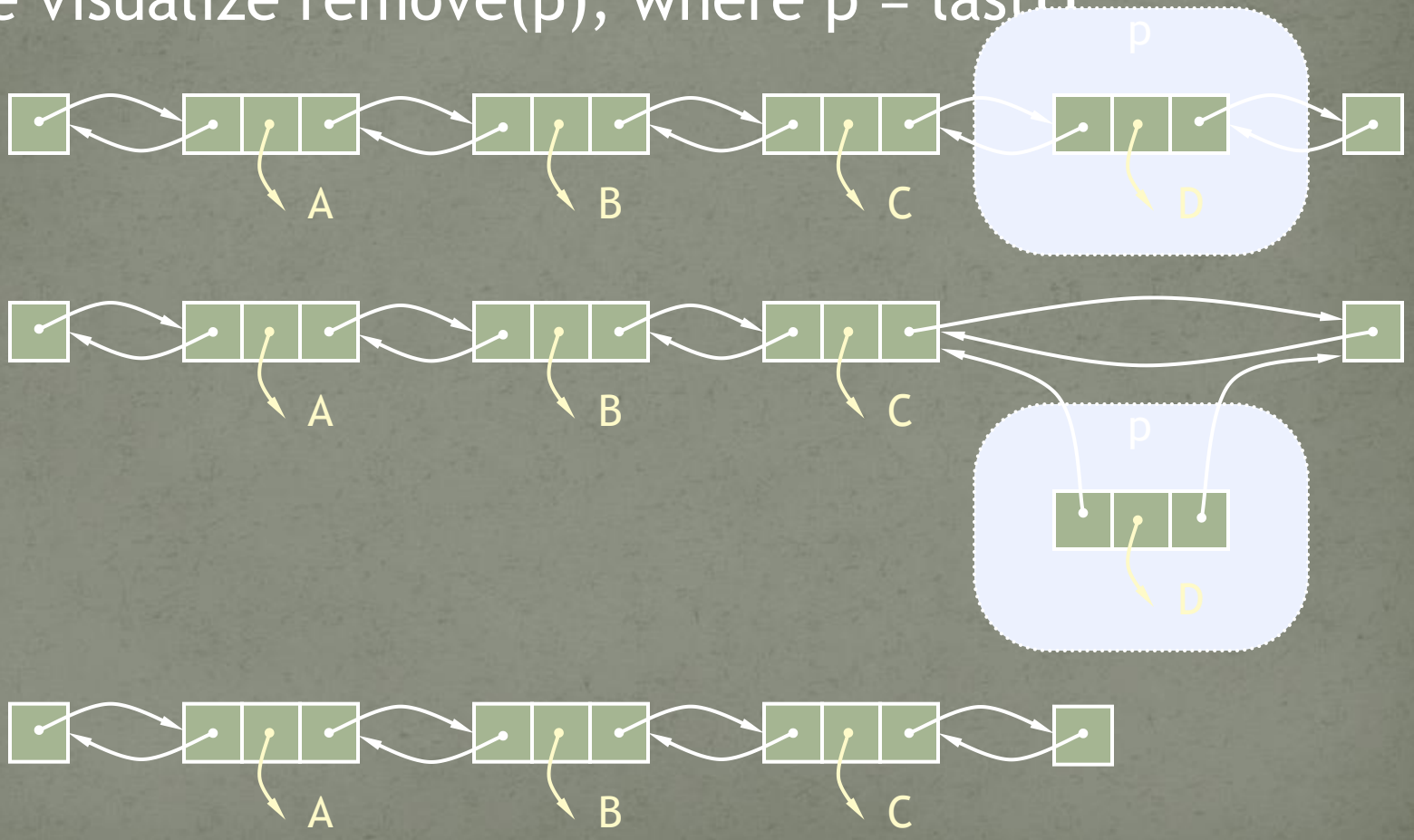
- A doubly linked list provides a natural implementation of the List
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





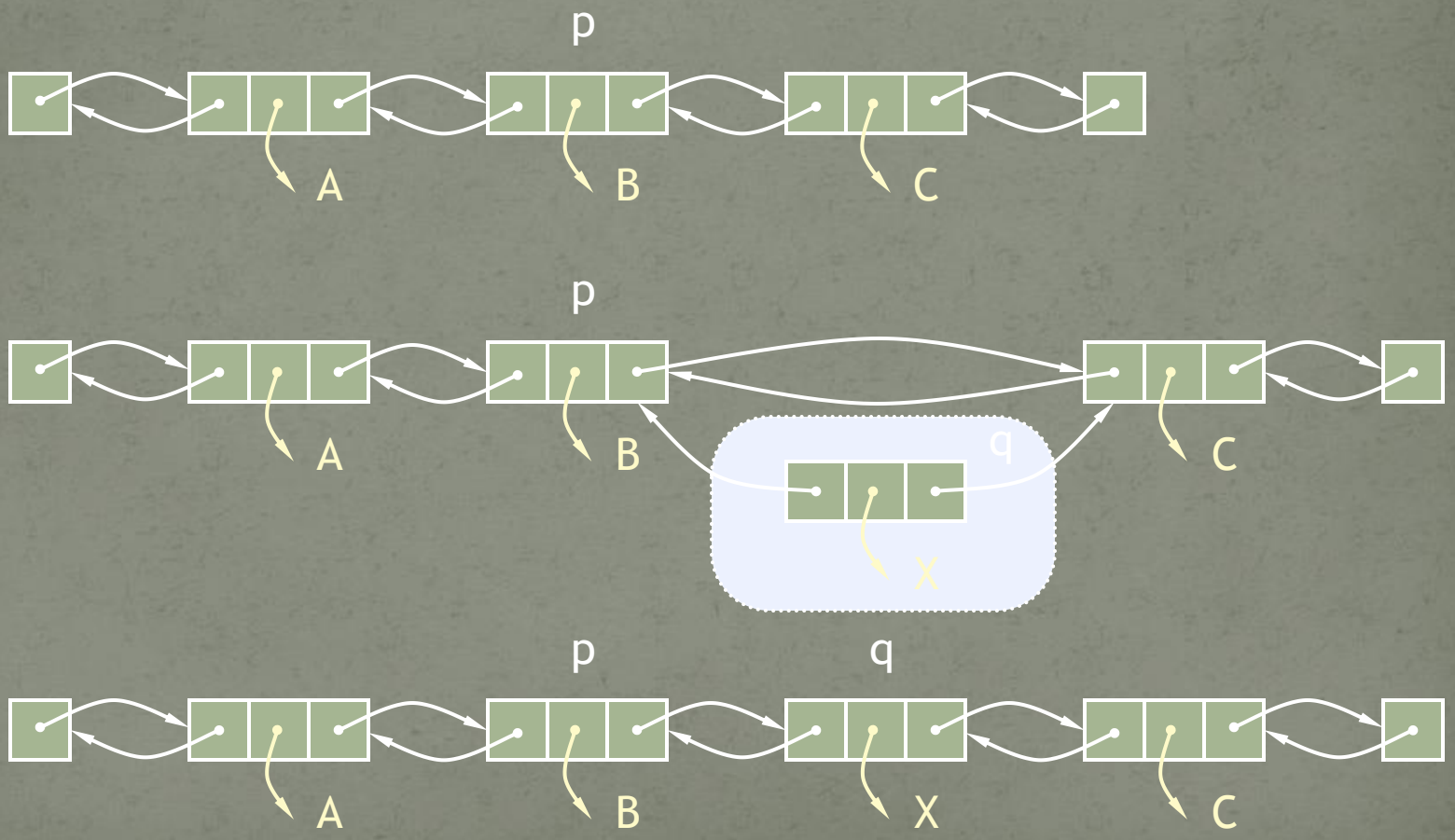
Deletion

- We visualize remove(p), where p = last()



Insertion

- We visualize operation insertAfter(p, X), which returns position q



Multi info and multi link list

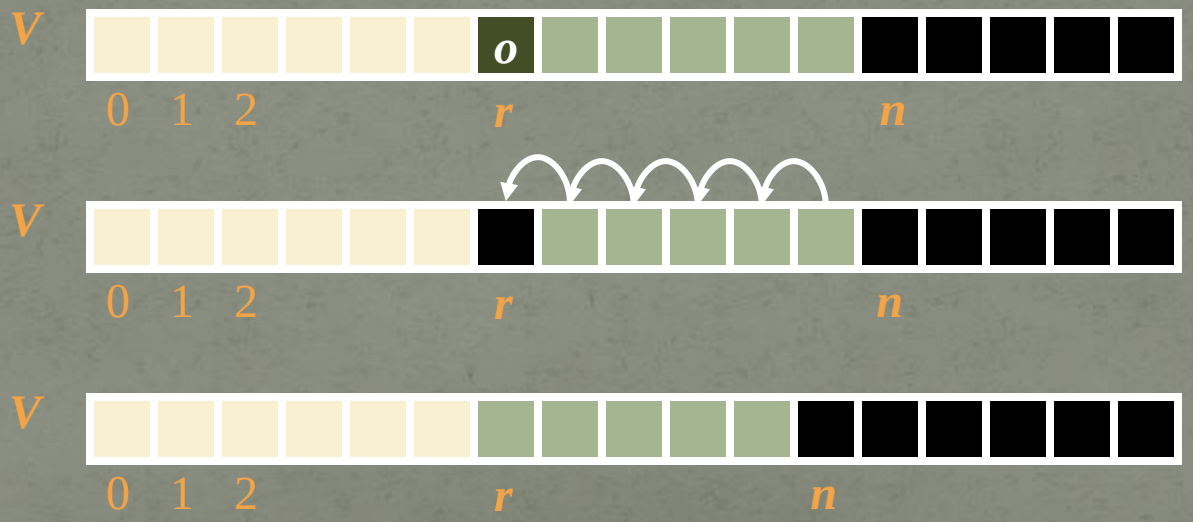
- we should be open to the idea and concept of linked records, tables, trees and multiple links to various other nodes as in the general trees.
- dynamic growth of memory
- garbage collection

Array

- Arrays are stored in contiguous memory locations and contain similar data
- An element can be accessed, inserted or removed by specifying its position (number of elements preceding it)

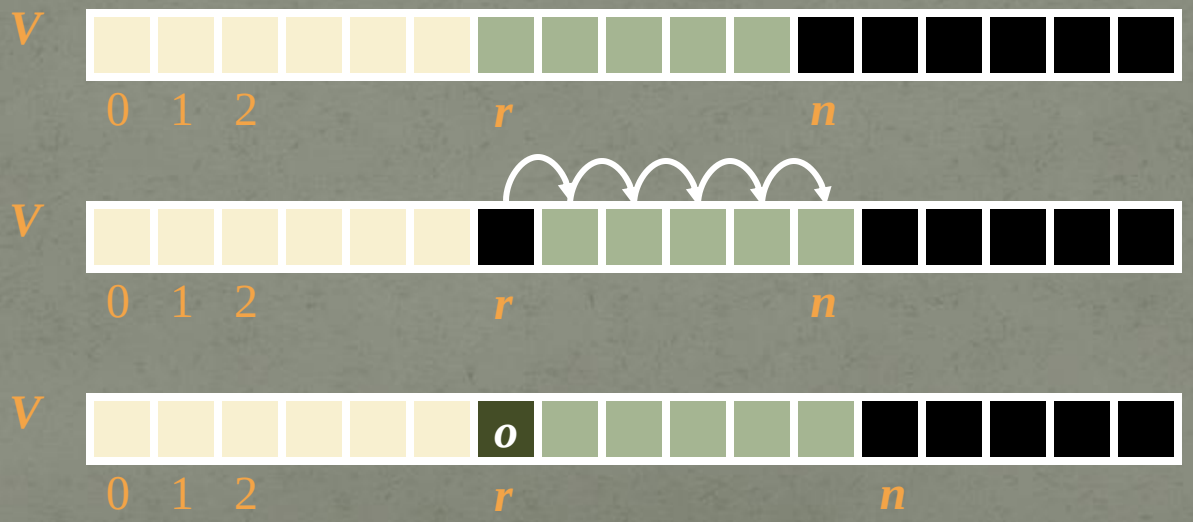
Deletion

- In operation *removeAtPosition*(r), we need to fill the hole left by the removed element by shifting backward the $n - r - 1$ elements $V[r + 1], \ldots, V[n - 1]$
- In the worst case ($r = 0$), this takes $O(n)$ time



Insertion

- In operation $insertAtposition(r, o)$, we need to make room for the new element by shifting forward the $n - r$ elements $V[r], \ldots, V[n-1]$
- In the worst case ($r = 0$), this takes $O(n)$ time



Differences between array and link list

- 1. It is easier to delete in the link list
- 2. it is easier to insert in the link list
- 3. It is easier to access in the array
- 4. Due to Address part there is wastage of memory in link list
- 5. We have to predefine the array so there can be wastage of memory
- 6. Array Size has to be defined and is limited to the defined size while link list can grow to the limit of the memory

Differences between array and link list

- 7. Continuous allocation is required for array, while this is not the case with link list
- 8. Arrays can be accessed backward and forward but we have to use special link lists like doubly linked list for backward access
- 9. Arrays definition is part of the language construct but link list we have to create
- 10. Merging two arrays is very difficult while merging link lists is easy

Stack

- Last in First out
- In and Out only from one end
- One end is closed
- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Function Calls
- Recursion
- In other data Structures

Stack Operations

- push(object): inserts an element
- object pop(): removes and returns the last inserted element
- object top(): returns the last inserted element without removing it
- integer size(): returns the number of elements stored
- boolean isEmpty(): indicates whether no elements are stored
- Boolean isFull(): indicates whether array limit is over

Performance & Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Growable Stack

- In a push operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- How large should the new array be?
  - incremental strategy: increase the size by a constant c
  - doubling strategy: double the size

Queue

- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Two variables keep track of the front and rear

Queue Operations

- operations:
  - enqueue(Object o): inserts an element o at the end of the queue
  - dequeue(): removes and returns the element at the front of the queue
  - front(): returns the element at the front without removing it
  - size(): returns the number of elements stored
  - isEmpty(): returns a Boolean indicating whether no elements are stored

Dequeues

- In these the deletions and insertions can be done at both ends. These are the most general implementations of linear data structures.

Applications

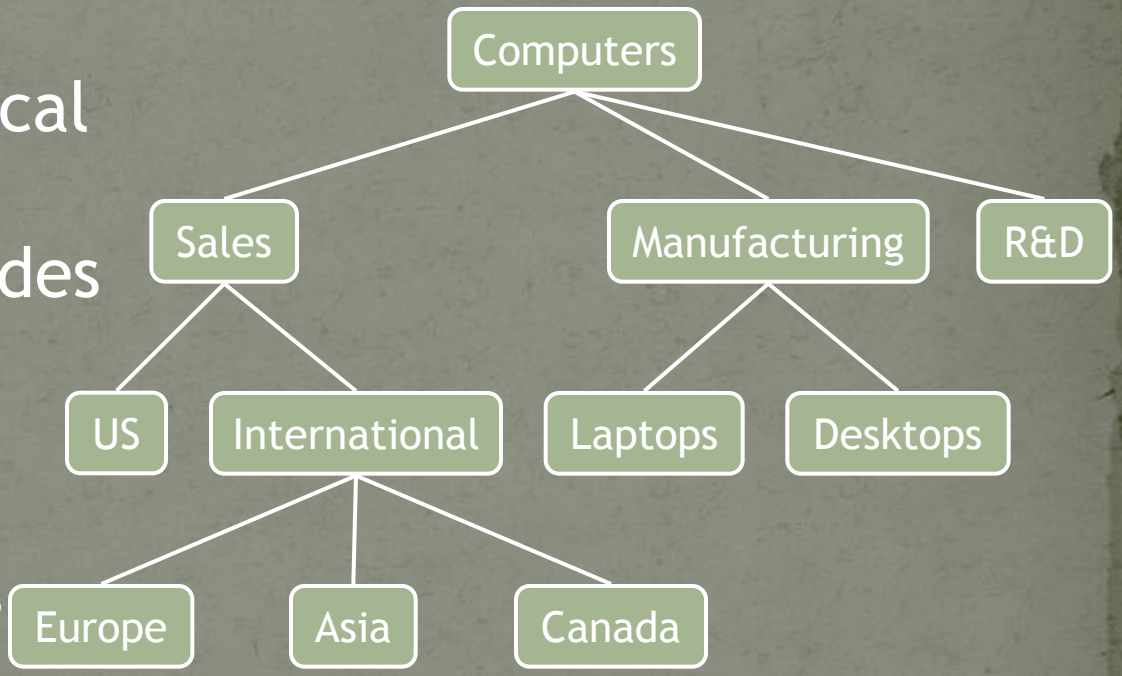
- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
  - Tunnel
  - In other Data Structures

Growability

- In an enqueue operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- Similar to what we did for an array-based stack

Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments

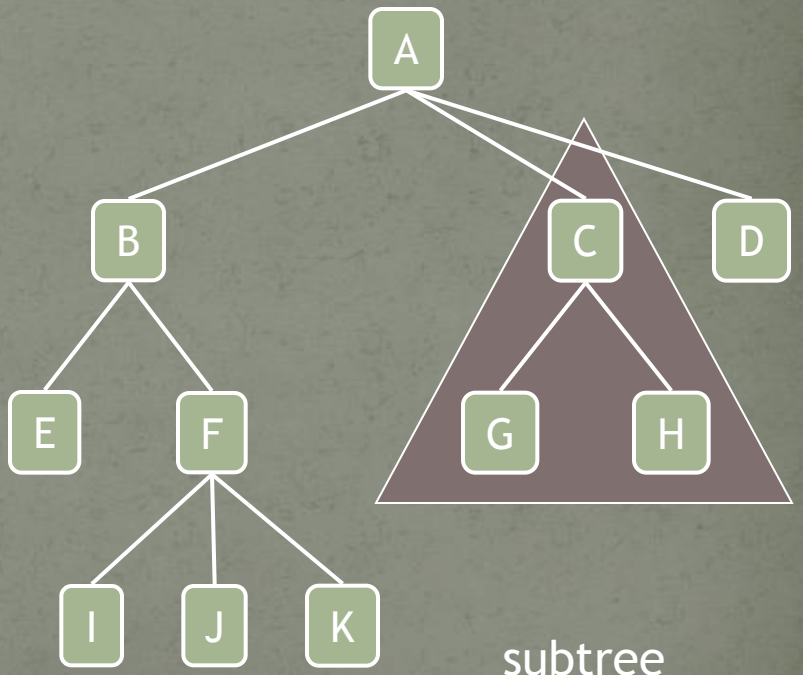


Basics

- There are four things associated with any tree
- Distinction between nodes
- Value of nodes
- orientation
- structure
- weight/value

Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (or leaf node): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild etc.
- Subtree: tree consisting of a node and its descendants



Tree Operations

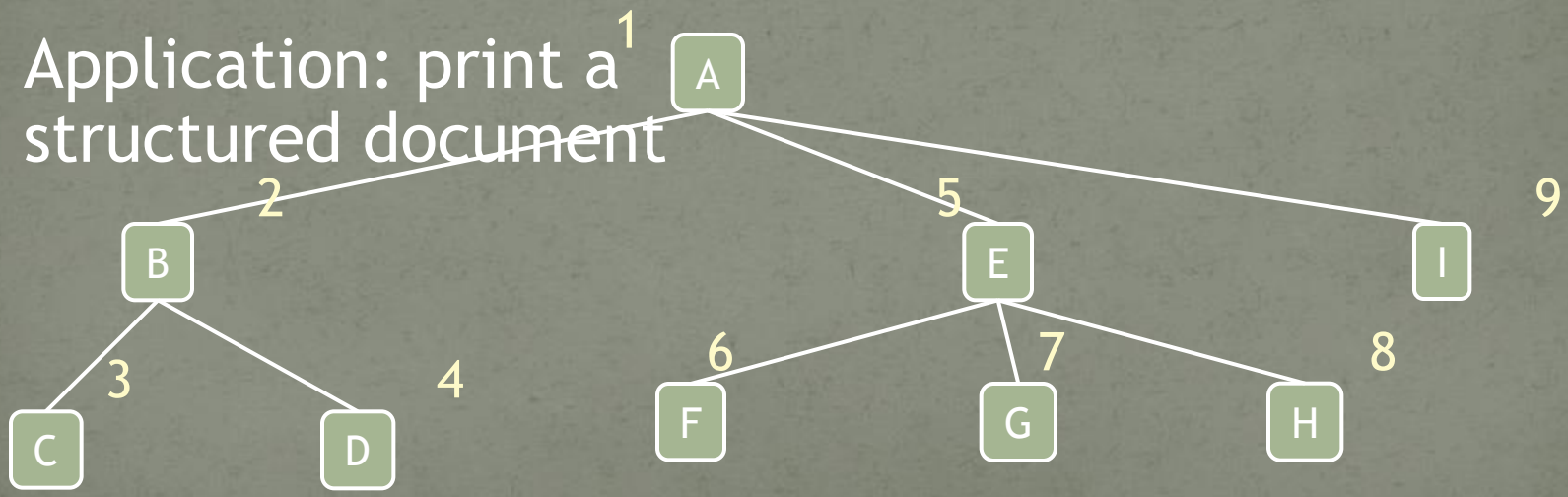
- integer size()
- boolean isEmpty()
- position root()
- position parent(p)
- positionIterator children(p)
- boolean isInternal(p)
- boolean isExternal(p)
- boolean isRoot(p)
- swapElements(p, q)
- object replaceElement(p, o)

Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- In a postorder traversal, a node is visited after its descendants

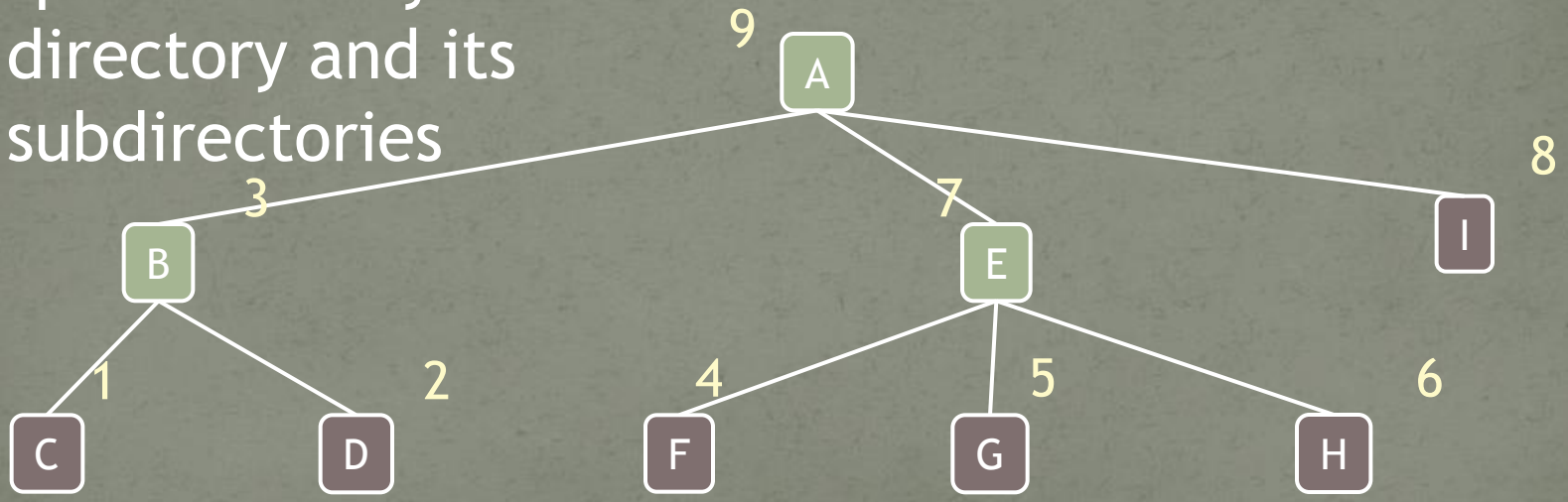
Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document



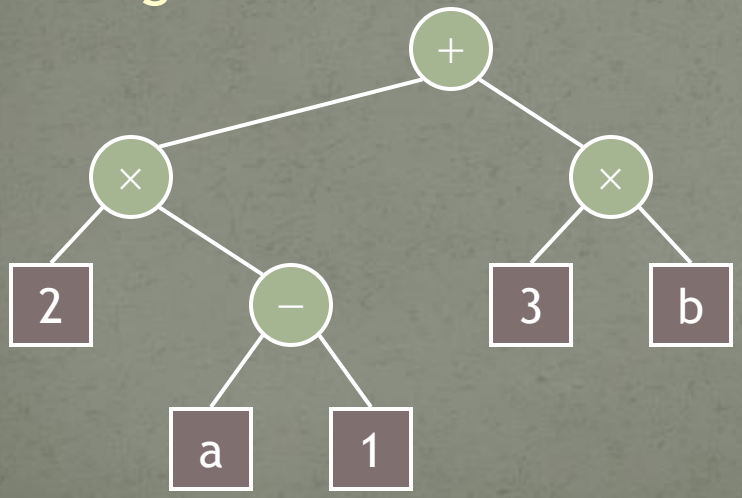
Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories



Inorder Traversal

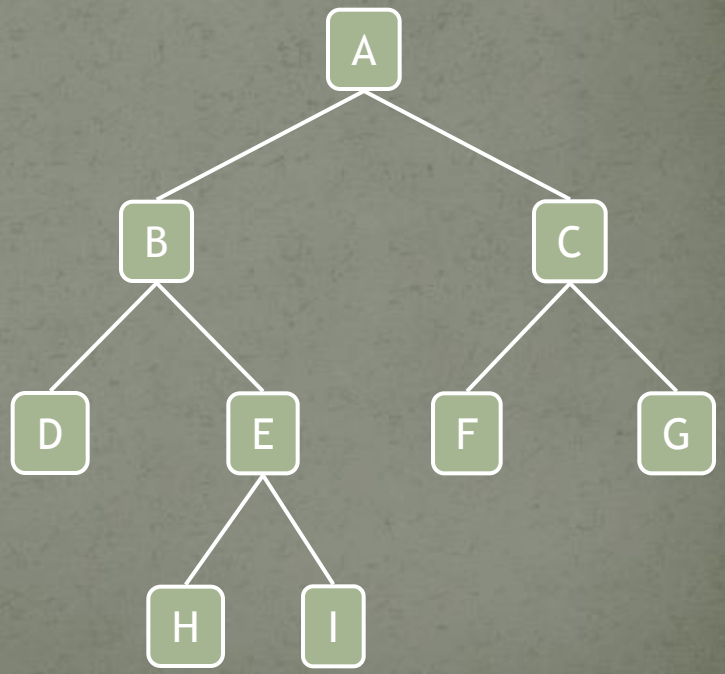
- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree



((2 × (a − 1)) + (3 × b))

Binary Tree

- is a tree with the following properties:
  - Each internal node has at most two children
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or a tree whose root has an ordered pair of children, each of which is a binary tree

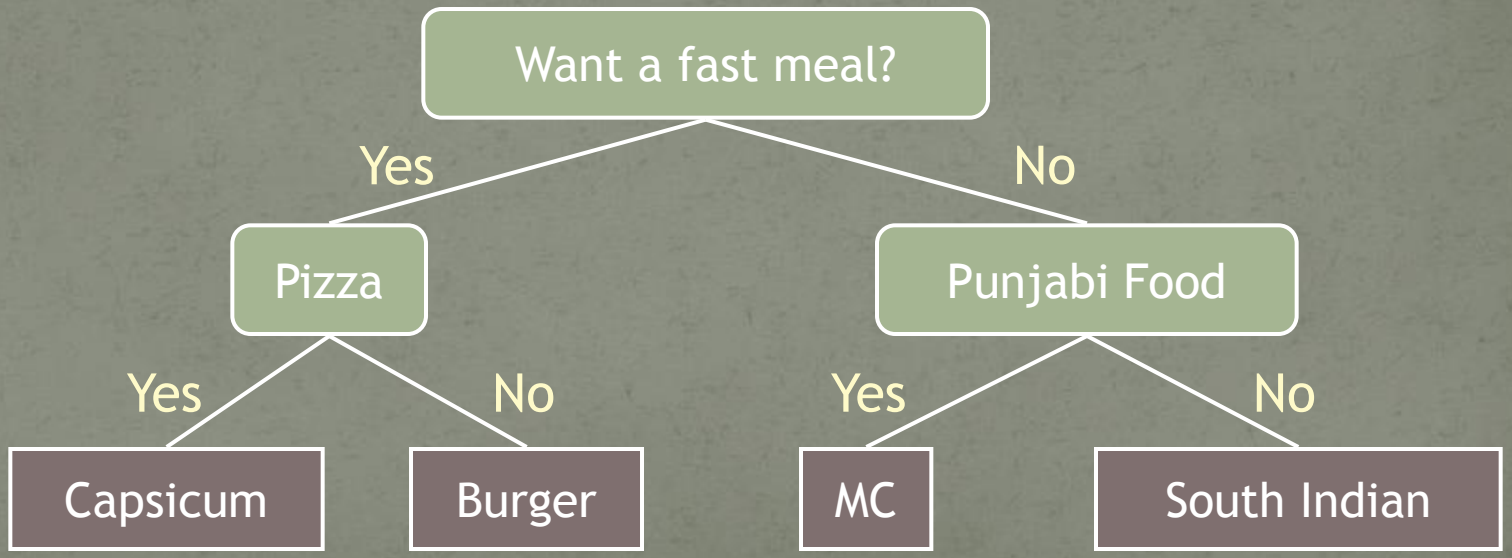


Applications

- arithmetic expressions
- decision processes
- searching

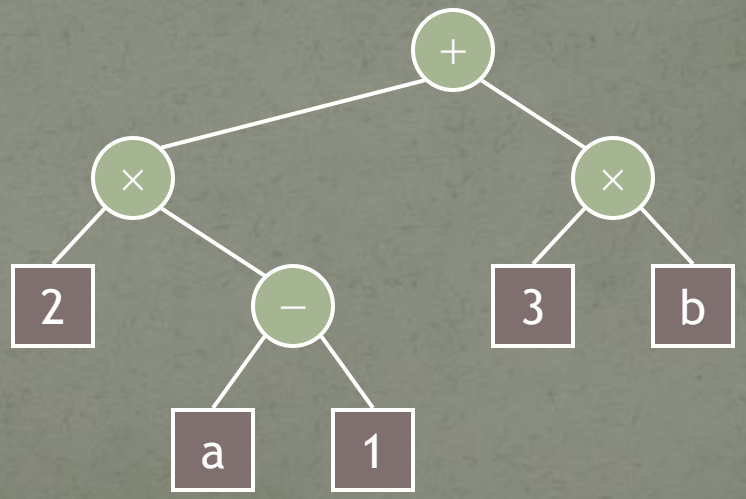
Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$

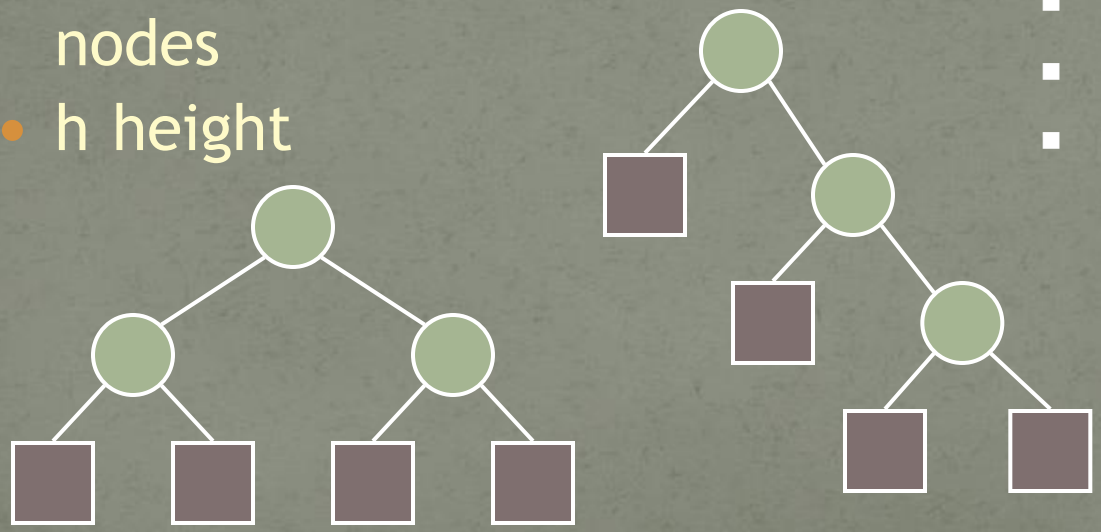


Additional Operations

- position leftChild(p)
- position rightChild(p)
- position sibling(p)

Properties of Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height

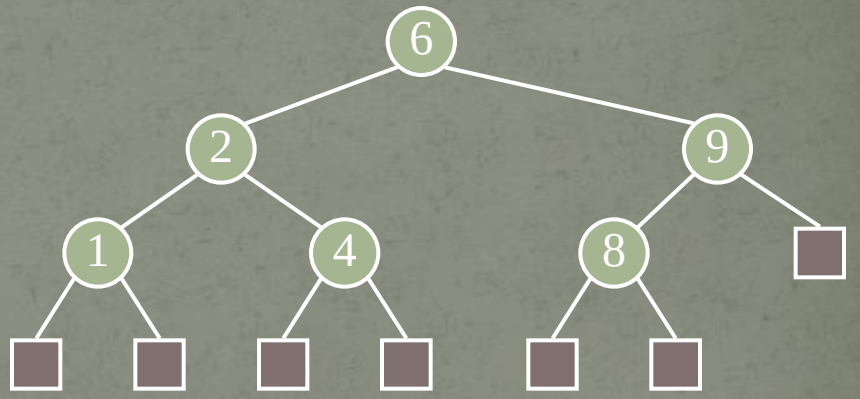


- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

Binary Search Tree

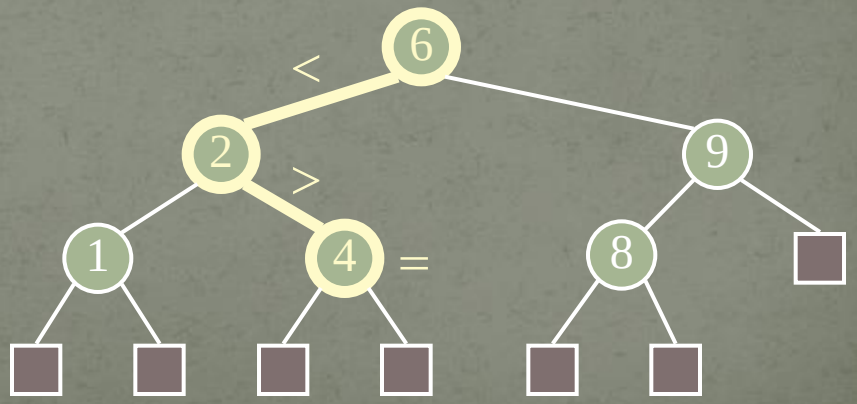
- A binary search tree is a binary tree storing keys (or key-element pairs) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items

- An inorder traversal of a binary search trees visits the keys in increasing order



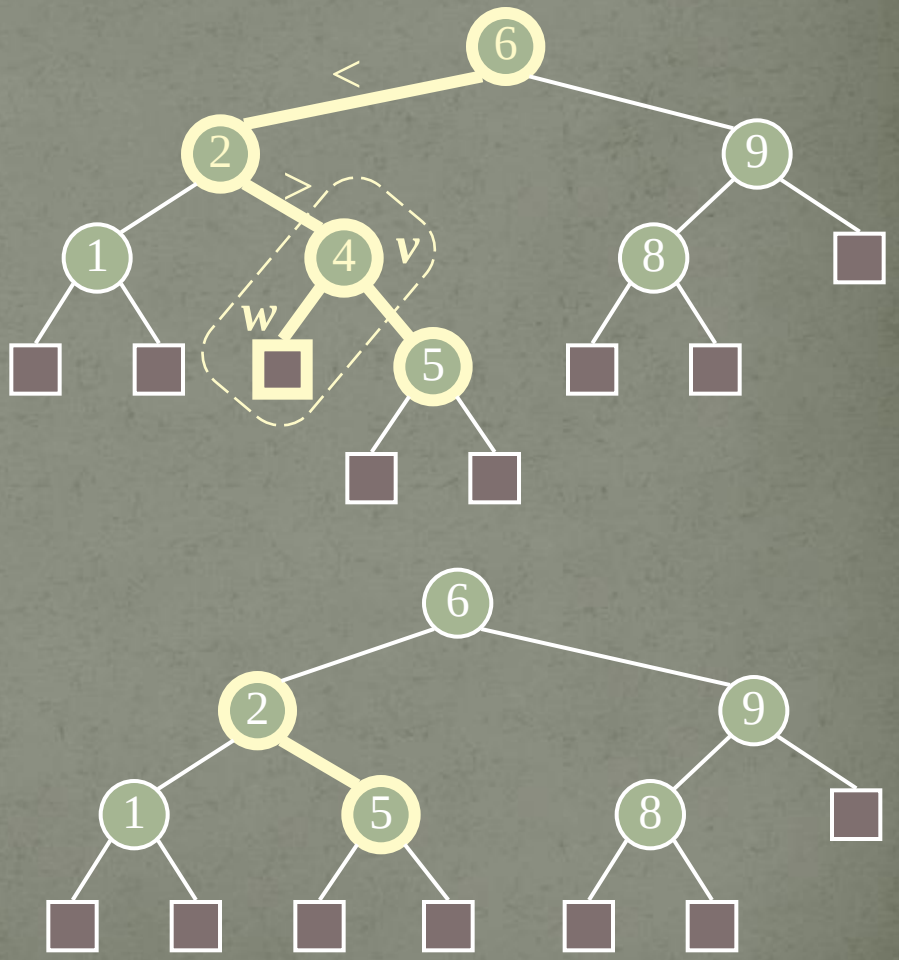
Search

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return a null position
- Example: find(4)



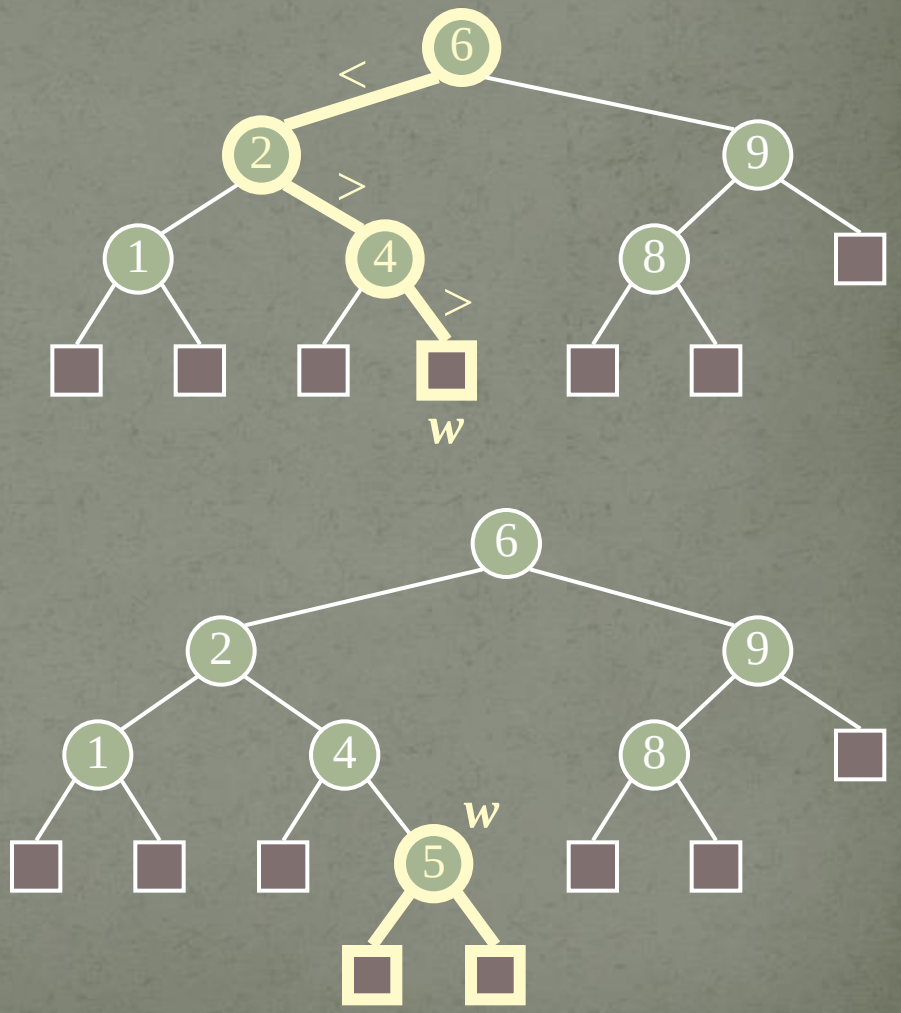
Deletion

- To perform operation removeElement(k), we search for key k
- Assume key k is in the tree, and let let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation removeAboveExternal(w)
- Example: remove 4



Insertion

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5



Code tuning techniques

- Used to write better code
- Needs better understanding of the programming language and its compiler
- Removes un-necessary portion of code
- Is equivalent of code optimization at Higher Language Level

Jamming of loops

- Looping constructs in a program are executed many times. There are quite a few of code tuning techniques which can be applied to looping constructs in order to improve the performance of the code.
- combining loops that operate over same range of values

```
for (k = 1 to n) { initialize T[k] };
for (m = 1 to n) { Max[m] = m * Max[m] };
```

Before Jamming, 2n Loop Checks

```
for (k = 1 to n) {
initialize T[k];    Max[k] = k * Max[k] };
```

After Jamming, n Loop Checks

Unswitching of Loops

- Unswitch loops that contain if tests, if the results of these tests do not change inside the loop

```
for (I=0; i<noofemployees; I++)
{if budgetpassed = YES
.........
else
.........
}
```

Code tuning techniques

- Switching basically refers to making a decision (using a selectional construct) inside a loop every time the loop is executed. If the result of the selectional statement does not change inside the loop then it makes more sense to unswitch the loop by making the decision outside the loop. In such cases the loop is turned inside out by putting the loop inside the selectional construct. The basic idea in unswitching is to minimize / remove unnecessary computation inside a looping construct.

Unrolling of Loops

```
i = 1;
 while(i < num) {
                a( i ) = i;
                i = i + 1; }
```

Before unrolling, n loop checks

```
i = 1;
 while(i < num) {
                a( i ) = i;
                a( i + 1 ) = i + 1;
                i = i + 2; }
```

After unrolling, n/2 loop checks

Minimize work inside loops

For (i = 1 to n/2)

{…}

Need to compute n/2 in every iteration is removed

n_2 = n/2;

For (i = 1 to n_2)

{…}

Use of Sentinel Values

- Sentinel values can be used in a search loop which looks for an element in an array. If the required size of the array is 10, declare it as 11. In all search operation, use the 11th position to store the element to be searched.

```
While (i< n) and  (x< >a[i])
{        i= i+1;
}
```

```
a[n+1] = x;
While (x< >a[i])
{        i = i+1;
}
```

Reduce the strength of operations inside loops

```
for (i  = 1 to Num ) {
commission ( i ) = i * Revenue * BaseCommission * Discount
}
```

```
Commission = Revenue * BaseCommission * Discount
for ( i = 1 to Num ) {
commission ( i ) = i * Commission
}
```

Order tests in case & if-else by frequency

```
read (empNo)
case Grade(empNo) of
1: {....}
2: {....}
3: {....}
....
7: {....}
endcase
```

```
read (empNo)
case Grade(empNo) of
7: { ....}
5: {....}
6: {....}
....
1: {....}
endcase
```

Stop testing when you know the result

```
if(a < 10 ) and (b < 20) then
{
  ....
}
```

```
if(a < 10) then
{
  if(b < 20) then
  {
    ....
  }
}
```

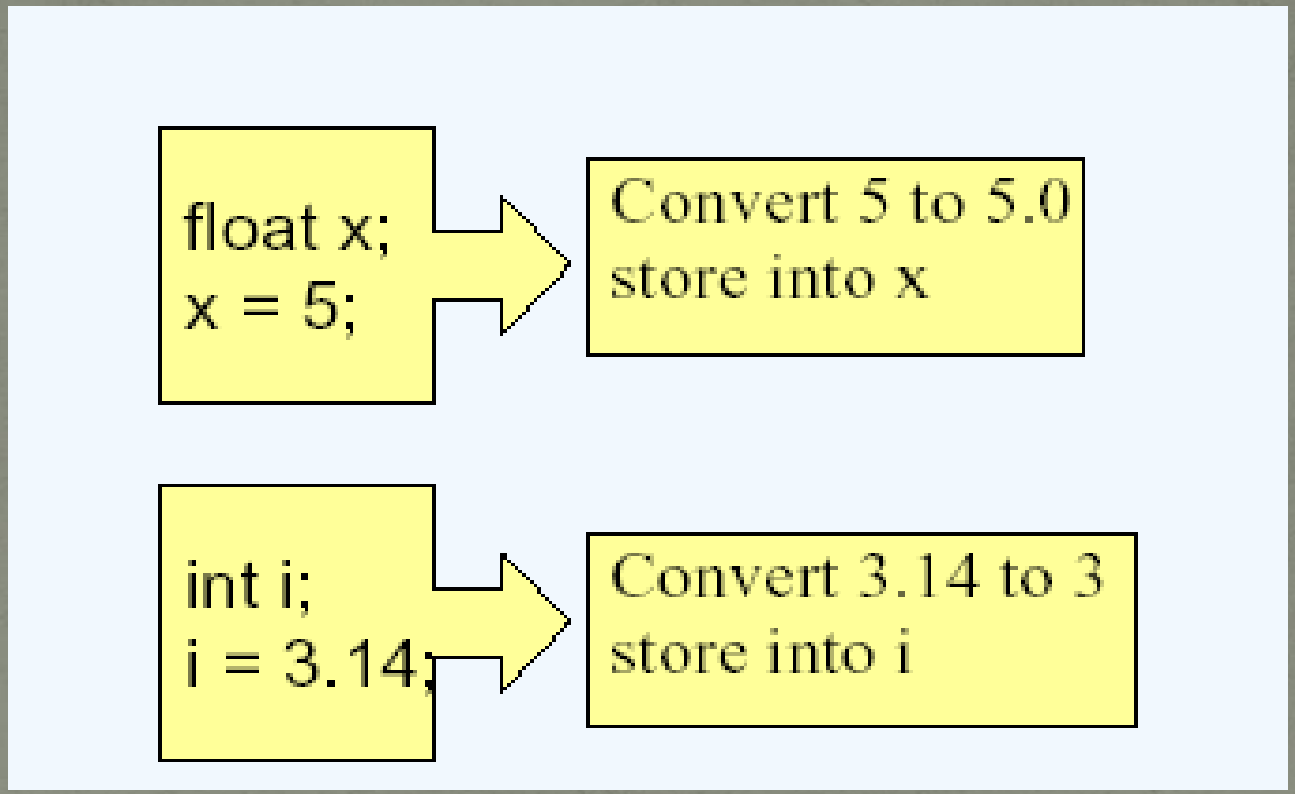
Minimize array references

If the same array element is repeatedly referred to inside a loop, then move it outside the loop

```
for (a=0; a < 5; a++)
{
  for (b =0; b < 10; b++)
  {
     total[b] = total[b] * sum[a];
  }
```

```
for (a=0; a < 5; a++)
{
  sum_now = sum[a];
  for (b =0; b < 10; b++)
  {
     total[b] = total[b] * sum_now;
  }
```

Use constant of correct type



Precompute results

```
y = log(x) / log(2)
b = log(a) / log(2)
```

Four Function

instead have

```
LOG2 = log(2)
y = log(x) / LOG2
b = log(a) / LOG2
```

Two Function

Exploit Algebraic Identities

- Algebraic identities can be used to replace costlier operations by cheaper ones Whenever we need to find whether x < y, we can use the algebraic identity which says x < y only when x < y. So it is enough to check if x < y in this case.
- not (A or B) is cheaper than not A and not B

Common subexpression elimination

Avoid re-computation of expressions

Make use of previously computed value

Example:

x = 2*i

y = 2*i

z = i*2

Transform y = x and z = x at appropriate points or detect i* 2 as a common expression

Dead Code Elimination

Consider the following fragment:

.....

x = 10 ;

y = ... ;

.....

if (x < 100) then y = y + 5 else y = y - 5

else branch will never get executed since the value of ‘x’ cannot be greater than or equal to 100

Lazy Computations

```
n=x*x+2*y+z

if q>10 then
{
……
}
else q > n then
{
…..
}
else
{
…
}
```

Lazy computations

```
if q>10 then
{
……
}
else
{n=x*x+2*y+z
if q > n then
{
…..
}
else
{
…
}}
```

Short circuiting & reordering

- if (a>b) && (c>d) && (e>f)

{

.....

}

Use Coroutines vs. subroutines

- Coroutines can run in single pass rather then multi pass algorithms. In this multiple routines can call each other at the same level instead of a hierarchical level
- Compilers

Euclid Algorithm

- Given two numbers m and n find their greatest common divisor.

1 divide m by n and let r be the remainder

2 if r=0, algorithm terminates; n is the answer

3 set m=n; n=r; and go to step 1

Recursive algorithm

GCD(m,n) = GCD(n,m mod n)

Algorithm without swapping

1 divide m by n and let r be the remainder
2 if r=0, algorithm terminates; n is the answer
3 divide n by r and let m be the remainder
4 if m=0, algorithm terminates; r is the answer
5 divide r by m and let n be the remainder
6 if n=0, algorithm terminates; m is the answer
7. go to step 1

Sorting

- There is nothing more difficult to take in world, more perilous to conduct, or more uncertain in its success than to take the lead in the introduction of new order of things
- Sorting solves togetherness problem
- Aids in searching

Sorting

- Computer Industry says that around 50 percent of the running time on their computers is currently being spent on sorting, when all of the users are taken into account
- What that mean to us
- There are many important applications of sorting
- Many people sort when they should not
- Inefficient algorithms are common in use

Permutations of sorting

- Given N nos. , we have to rearrange them so that

$a1 \leq a2 \leq a3 \leq a4 \leq a5 \leq a6 \leq a7 \leq a8 \ldots\ldots \leq a_{n-1} \leq a_n$

There are n! different permutations possible for this sequence, out of this only 1 is the right sequence

Inversions are also very important in sorting that how many numbers are in reverse order

1st Sorting Type-Sorting by Exchange

- Interchanging pairs of elements that are out of order until no such pairs exist
- Bubble sort works by comparing adjacent elements of an array and exchanges them if they are not in order.
- After each iteration (pass) the largest element bubbles up to the last position of the array. In the next iteration the second largest element bubbles up to the second last position and so on

Bubble Sort

1. Begin
2. For i = 1 to n-1 do

2.1 For j = 1 to n-1-i do

2.2.1 If (a[j+1] < a[j]) then swap a[j] and a[j+1]

3. End

	pass1	pass2	pass3	pass4	pass5	pass6	pass7	pass8	pass9
703	908	908	908	908	908	908	908	908	908
765	703	897	897	897	897	897	897	897	897
677	765	703	765	765	765	765	765	765	765
612	677	765	703	703	703	703	703	703	703
509	612	677	677	677	677	677	677	677	677
154	509	612	653	653	653	653	653	653	653
426	154	509	612	612	612	612	612	612	612
653	426	154	509	512	512	512	512	512	512
275	653	426	154	509	509	509	509	509	509
897	275	653	426	154	503	503	503	503	503
170	897	275	512	426	154	426	426	426	426
908	170	512	275	503	426	154	275	275	275
061	512	170	503	275	275	275	154	170	170
512	061	503	170	170	170	170	170	154	154
087	503	061	087	087	087	087	087	087	087
503	087	087	061	061	061	061	061	061	061

Complexity bubble sort

- We have a nested for loop and hence we need to do an inside out analysis.

Step 2.2.1 at the most performs **2** operations (one comparison and one swapping) and these operations are repeated **((n-1-i)** times. i.e.

Step 2.1 performs **(n-i-1)** operation where **i** varies from **1** to **n-1 (step 2**). So the total number of operations performed by bubble sort is : **(n-1-1)+(n-2-1)+(n-3-1)+…+(n-(n-1) – 1) = (n-2)+(n-3)+(n-4)+…+2 = ((n-2)(n-1))/2 – 1 = $(n^2-3n)/2$**.

- Hence the worst case complexity of bubble sort is **$O(n^2)$**

Enhancements in bubble sort

- traversing in opposite direction in alternate passes
- if two adjacent elements don’t exchange for two consecutive passes then we can fix their position

Insertion Sort

- Before examining record R_j , we assume that the preceding records R1 to Rj-1 have already been sorted and we insert Rj into its proper place among the previously sorted records

Algorithm

- INSERTION-SORT (A, n)⊳A[1 . . n]
- for j ←2 to n do
- key ←A[j]
- i ←j -1
- while i > 0 and A[i] > key
- do A[i+1] ←A[i]
- i ←i -1
- A[i+1] = key

503		087													
087	503		512												
028	503	512		061											
061	087	503	512		908										
061	087	503	512	908		170									
061	087	170	503	512	908		897								
….	….	….	…	..	..	..	..	..	..						
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Improvements

- Two Way insertion
- Binary Search with in sorted sequence

Shell Sort

- If we have a sorting algorithm which moves items only one position at a time, its average running time will be at best proportional to n^2

since each record must travel an average of about n/3 positions during the sorting process. So if we want to make substantial improvements over straight insertion, we need some mechanism by which the records can take long leaps instead of short steps

Shell Sort or diminishing increment sort

	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
8-sort	503	087	154	061	612	170	765	275	653	426	512	509	908	677	897	703
4-sort	503	087	154	061	612	170	765	275	653	426	512	509	908	677	897	703
2-sort	154	061	503	087	512	170	612	275	653	426	765	509	897	677	908	703
1-sort	061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Improvements

- This does not allows the interaction between even and odd keys, so we can try 7-sort,5-sort,3-sort and 1-sort

Address Calculation Sort or multi list sort

- Idea is to use each list for certain range of keys. The set of all possible values of the keys is partitioned into m parts, we provide additional storage of M list heads.
- Suppose we have 16 keys used in our example and divided into 4 parts 0-249, 250-499,500-749,750-999.

	4 items entered	8 items entered		12 items entered		16 items entered
List 1	061,087	061,087, 170		061,087, 154,170		061,087, 154,170
List 2		275		275,426		275,426
List3	503,512	503,512		503,509, 512,653		503,509, 512,612, 653,677, 703
List 4		897,908		897,908		765,897, 908

Quick Sort or partition exchange sort

- Burning the candle at both ends
- Here we partition the given list into two partitions based on the pivot element. The lesser elements come on the left of the pivot and the greater come on the right of the pivot.
- We keep two pointers l and r . Increase l by 1 and continue until encountering a record belonging to right partition. Similarly decrease r by 1 and continue until encountering a record belonging to the left partition. if l< r then exchange these records, and move on to process the next records in the same way.
- All comparisons during a given stage are made against the same key, so this key may be kept in a register and only a single index needs to be changed between comparisons. So in this inner loops of computation become very fast.

l,r 1,16	[503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703]
1,6	[275	087	154	061	426	170]	503	[897	653	908	512	509	612	677	765	703]
1,4	[170	087	154	061]	275	426	503	[897	653	908	512	509	612	677	765	703]
1,3	[061	087	154]	170	275	426	503	[897	653	908	512	509	612	677	765	703]
2,3	061	[087	154]	170	275	426	503	[897	653	908	512	509	612	677	765	703]
8,16	061	087	154	170	275	426	503	[897	653	908	512	509	612	677	765	703]
8,14	061	087	154	170	275	426	503	[765	653	703	512	509	612	677]	897	908
8,13	061	087	154	170	275	426	503	[677	653	703	512	509	612]	765	897	908
8,11	061	087	154	170	275	426	503	[509	653	612	512]	677	703	765	897	908
9,11	061	087	154	170	275	426	503	509	[653	612	512]	677	703	765	897	908
9,10	061	087	154	170	275	426	503	509	[512	612]	653	677	703	765	897	908
--	061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Improvements

- Bottom out when the array size is less then 5 or 10
- Use 3 elements and take their middle element as the pivot element

Selection Sort

- In selection sort, the basic idea is to find the smallest number in the array and swap this number with the leftmost cell of the unsorted array, thereby increasing the sorted part of the array by one more cell
- Selection method requires all the input items to be present before the sorting may proceed. While in insertion sort inputs may be received sequentialy.

Algorihtm

- 1. Begin
- 2. For i = 1 to n-1 do
- 2.1 set min = i
- 2.2 For j = i+1 to n do
- 2.2.1 If (a[j] < a[min]) then set min = j
- 2.3 If (i < min) then swap a[i] and a[min]
- 3. End

	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
	503	087	512	061	703	170	897	275	653	426	154	509	612	677	765	908
	503	087	512	061	703	170	765	275	653	426	154	509	612	677	897	908
	503	087	512	061	703	170	677	275	653	426	154	509	612	765	897	908
	..	..	..	..	..	..	.	..	..	..	..	..	..	..	..	..
	061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Improvements

- Group Selection can lead to nlog n time
- selecting min and max at the same time

Merge Sort

- Means the combination of two or more sorted files into one sorted file
- Dividing
- 503 703 765 087 512 677 908 275
- 503 703 765 087 | 512 677 908 275
- 503 703 | 765 087 512 677 | 908 275
- 503 | 703 765 | 087 512 | 677 908 | 275
- Merging
- 503 703 | 087 765 512 677 | 275 908
- 087 503 703 765 | 275 512 677 908
- 087 275 503 512 677 703 765 908

Distribution Sort- Radix sort

- Start with a distribution sort based on the least significant digit of the keys, moving records from the input area to the auxiliary area. Then do another on the next least significant digit, moving the records back into the original input area and so on.

radix Sorting

- Input area contents **503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703**
- **Counts for unit digit distribution 1 1 2 3 1 2 1 3 1 1**
- **storage allocation based on these counts 1 2 4 7 8 10 11 14 15 16**
- **Auxiliary area contents 170 061 512 612 503 653 703 154 275 765 426 087 897 677 908 509**
- **Counts for ten digit distribution 4 2 1 0 0 2 2 3 1 1**
- **Storage allocation based on these counts** 4 6 7 7 7 9 11 14 15 16
- Input area contents 503 703 908 509 512 612 426 653 154 061 765 170 275 677 087 897
- Counts for hundreds digit distribution 2 2 1 0 1 3 3 2 1 1
- Storage allocations based on these counts 2 4 5 5 6 9 12 14 15 16
- Auxiliary area contents 061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908

Comparison counting sort

- jth key in the final sorted sequence is greater then exactly (j-1) of other keys. This algorithm is important because it does not involve any movement of records

	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
i=N	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=n-1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	12
i=n-2	0	0	0	0	2	0	2	0	0	0	0	0	0	0	13	12
;	0	0	0	0	3	0	3	0	0	0	0	0	0	11	13	12
;	0	0	0	0	4	0	4	0	1	0	0	0	9	11	13	12
;	0	0	1	0	5	0	5	0	2	0	0	7	9	11	13	12
													9	11	13	12
													9	11	13	12
i=2	6	1	8	0	15	3	14	4	10	5	2	7	9	11	13	12

Distribution counting sort

- This is only applicable in the case many equal keys are present, and when all keys fall in the range of u≤ v where (v-u) is small

Distribution counting sort

- U=0 and v=9
- 5, 0, 5,0,9,1,8,2,6,4,1,5,6,6,7,7

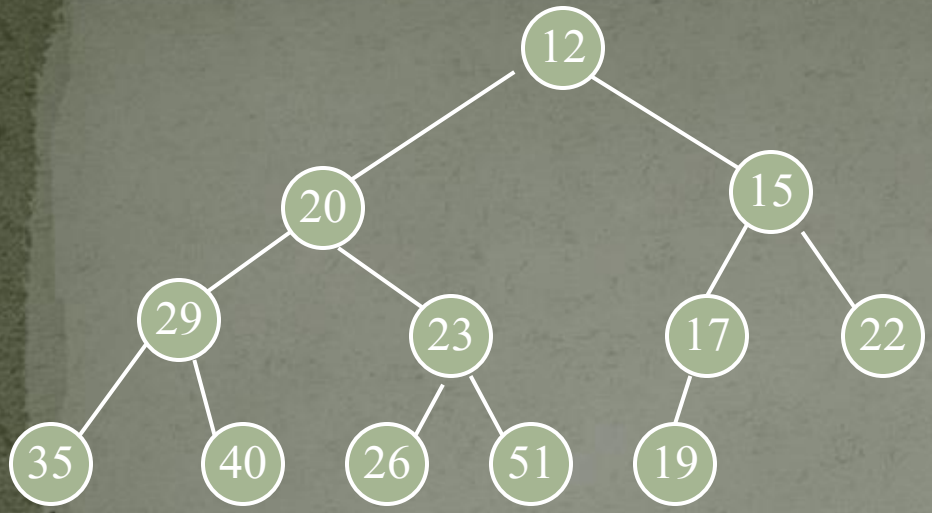
- Count 2 2 1 0 1 3 3 2 1 1
- total count 2 4 5 5 6 9 12 14 15 16
- final list 0 0 1 1 2 4 5 5 5 6 6 6 7 7 8 9

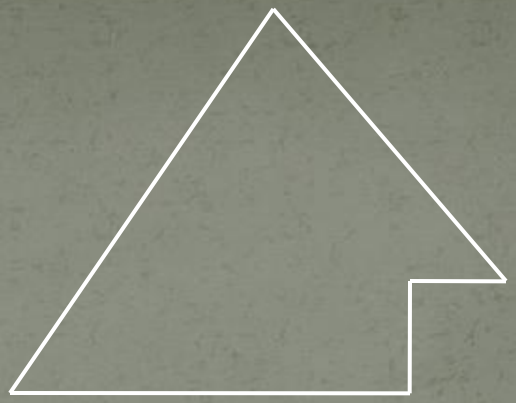
Heap Sort

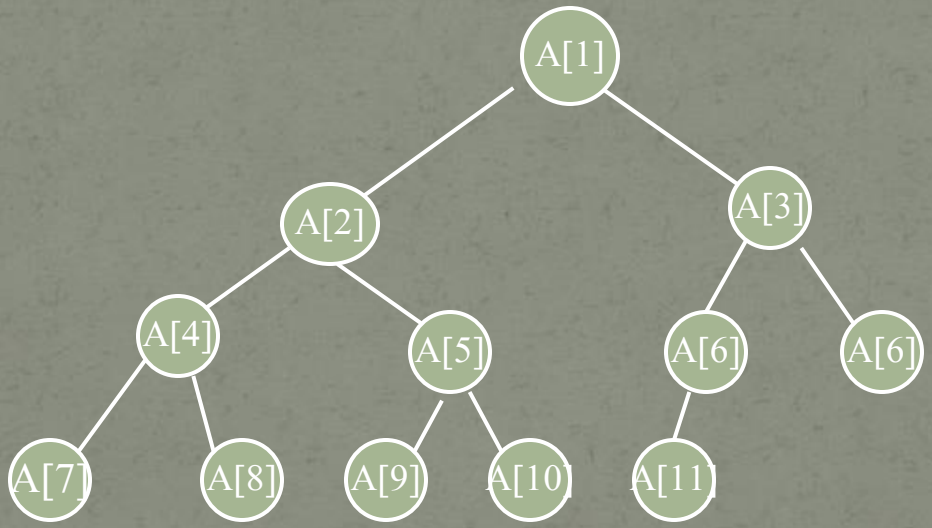
- Any binary tree is a heap by virtue of two properties.
- order : the value at any node is less than or equal to the values of the node's children. So least element of the set is the root of the tree.
- Shape : it has its terminal nodes on at most two levels, with those on the bottom level as far left as possible. There is no hole in the tree and there is no distinction between left and right node
- If an array then we can waste x[0]

functions of heap tree

- root = 1
- value(i) = x[i]
- leftchild(i)= 2*i
- rightchild(i)=2*i+1
- parent(i) = i/2
- null(i) = i< 1 or i>n



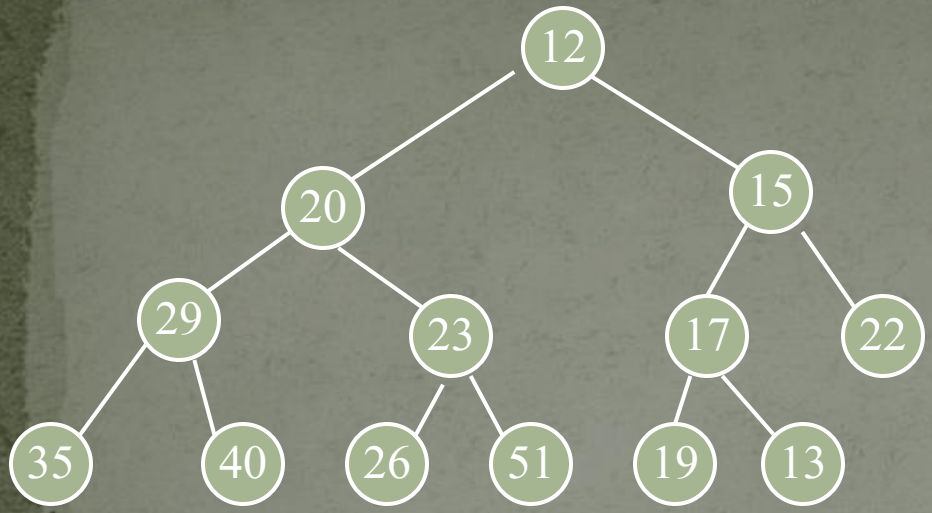


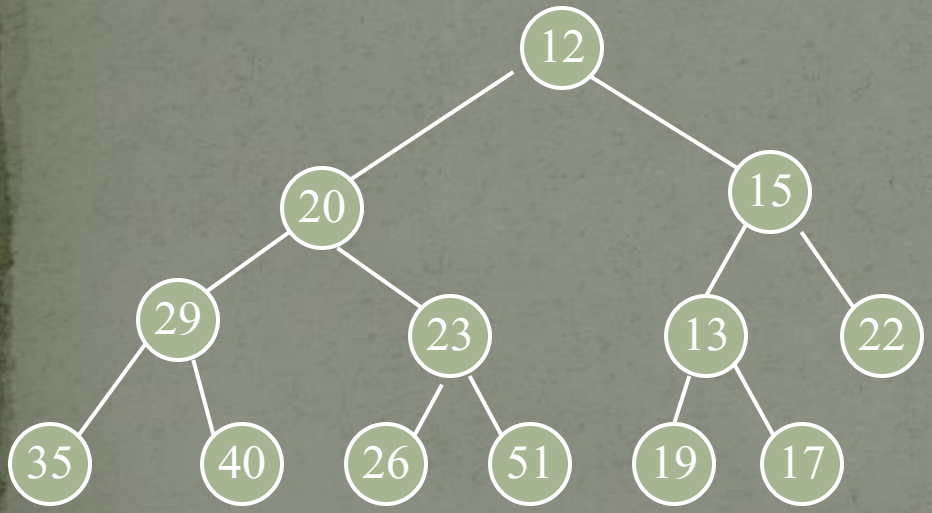


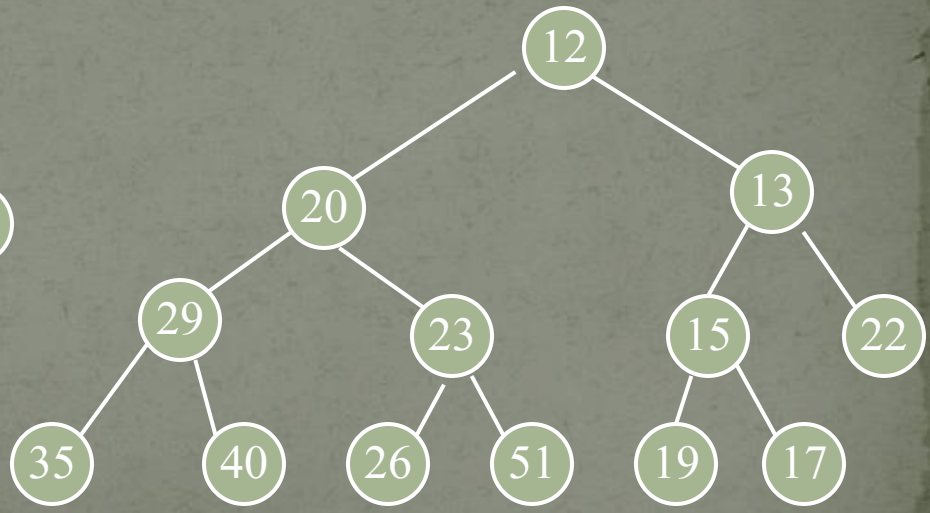
12 20 15 29 23 17 22 35 40 26 51 19

Two critical functions

- Sift up
- placing a 13 at x[n] will not yield a heap so re establishing the property is the job of the function siftup : it sifts the new element up the tree as far as it should go, swapping with its parents along the way. So the loop will end by the check that either it becomes the root or parent is smaller then the element

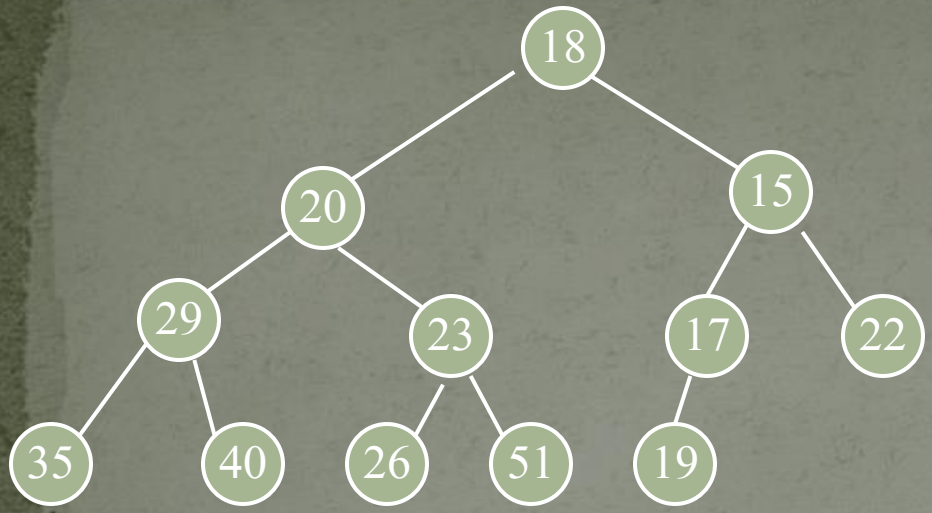

12
20
15
29
23
17
22
35
40
26
51
19
13

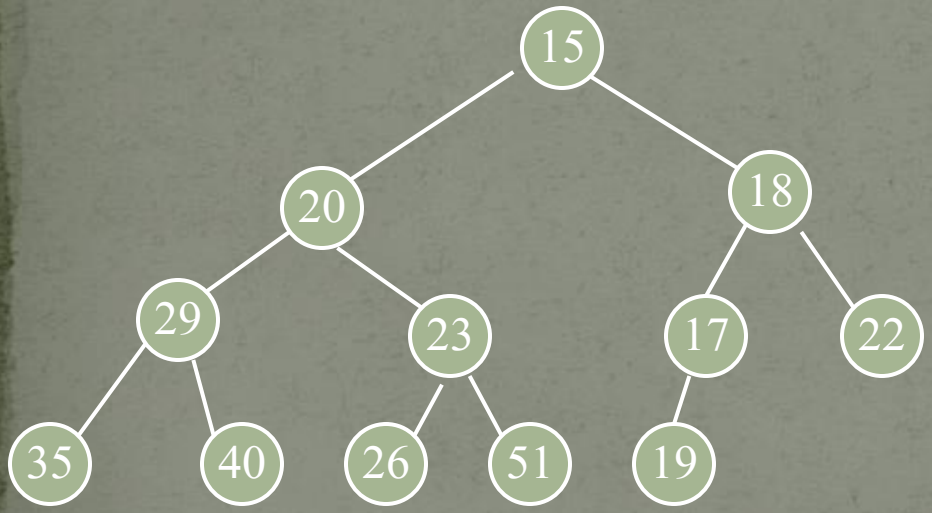

12
20
15
29
23
13
22
35
40
26
51
19
17

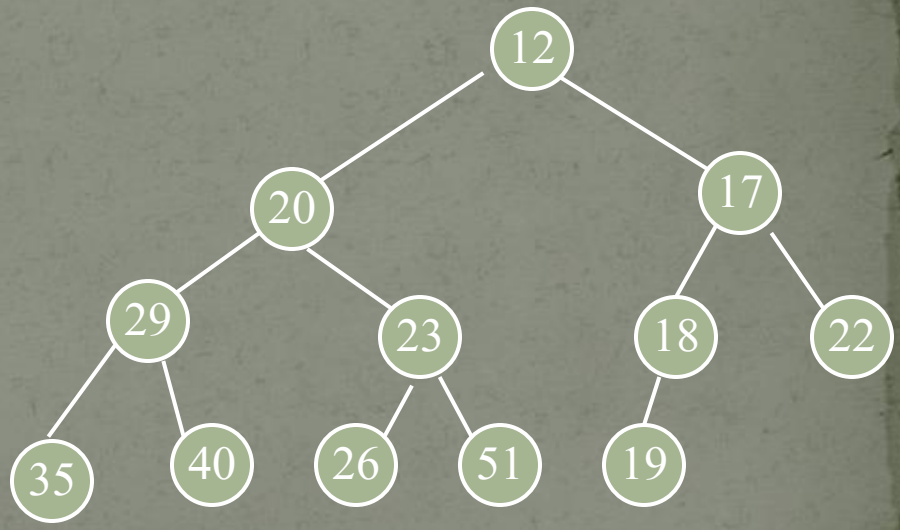

12
20
13
29
23
15
22
35
40
26
51
19
17

sift down

- assigning a new value to x[1] when x[1..n] is a heap leaves heap(2,n): function siftdown makes heap(1,n) true. It does so by sifting x[1] down the array until either it has no children or it is less than or equal to the children it does have


18
20
15
29
23
17
22
35
40
26
51
19


15
20
18
29
23
17
22
35
40
26
51
19


12
20
17
29
23
18
22
35
40
26
51
19

Priority Queue

- A priority queue stores a collection of items
- An item is a pair
  (key, element)
- Main methods of the Priority Queue ADT
  - insertItem(k, o)
    inserts an item with key k and element o
  - removeMin()
    removes the item with the smallest key

Applications

- Additional methods
  - minKey(k, o)
    returns, but does not remove, the smallest key of an item
  - minElement()
    returns, but does not remove, the element of an item with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Implementation with an unsorted sequence

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct items in a priority queue can have the same key
  - Store the items of the priority queue in a list-based sequence, in arbitrary order
- Performance:
  - insertItem takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin, minKey and minElement take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

Implementation with a sorted sequence

- Store the items of the priority queue in a sequence, sorted by key
- Performance:
  - insertItem takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin, minKey and minElement take $O(1)$ time since the smallest key is at the beginning of the sequence

Implementation with a heap

- Insertion taken log(n) time
- removemin(), Minkey(), Minelement() takes O(1) time.
- Priority queues can be best implemented by the heap data structure.
- It can be a min heap or max heap as per our problem requirements

Comparison-Sorting

Algorithm	Worst Case	Average	Space comp	In Place	Stable	
Merge Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n)$	No	Yes	Better external sort
Heap Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n)$	Yes	No	
Quick Sort	$\Theta(n^2)$	$\Theta(n\log n)$	$\Theta(n)$	Yes	No	
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	Yes	Yes	
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	Yes	Yes	
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	Yes	Yes	

Stable- if it leaves the original order of the items with same key value undisturbed
In Place – Input is overwritten by the output as the algorithm executes, however a constant amount of additional space may be used for calculations.
Internal Sort- That takes place entirely in the main memory of the computer.

Design Strategy-brute force

- Closest pair problem
- ${}^{N}C_{2}$ pairs.
- For each pair find distance.
- Find the minimum
- No clever thinking, no strategy

Brute Force

- Brute force approach is a straight forward approach to solve the problem. It is directly based on the problem statement and the concepts
- It is one of the simplest algorithm design to implement and covers a wide range of problems under its gamut

Incremental Design

- Partial solution to be extended, every time you process more input data in incremental stages
- Solution from a smaller sub instance to a larger subinstance
- Solve a1 then a1,a2 then a1,a2,ai then a1,a2,an
- Finding max, adding numbers, insertion sort

Incremental design

- ----------*----------||----------
- You have an infinite wall on both sides where you are standing and it has a gate somewhere in one direction, you have to find out the gate,
- no design-infinite time
- incremental design- you go one step in one direction , come back go to other direction one step, come back and then go 2 steps in other direction

Wall and gate problem

- 1+2.1+1
- 2+2.2+2
- 3+2.3+3
-
- n-1+2.(n-1)+n-1
- n+2n
- $4\sum i+3n$
- $4n(n-1)/2+3n=2n^2+n$

Improvement

- You go 2^0 step in one direction, come back go 1 step in other then 2^1direction in one way then come back and then up to 2^k
- $2^0+2.2^0+2^0$
- $2^1+2.2^1+2^1$
- :
- $2^{K-1}+2.2^{K-1}+2^{K-1}$
- 3.2^K
-
- $4(2^{K-1}+2^{K-2}+\ldots\ldots 1)+ 3.2^K$
- $4(2^K-1)+ 3.2^K$
- $7.2^K -4$
- 7N-4

Power of a number

- Why increment by 1 but by some I units
- A^n a^{16}
- $A^n = A^{n/2} A^{n/2}$
- $= A. A^{(n-1)/2} A^{(n-1)/2}$
- $=A. A^{(n-1)}$
- A^{107} 1101011
- instead of n multiplications it uses 2logn multiplications

Divide and conquer

- Divide and Conquer algorithm design works on the principle of dividing the given problem into smaller sub problems which are similar to the original problem. The sub problems are ideally of the same size.
- These sub problems are solved independently using recursion
- The solutions for the sub problems are combined to get the solution for the original problem

Divide and conquer

- The Divide and Conquer strategy can be viewed as one which has three steps.
- The first step is called Divide which is nothing but dividing the given problems into smaller sub problems which are identical to the original problem and also these sub problems are of the same size.
- The second step is called Conquer where in we solve these sub problems recursively.
- The third step is called Combine where in we combine the solutions of the sub problems to get the solution for the original problem

Decrease and conquer

- Decrease by a constant (factorial)
- decrease by constant factor (Binary search)
- decrease by arbitrary value (GCD)
- avoid too much fragmentation
- split carefully

Divide and conquer

- I divided into I1 and I2
- S1 and S2 are corresponding solutions
- Combine into S

- Quick Sort
- Binary Search
- In quick sort we did not do anything for combining
- In binary search we did not do anything for dividing

Master Theorem

- T(n) = aT(n/b) + f(n) if n>= d

- a= # of sub problems
- b size of sub problem
- f(n) time to combine sub problems

Master Theorem

1. if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
   provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

Case 1: When f(n) is polynomially smaller then the special function $n^{\log_b a}$

2. When f(n) is close to the special function
3. When f(n) is polynomially larger then the special function

Master Theorem

- $T(n) = 4\,T(n/2) + n$
- Solution: $\log_b a = 2$, so case 1 says $T(n)$ is $O(n^2)$.
- $T(n) = 2T(n/2) + n\log n$
- Solution: $\log_b a = 1$, so case 2 says $T(n)$ is $O(n \log^2 n)$.
- $T(n) = T(n/3) + n\log n$
- Solution: $\log_b a = 0$, so case 3 says $T(n)$ is $O(n \log n)$.

Integer Multiplication

- How fast we can multiply
- Normally we require n^2 complexity to multiply two numbers

Integer multiplication

- Multiply two n-bit integers I and J.
  - Divide step: Split I and J into high-order and low-order bits

$$I = I_h 2^{n/2} + I_l$$

$$J = J_h 2^{n/2} + J_l$$

  - We can then define I*J by multiplying the parts and adding:

$$I * J = (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l)$$

$$= I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l$$

  - So, T(n) = 4T(n/2) + n, which implies T(n) is O(n2).
  - But that is no better than the algorithm we learned in grade school.

Improved integer multiplication algo

$$I * J = (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l)$$

$$= I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l$$

$$I * J = I_h J_h 2^n + [(I_h - I_l)(J_l - J_h) + I_h J_h + I_l J_l] 2^{n/2} + I_l J_l$$

$$= I_h J_h 2^n + [(I_h J_l - I_l J_l - I_h J_h + I_l J_h) + I_h J_h + I_l J_l] 2^{n/2} + I_l J_l$$

$$= I_h J_h 2^n + (I_h J_l + I_l J_h) 2^{n/2} + I_l J_l$$

- So, T(n) = 3T(n/2) + n, which implies T(n) is $O(n^{\log_2 3})$, by the Master Theorem.
- Thus, T(n) is $O(n^{1.585})$.

Strassen Multiplication

- Suppose we want to multiply two matrices of size *N* x *N*: for example *A* x *B* = *C*.

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

$$\mathbf{C_{11} = a_{11}b_{11} + a_{12}b_{21}}$$

$$\mathbf{C_{12} = a_{11}b_{12} + a_{12}b_{22}}$$

$$\mathbf{C_{21} = a_{21}b_{11} + a_{22}b_{21}}$$

$$\mathbf{C_{22} = a_{21}b_{12} + a_{22}b_{22}}$$

- 2x2 matrix multiplication can be accomplished in 8 multiplication.**($2^{\log_2 8} = 2^3$)**
- Strassen showed that 2x2 matrix multiplication can be accomplished in 7 multiplication and 18 additions or subtractions. .**($2^{\log_2 7} = 2^{2.807}$)**

- This reduction can be done by Divide and Conquer Approach.

A_0	A_1
A_2	A_3

$\times$

B_0	B_1
B_2	B_3

$=$

$A_0 \times B_0 + A_1 \times B_2$	$A_0 \times B_1 + A_1 \times B_3$
$A_2 \times B_0 + A_3 \times B_2$	$A_2 \times B_1 + A_3 \times B_3$

Strassen

- Divide matrices into sub-matrices: A_0 , A_1, A_2 etc
- Use blocked matrix multiply equations
- Recursively multiply sub-matrices
- Terminate recursion with a simple base case

Strassen

- $P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
  $P_2 = (A_{21} + A_{22}) * B_{11}$
  $P_3 = A_{11} * (B_{12} - B_{22})$
  $P_4 = A_{22} * (B_{21} - B_{11})$
  $P_5 = (A_{11} + A_{12}) * B_{22}$
  $P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$
  $P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$
- $C_{11} = P_1 + P_4 - P_5 + P_7$
  $C_{12} = P_3 + P_5$
  $C_{21} = P_2 + P_4$
  $C_{22} = P_1 + P_3 - P_2 + P_6$

Strassen Algorithm

```
void matmul(int *A, int *B, int *R, int n) {
  if (n == 1) {
    (*R) += (*A) * (*B);
  } else {
    matmul(A, B, R, n/4);
    matmul(A, B+(n/4), R+(n/4), n/4);
    matmul(A+2*(n/4), B, R+2*(n/4), n/4);
    matmul(A+2*(n/4), B+(n/4), R+3*(n/4), n/4);
    matmul(A+(n/4), B+2*(n/4), R, n/4);
    matmul(A+(n/4), B+3*(n/4), R+(n/4), n/4);
    matmul(A+3*(n/4), B+2*(n/4), R+2*(n/4), n/4);
    matmul(A+3*(n/4), B+3*(n/4), R+3*(n/4), n/4);
  }
```

Recursion

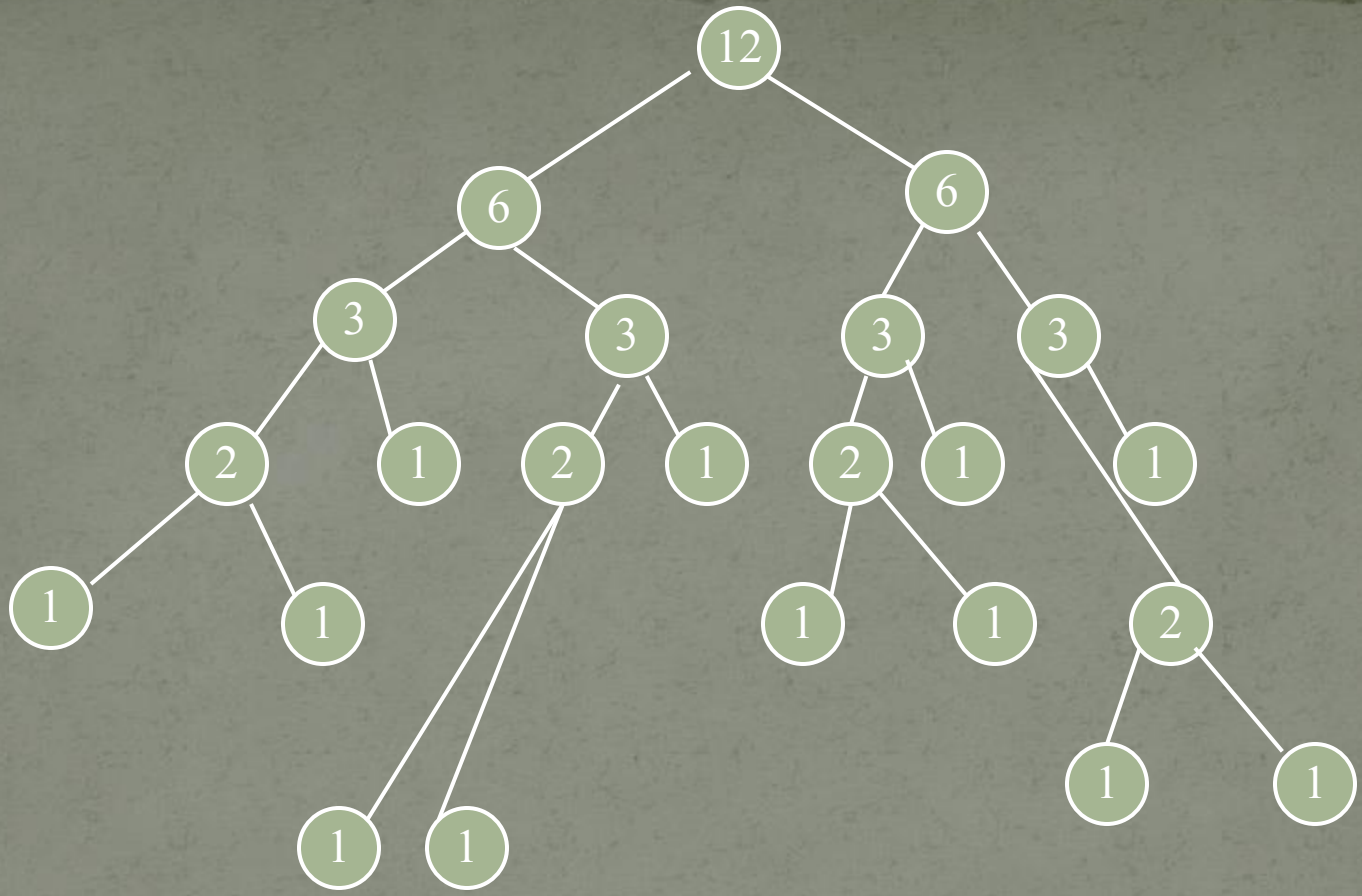
- Recursive formulation is easy to make
- There are lot of function calls
- lot of book keeping
- Should try an equivalent iterative solution

Recursion

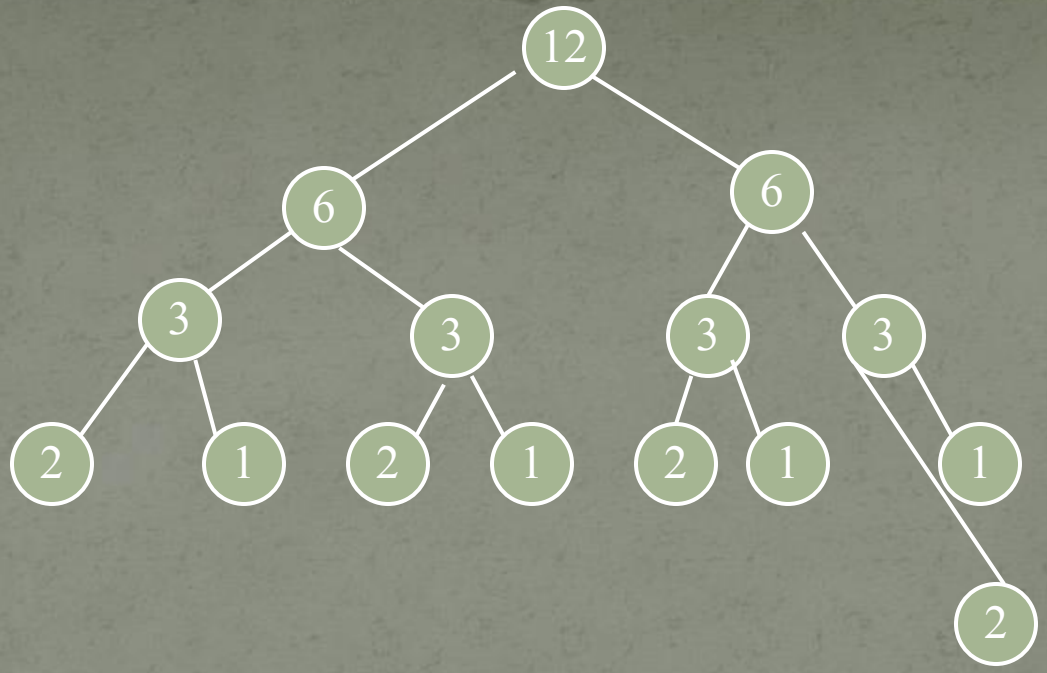
- Single recursive call is like accessing multiple recursions from a tree.
- Minmax(A[1..n], min, max)
- if n=1 min=a, max = a
- else if n=2
- if (a1>a2) max=a1 min=a2
- else min=a1 max=a2
- else recurse

Recursion Vs. iteration

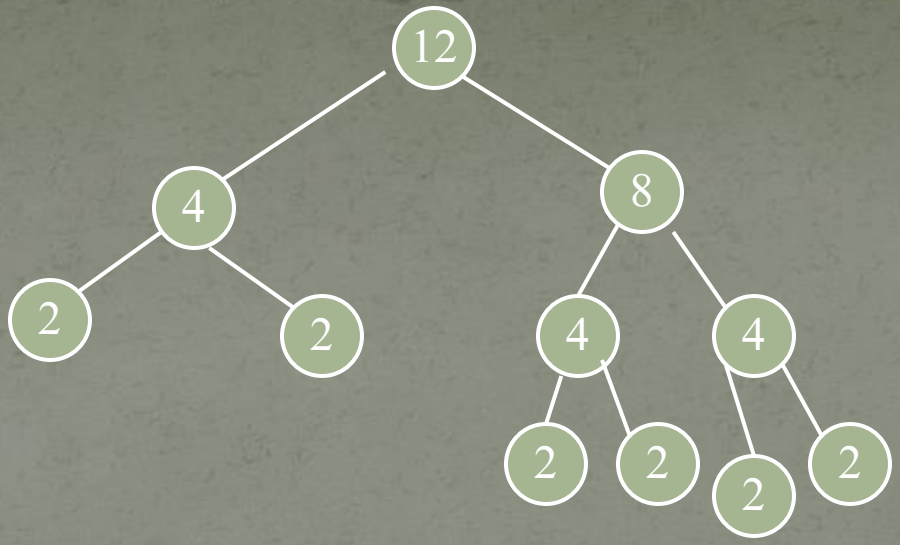
- Terminate at slightly higher levels in turn decreasing the recursive calls
- iteration has less book keeping and less function calls



2n-2



5n/3-2



3n/2 -2

Greedy Programming

- Greedy design technique is primarily used in Optimization problems
- The Greedy approach helps in constructing a solution for a problem through a sequence of steps where each step is considered to be a partial solution. This partial solution is extended progressively to get the complete solution
- The choice of each step in a greedy approach is done based on the following
- It must be feasible
- It must be locally optimal
- It must be irrevocable

Greedy Programming

- Optimization problems are problems where in we would like to find the best of all possible solutions. In other words we need to find the solution which has the optimal (maximum or minimum) value satisfying the given constraints.
- In the greedy approach each step chosen has to satisfy the constraints given in the problem. Each step is chosen such that it is the best alternative among all feasible choices that are available. The choice of a step once made cannot be changed in subsequent steps.

Coin Change Problem

- An amount to reach and an collection of coins to reach to that amount
- Objective is to minimize the number of coins
- Greedy theory says always return the largest coin
- if coins are of denomination 32, 8, 1 it has the greedy choice property because no amount over 32 can be made without omitting 32
- if coins are of denomination 30,20,5,1 then it does not have greedy choice property because 40 is best made with two 20 coins but greedy will return 30,5,5 coins

Activity Selection Problem

- An Activity Selection problem is a slight variant of the problem of scheduling a resource among several competing activities.
- Suppose that we have a set S = {1, 2, …, n} of n events that wish to use an auditorium which can be used by only one event at a time. Each event i has a start time si and a finish time fi where si < fi. An event i if selected can be executed anytime on or after si and must necessarily end before fi. Two events i and j are said to compatible if they do not overlap (meaning si fj or sj fi). The activity selection problem is to select a maximum subset of compatible activities

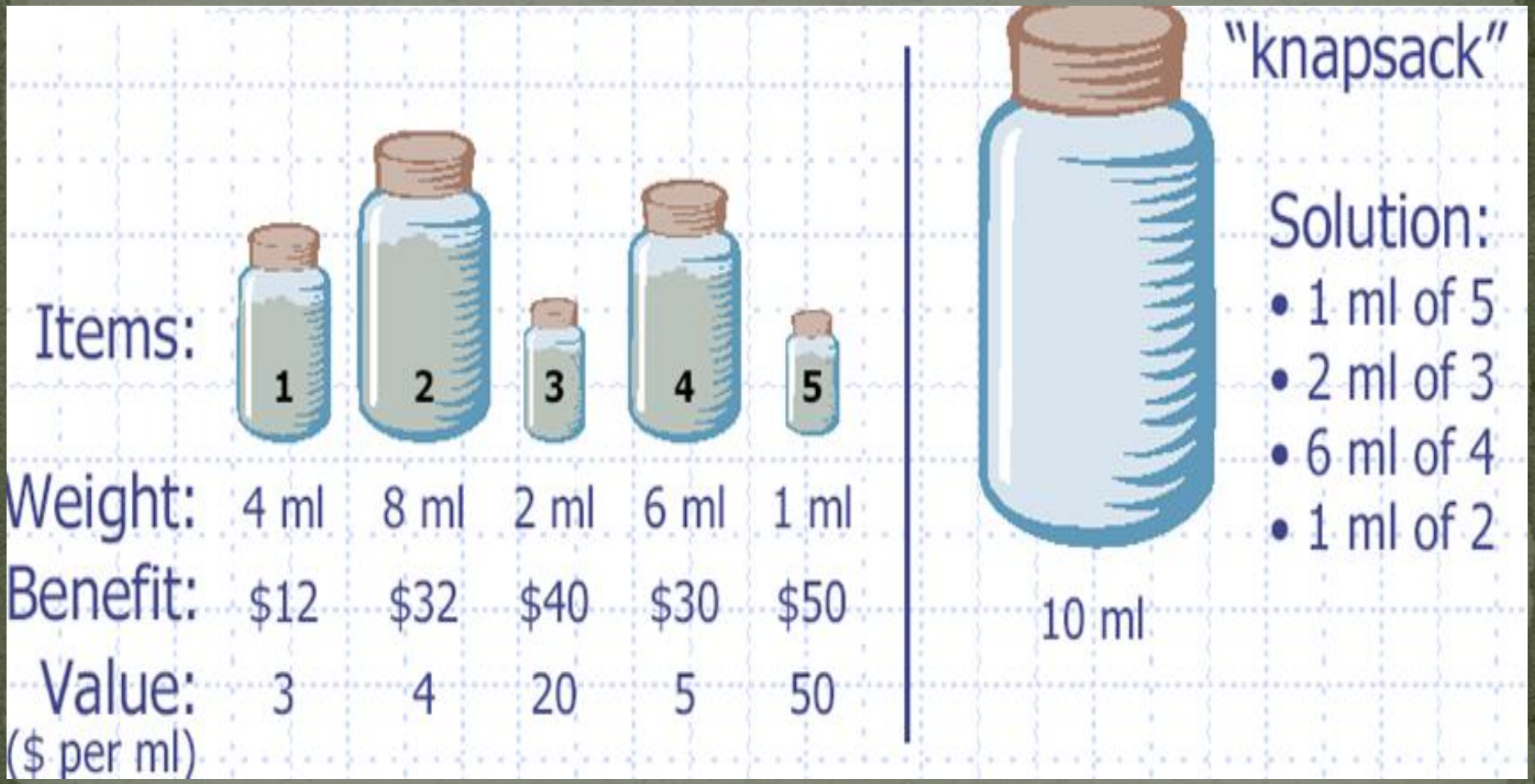
Activity Selection

- 1. Begin
- 2. Set A = {1}, j = 1
- 3. For i = 2 to n do
- 2.1 If (si >= fj) then A = A union {i}, j = i
- 3. End with output as A

Fractional knapsack problem

- We are given n objects and a knapsack. Object i has weight w_i and the knapsack has a capacity M. if a fraction xi, $0 <= x^i <= 1$ of object i is placed into the knapsack then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is M, the total weight of all chosen objects should be M. The profits & weights are positive.

Partial Knapsack



Algorithm fractionalknapsack(s,w)

- Input Set S of items with weight w_i and benefit b_i with total weight W
- Output Amount x_i of each item i to maximize benefit with weight at most W
- for each item i in S
- x_i=0
- v_i=b_i/w_i
- w=0
- while w<W
- remove item i with highest v_i
- x_i=min(w_i,W-w)
- w=w+min(w_i,W-w)

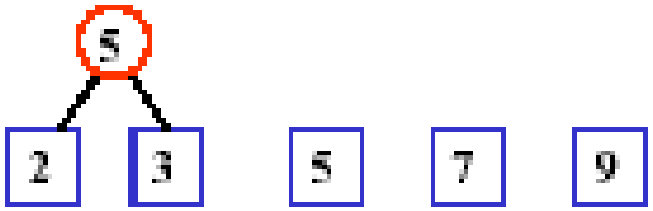
Optimal merge patterns

- Suppose there are 3 sorted lists L1, L2, and L3, of sizes 30, 20, and 10, respectively, which need to be merged into a combined sorted list but we can merge only two at a time.
- We intend to find an optimal merge pattern which minimizes the total number of comparisons.

Optimal merge patterns

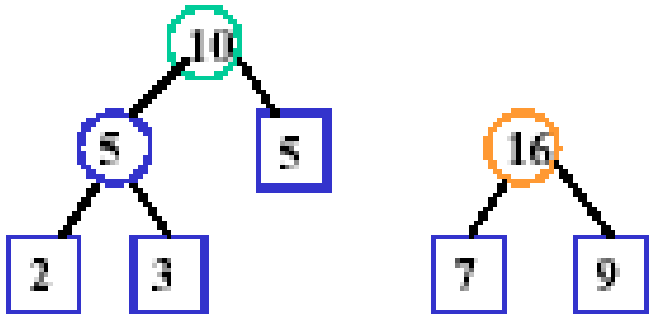
- merge L1 & L2,: 30 + 20 = 50 comparisons , then merge the list & L3: 50 + 10 = 60 comparisons
- total number of comparisons: 50 + 60 = 110.
- Alternatively, merge L2 & L3: 20 + 10 = 30 comparisons, the resulting list (size 30) then merge the list with L1: 30 + 30 = 60 comparisons
- total number of comparisons: 30 + 60 = 90.

Initially, 5 leaf nodes with sizes

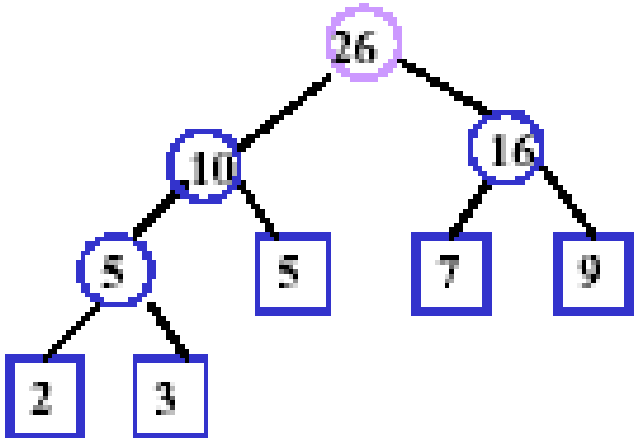


Iteration 1: merge 2 and 3 into 5

Iteration 2: merge 5 and 5 into 10



Iteration 3: merge 7 and 9 (chosen among 7, 9, and 10) into 16



Iteration 4: merge 10 and 16 into 26

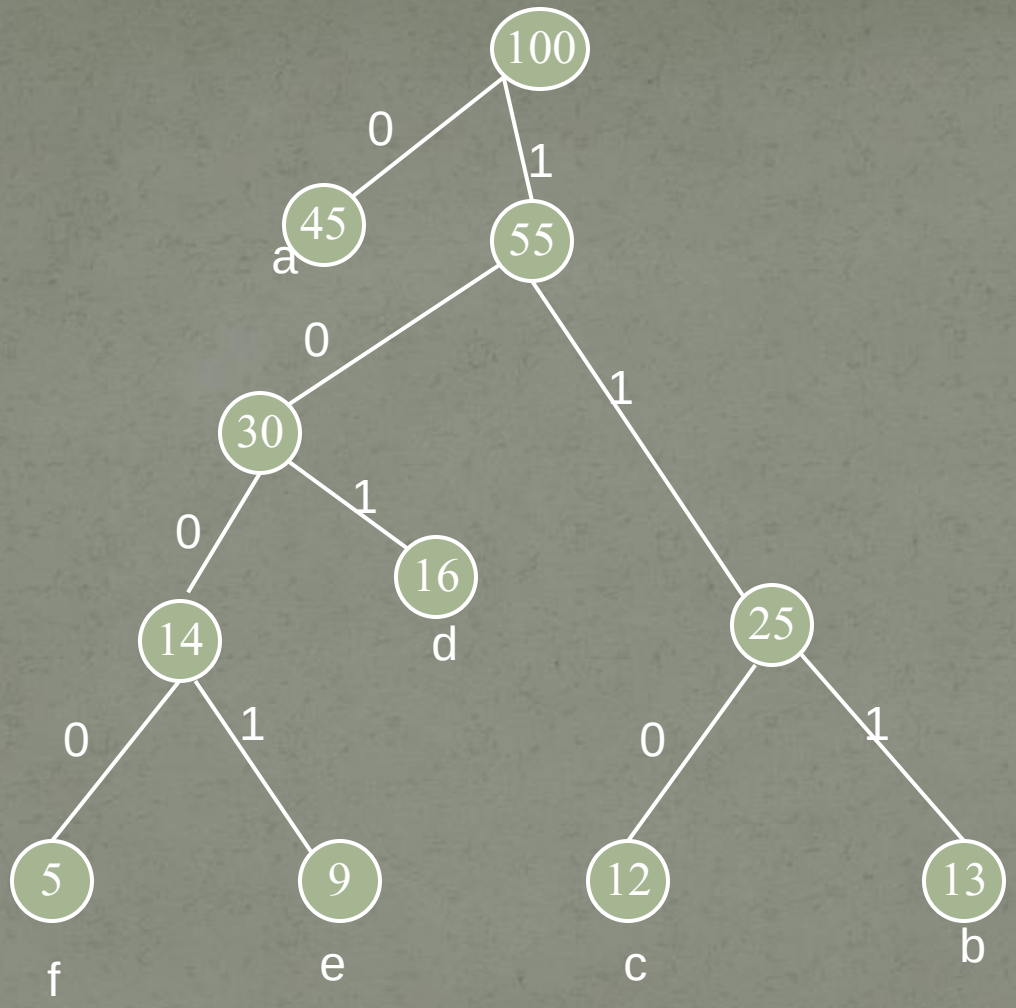
Cost = 2*3 + 3*3 + 5*2 + 7*2 + 9*2 = 57.

File Compression

- File f has one million characters
- only a,b,c,d,e,f chars come in the file then the total size of the file is 8 million bits
- if we represent these with 000 001 010 011 101 111 then only 3 million bits
- 0 1 00 11 10 11 will give us less then 2 million bits

Huffman Code

- assume frequency of occurrence of a, b, c, d, e, f is 45%,13%,12%,16%,9%,5%
- leaves of the binary tree will represent the bit strings
- It will give us the prefix free code
- use optimal merge patterns for making the binary tree and then assign 0 to every left node and 1 to every right node
- The nodes with larger frequency will be nearer to the root and with lesser frequency will be towards the leaves


100
0
1
45
a
55
0
1
30
1
0
16
d
14
25
0
1
0
1
5
9
12
13
f
e
c
b

Huffman Result

- a b c d e f
- 0 111 110 101 1001 1000

- Total = 1*45+3*13+3*12+3*16+4*9+4*5=224
- So total bits used are 2,24,000 bits

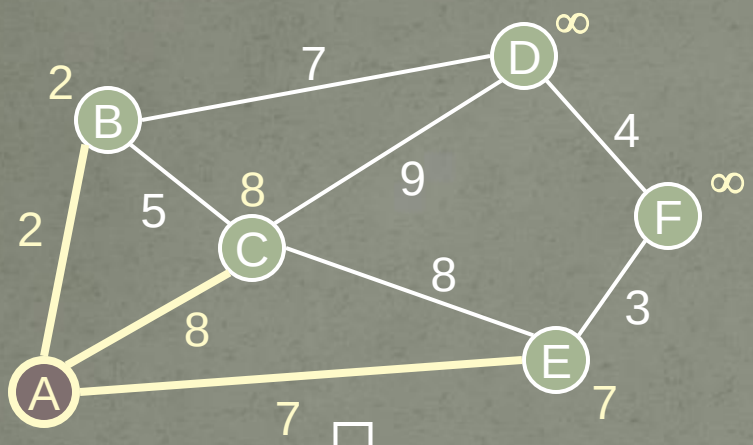
Minimum Spanning tree

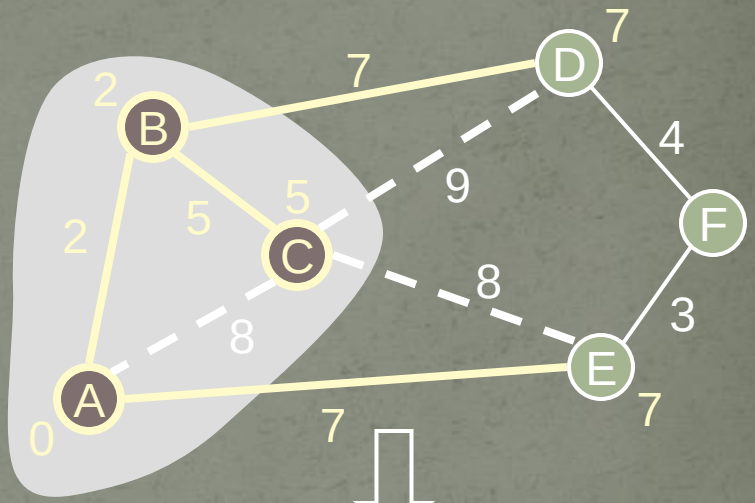
- Spanning subgraph
  - Subgraph of a graph G containing all the vertices of G
- Spanning tree
  - Spanning subgraph that is itself a (free) tree
- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight
- Applications
  - Communications networks
  - Transportation networks

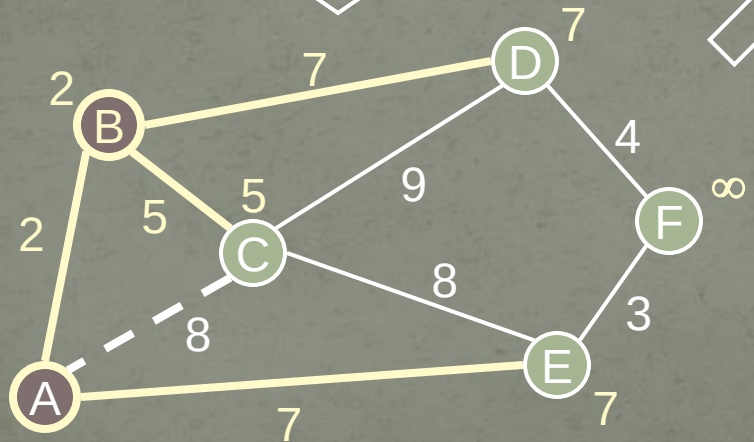
Prim Jarnik Algorithm

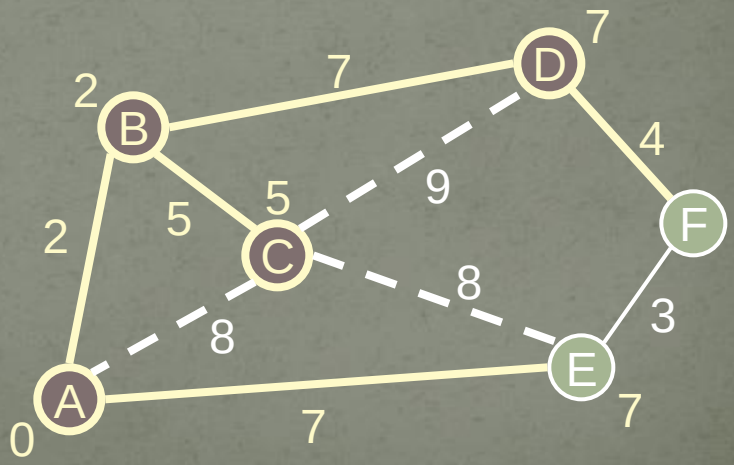
- Similar to Dijkstra's algorithm (for a connected graph)
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v a label $d(v)$ = the smallest weight of an edge connecting v to a vertex in the cloud
  - We add to the cloud the vertex u outside the cloud with the smallest distance label
  - We update the labels of the vertices adjacent to u

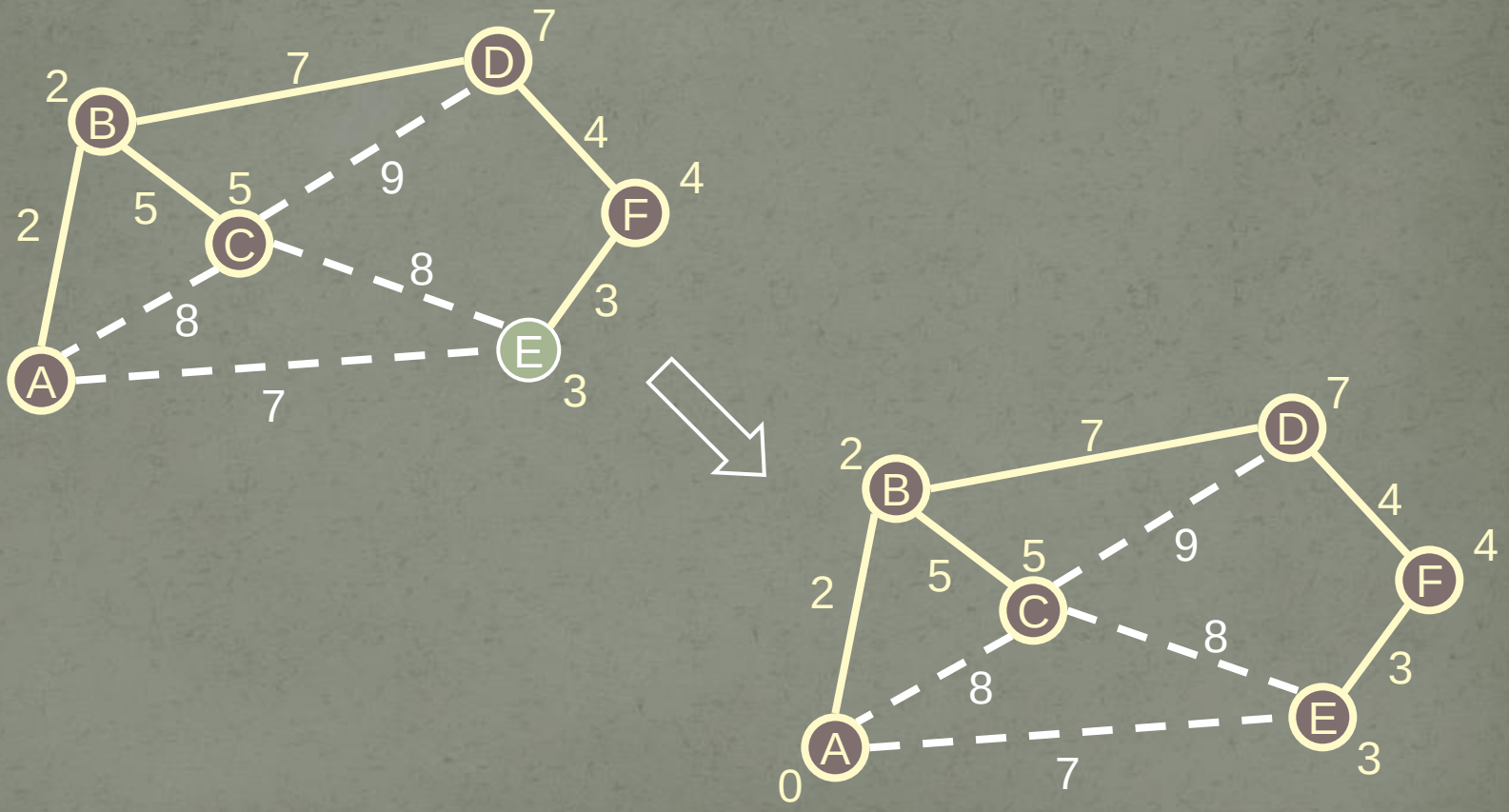
Prim Algorithm










A
B
C
D
E
F
2
7
7
2
5
5
9
4
4
8
8
3
7
3
0

Analysis

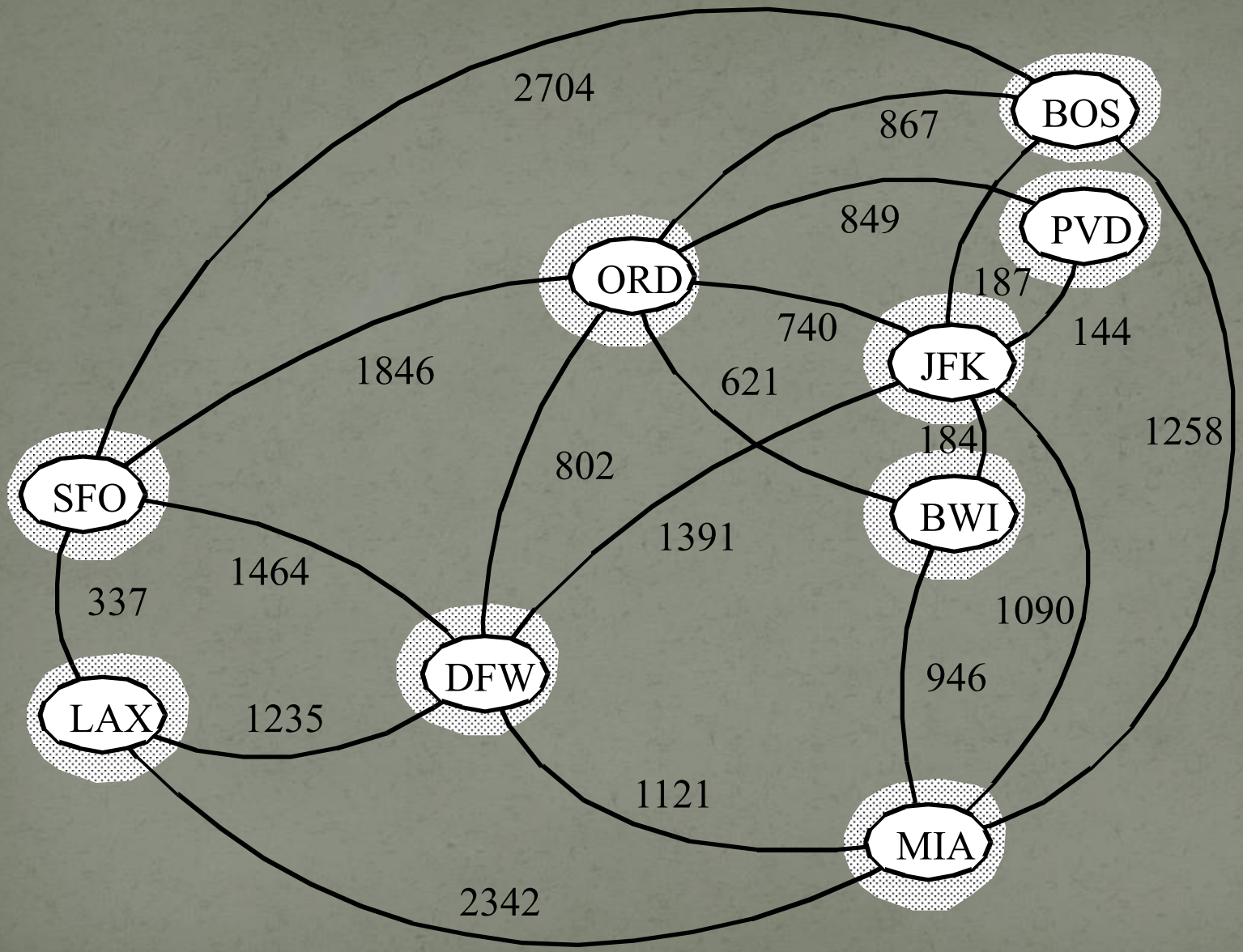
- Graph operations
  - Method incidentEdges is called once for each vertex
- Label operations
  - We set/get the distance, parent and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex w in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Prim-Jarnik's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$
- The running time is $O(m \log n)$ since the graph is connected

Kruskal Algorithm

- The algorithm maintains a forest of trees
- An edge is accepted it if connects distinct trees
- We need a data structure that maintains a partition, i.e., a collection of disjoint sets, with the operations:
- -find(u): return the set storing u
- -union(u,v): replace the sets storing u and v with their union

Analysis

- Each set is stored in a sequence
- Each element has a reference back to the set
- operation find(u) takes O(1) time, and returns the set of which u is a member.
- in operation union(u,v), we move the elements of the smaller set to the sequence of the larger set and update their references
- the time for operation union(u,v) is min(nu,nv), where nu and nv are the sizes of the sets storing u and v
- Whenever an element is processed, it goes into a set of size at least double, hence each element is processed at most log n times
- Running time: O((n+m)log n)


2704
867
849
187
740
144
1846
621
1258
184
802
1391
1464
337
1090
946
1235
1121
2342
BOS
PVD
ORD
JFK
SFO
BWI
DFW
LAX
MIA

Graphs

- A graph is a pair(V,E) where
- V is set of nodes called vertices
- E is a collection of pair of vertices called edges
- vertices and edges are positions and store elements

Directed Edge

- ordered pair of vertices(u,v)
- first vertex u is the origin
- second vertex v is the destination

Undirected Edge

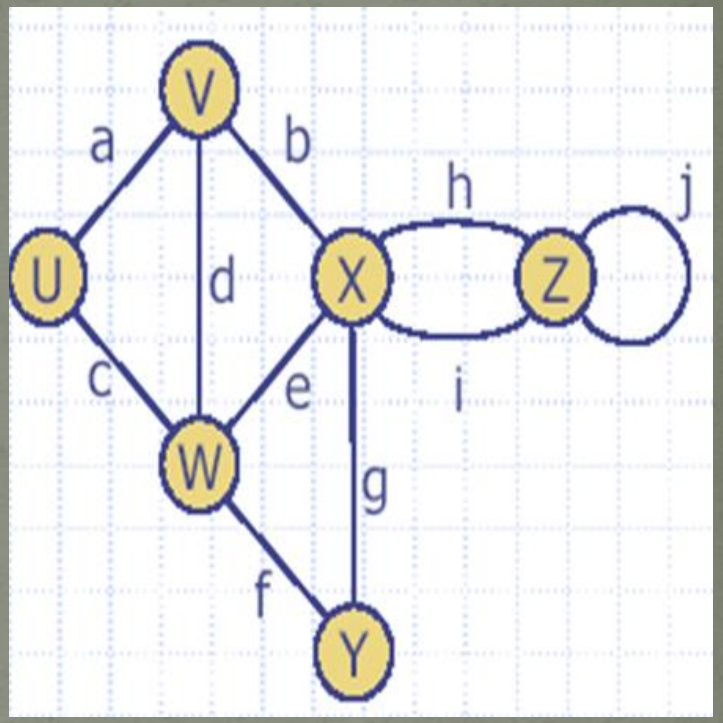
- unordered pair of vertices for example distance between two cities
- In Directed graph all the edges are directed and in undirected graph all the edges are undirected

Applications

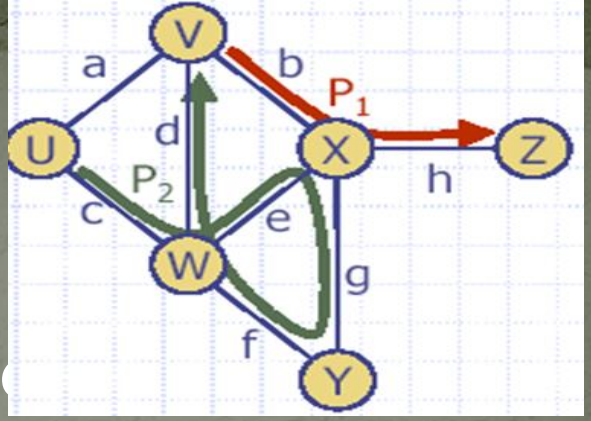
- Electronic circuits
- printed circuit boards
- Integrated Circuits
- Transportation network Highway network , rail network, flight network, traffic network, electricity network, water network
- Computer networks
- LAN
- Web
- Databases- ER diagrams

Terminology

- End vertices of an edge
- U and V are the endpoints of a
- Edge incident on a vertex
- a, d,b are incident on v
- adjacent vertices
- u and v are adjacent
- Degree of a vertex
- x has degree 5
- parallel edges
- h and i are parallel edges
- Self loop
- j is a self loop

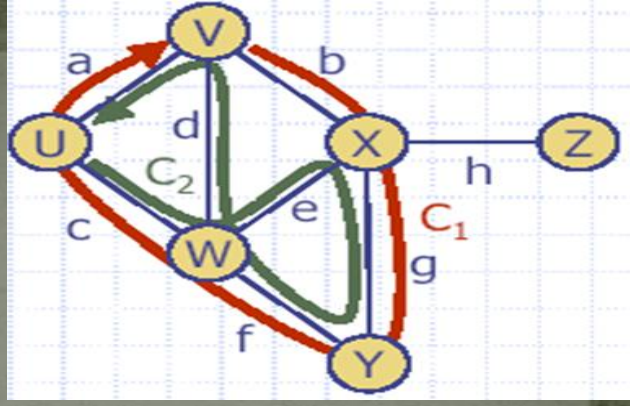


Path



- sequence of alternating vertices an
- begins with a vertex
- ends with a vertex
- each edge is preceded and followed by its endpoints
- Simple path- a path such that all edges and vertices are distinct
- v,b,x,h,z is a simple path
- u,c,w,e,x,g,y,f,w,d,v is not a simple path

cycle



- cycle- circular sequence of alternating vertices and edges
- each edge is preceded and followed by its endpoints
- Simple cycle : such that all its vertices and edges are distinct
- v,b,x,g,y,f,w,c,u,a,v is a simple cycle
- u,c,w,e,x,g,y,f,w,d,v,a, u is not a simple cycle

Incident edges

- incidentedges(v)
- endvertices(e)
- isdirected(e)
- origin(e)
- destination(e)
- opposite(v,e)
- areadjacent(v,w)
- insertvertex(o)
- insertedge(v,w,o)
- insertdirectededge(v,w,o)
- removevertex(v)
- removeedge(e)
- numvertices()
- numedges()

Hashing

- A hash function h maps keys of a given type to integers in a fixed interval [0,N-1]. The goal of a hash function is to uniformly disperse keys in the range [0,N-1]
- Develop a structure that will allow user to insert/delete/find records in
- constant average time
  - structure will be a table (relatively small)
  - table completely contained in memory
  - implemented by an array
  - capitalizes on ability to access any element of the array in constant time

Hashing

- Determines position of key in the array.
- Assume table (array) size is *N*
- Function *f(x)* maps any key *x* to an int between 0 and *N*–1
- For example, assume that *N*=15, that key *x* is a non-negative integer between 0 and MAX_INT, and hash function *f(x)* = *x* % 15

Hashing

- Let $f(x) = x \% 15$. Then,
- if x = 25 129 35 2501 47 36
- $f(x)$= 10 9 5 11 2 6

- Storing the keys in the array is straightforward:

- 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
- _ _ 47 _ _ 35 36 _ _ 129 25 2501 _ _
- Thus, *delete* and *find* can be done in O(1), and also *insert*, except…

Collision

- What happens when you try to insert: $x = 65$?
- $x = 65$
- $f(x) = 5$

- 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
- _ _47 _ _ 35 36 _ _ 129 25 2501 _ _

65(?)

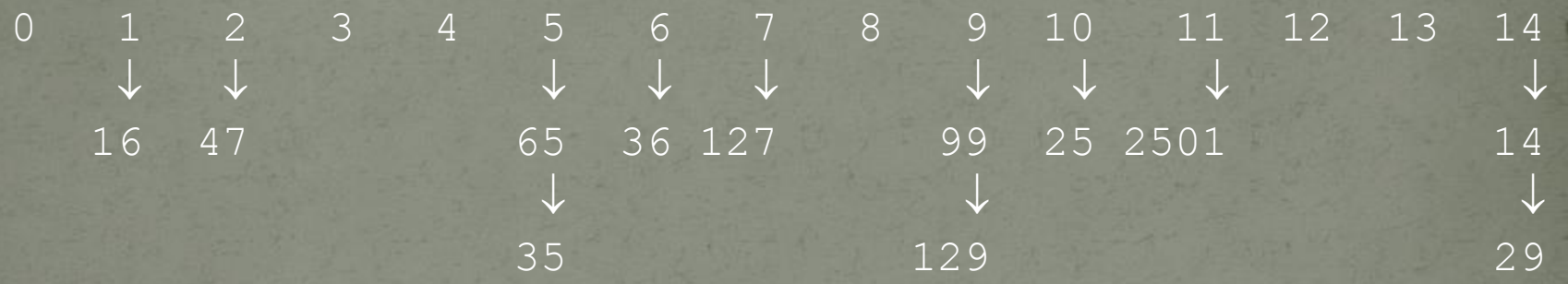
- This is called a *collision*.

Collision Avoidance techniques

- Separate Chaining
- Open Addressing
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

Separate Chaining

- Let each array element be the head of a chain:
- Where would you store: 29, 16, 14, 99, 127 ?



- New keys go at the front of the relevant chain.

separate chaining

- Parts of the array might never be used.
- As chains get longer, search time increases to O(n) in the worst case.
- Constructing new chain nodes is relatively expensive (still constant time, but the constant is high).
- Is there a way to use the "unused" space in the array instead of using chains to make more space?

linear probing

- Let key *x* be stored in element *f(x)=t* of the array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36			129	25	2501			
					65(?)									

- What do you do in case of a collision?
  - If the hash table is *not full*, attempt to store key in array elements (*t*+1)%*N*, (*t*+2)%*N*, (*t*+3)%*N* ...
- until you find an empty slot

linear probing

- If the hash table is *not full*, attempt to store key in array elements (t+1)%N, (t+2)%N, ...

- Where would you store: 65,29,16,14,99,127 ?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	16	47			35	36	65	127	129	25	2501	29	99	14

linear probing

- Eliminates need for separate data structures (chains), and the cost of constructing nodes.

- Leads to problem of clustering. Elements tend to cluster in dense intervals in the array.
- Search efficiency problem remains.
- Deletion becomes trickier….

quadratic probing

- Let key *x* be stored in element *f(x)=t* of the array

```
0   1   2   3  4  5   6   7   8   9  10   11  12  13  14
        47         35  36         129  25 2501
                   65(?)
```

- What do you do in case of a collision?
  - If the hash table is *not full*, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$, $(t+3^2)\%N$ ...
- until you find an empty slot.

quadratic probing

- If the hash table is *not full*, attempt to store key in array elements $(t+1^2)\%N$, $(t+2^2)\%N$ ...

- Where would you store: 65,29,16,14,99,127 ?

```
 0   1   2   3   4   5   6   7   8   9  10   11  12  13  14
29  16  47  14      35  36 127      129  25 2501      99  65
                        ↑
                        t
                    attempts
```

Quadratic probing

- Tends to distribute keys better than linear probing
- Alleviates problem of clustering
- Runs the risk of an infinite loop on insertion, unless precautions are taken.
- E.g., consider inserting the key 16 into a table of size 16, with positions 0, 1, 4 and 9 already occupied.
- Therefore, table size should be prime

Double hashing

- Use a hash function for the decrement value
  - Hash(key, i) = H_1(key) – (H_2(key) * i)
- Now the decrement is a function of the key
  - The slots visited by the hash function will vary even if the initial slot was the same
  - Avoids clustering
- Theoretically interesting, but in practice slower than quadratic probing, because of the need to evaluate a second hash function.

Factors affecting efficiency

- Choice of hash function
- Collision resolution strategy
- Load Factor

- Hashing offers excellent performance for insertion and retrieval of data

Dynamic Programming

- Dynamic programming always gives a correct solution
- Steps:
- Obtain recursive formulation of optimization problem
- Subproblem space
- Dependency relation -> order in which solutions of subproblems are to be obtained
- build the solution in that order and get the solution

Dynamic Programming

- Dynamic Programming is a design principle which is used to solve problems with overlapping sub problems
- It solves the problem by combining the solutions for the sub problems
- The difference between Dynamic Programming and Divide and Conquer is that the sub problems in Divide and Conquer are considered to be disjoint and distinct various in Dynamic Programming they are overlapping

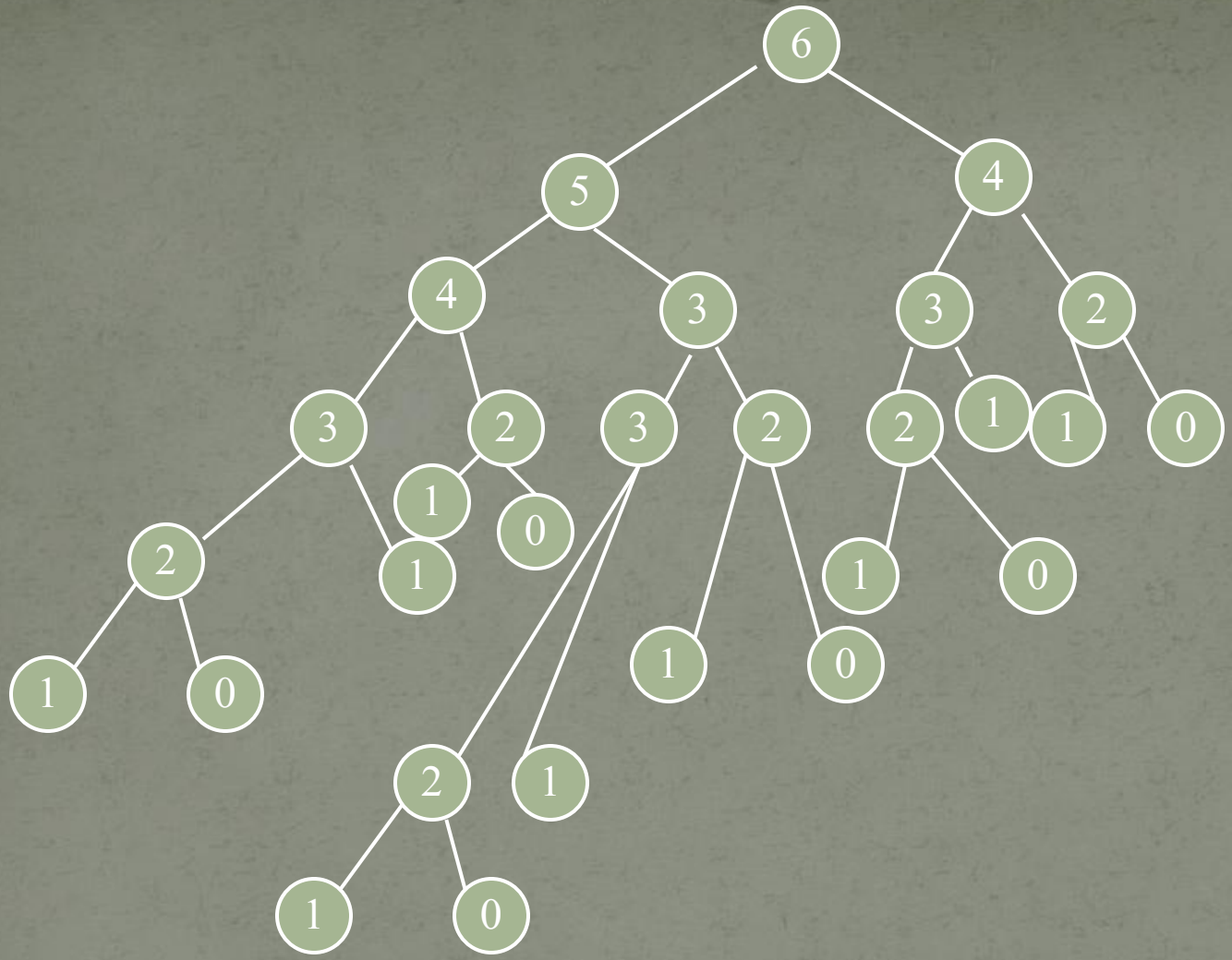
Properties of DP Problems

- Simple Subproblems
  - We should be able to break the original problem to smaller subproblems that have the same structure
- Optimal Substructure of the problems
  - The solution to the problem must be a composition of subproblem solutions
- Subproblem Overlap
  - Optimal subproblems to unrelated problems can contain subproblems in common

Dynamic programming

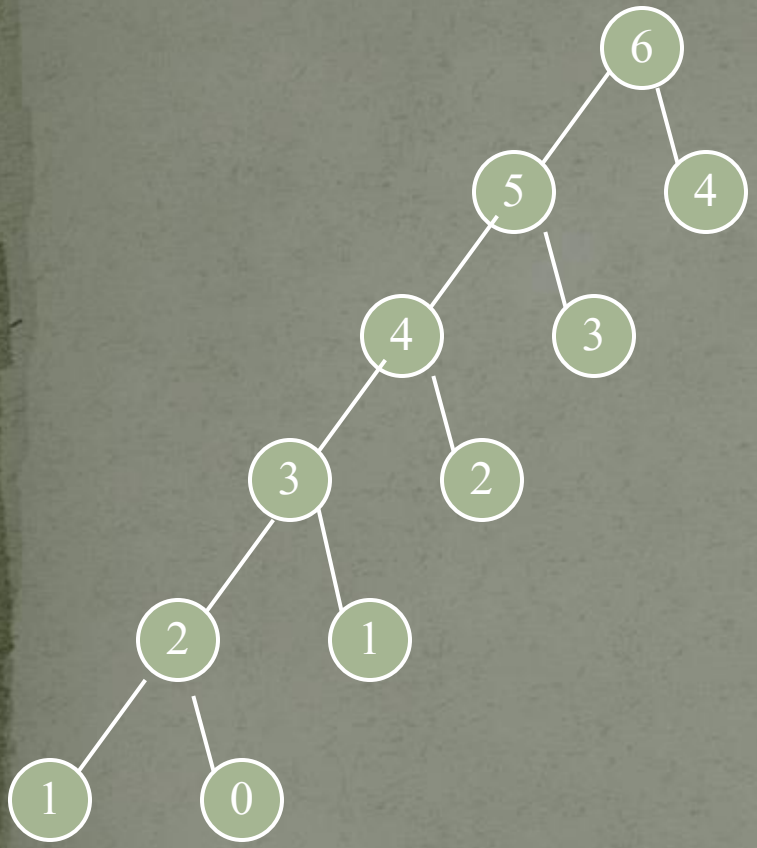
- Fibonacci Series
- $f_n=f_{n-1}+f_{n-2}$
- $f_1=1$
- $f_0=0$

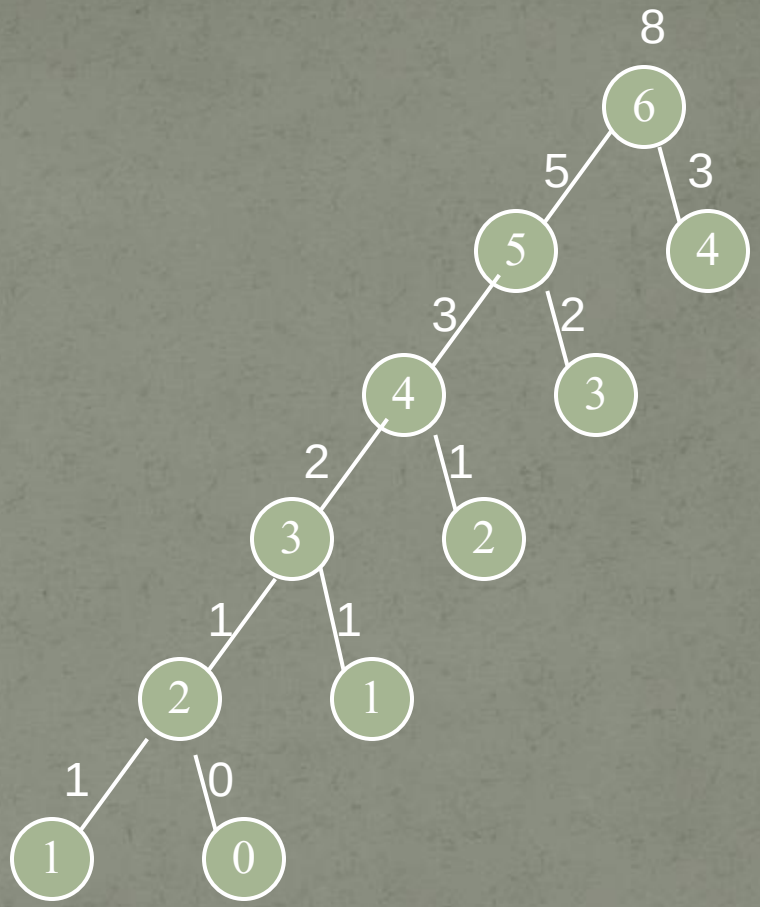
- fib(n)
- if (n=0 or 1) return n
- else
- fib=fib(n-1) + fib(n-2)
- return fib


6
5
4
4
3
3
2
3
2
3
2
2
1
1
0
2
1
1
0
1
0
1
0
1
0
2
1
1
0

Exponential recurrence relation

- It will be exponential recursive call.
- we are repeating the computation of common overlapping sub problems
- “if you forget the past you are condemned to repeat it”
  f[0] f[1] f[2] f[3] f[4] f[5] f[6]
- fib(n)
- if f[n]<> -1 then return f[n]
- else f[n] = fib(n-1)+fib(n-2)
- return f[n]
- from exponential to linear


6
5
4
4
3
3
2
2
1
1
0


8
6
5
3
5
4
3
2
4
3
2
1
3
2
1
1
2
1
1
0
1
0

Application of Dynamic Programming

- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - Simple subproblems: the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - Subproblem optimality: the global optimum value can be defined in terms of optimal subproblems
  - Subproblem overlap: the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

Matrix multiplication

- B is 3 × 100
- C is 100 × 5
- D is 5 × 5
- (B*C)*D takes 1500 + 75 = 1575 ops
- B*(C*D) takes 1500 + 2500 = 4000 ops

Matrix multiplication

- Original matrix chain product
  A x B x C x D x E (ABCDE for short)
- Calculate in advance the cost (multiplies)
  AB, BC, CD, DE
- Use those to find the cheapest way to form
  ABC, BCD, CDE
- then
- ABCD,BCDE
- From that derive best way to form
  ABCDE

Parenthesization

- Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
- Calculate number of ops for each one
- Pick the one that is best

- Running time:
  - The number of paranethesizations is equal to the number of binary trees with n nodes
  - This is **exponential**!
  - It is called the Catalan number, and it is almost 4^n.
  - This is a terrible algorithm!

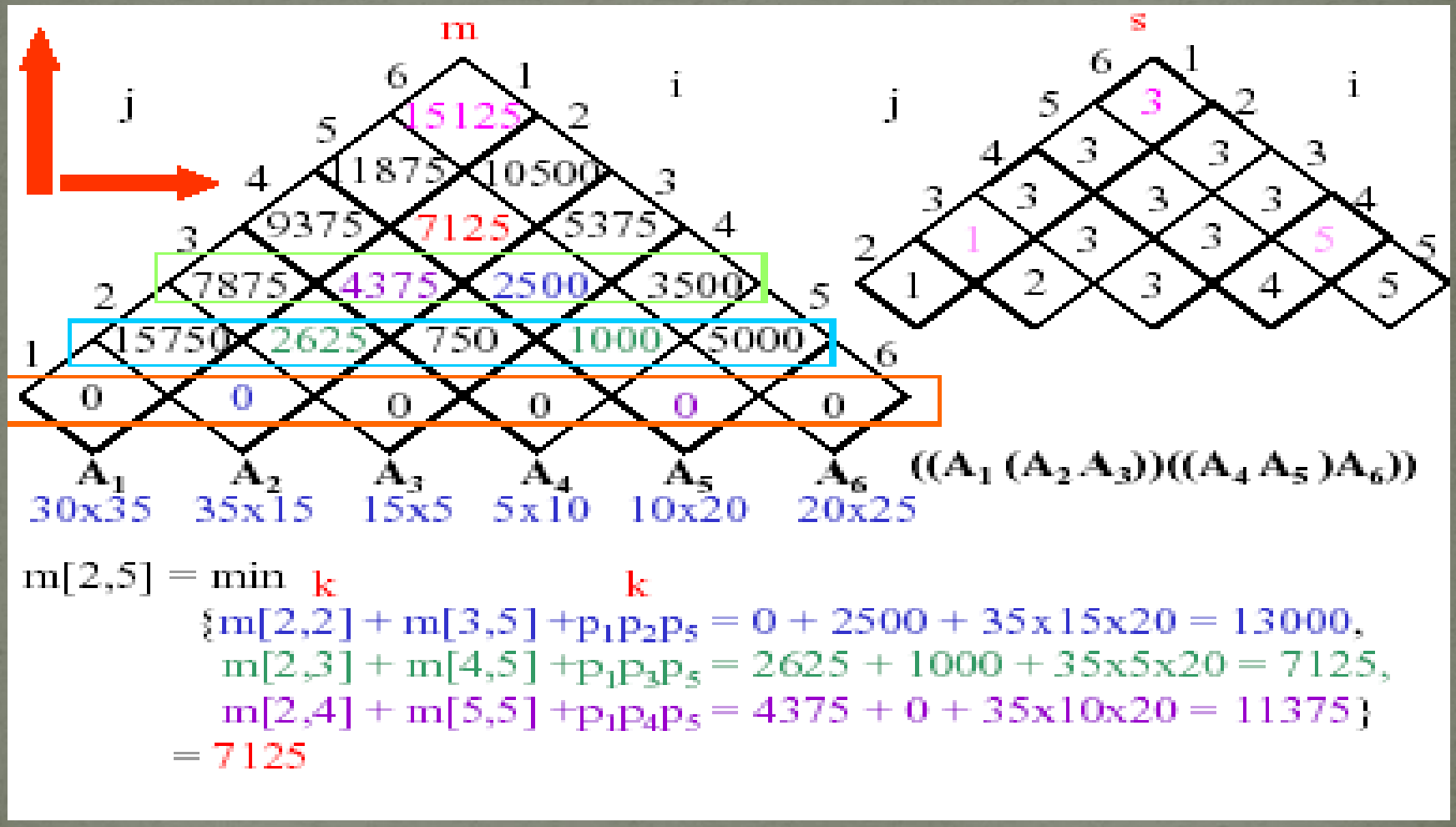
Developing recurrence

- I' is the subscript of the first matrix in the sub problem
- 'j' is the subscript of the last
- 'k' is the subscript for the way to break $A_{i,j}$ into sub problems $A_{i,k}$ and $A_{k+1,j}$
- The problem of determining the optimal sequence of multiplications is broken into two parts
- How do we decide where to split the chain(which k)
- try for all values of k
- how do we parenthesize the sub chains from $A_{1..k}$ and $A_{k+1...n}$

Optimal recurrence relation

- The optimal cost can be described as follows
- if i=j then sequence contains only 1 matrix so m[i,j]=0
- i<j then min (m[i,k]+m[k+1,j] $+d_0 d_k d_j$)

Example

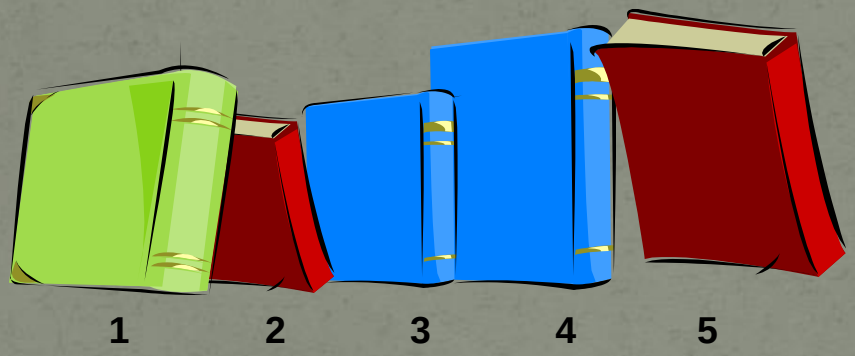


$m[2,5] = \min_k \{ m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \times 15 \times 20 = 13000,$
$m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \times 5 \times 20 = 7125,$
$m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \times 10 \times 20 = 11375 \}$
$= 7125$

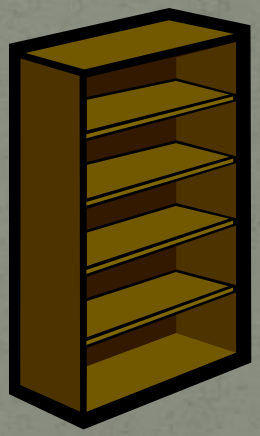
0/1 knapsack problem

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are **not** allowed to take fractional amounts, then this is the **0/1 knapsack problem**.
  - In this case, we let T denote the set of items we take
  - Objective: maximize benefit
  - Constraint the knapsack weight should not exceed W

Example



	1	2	3	4	5
Weight:	4 in	2 in	2 in	6 in	2 in
Benefit:	$20	$3	$6	$25	$80



9 in

"knapsack"

Solution:
- 5 (2 in)
- 3 (2 in)
- 1 (4 in)

- S_k: Set of items numbered 1 to k.
- Define B[k,w] = best selection from S_k with weight exactly equal to w
- this does have subproblem optimality:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{ B[k-1,w],\ B[k-1,w-w_k] + b_k \} & \text{else} \end{cases}$$

- I.e., best subset of S_k with weight exactly w is either the best subset of S_{k-1} w/ weight w or the best subset of S_{k-1} w/ weight w-w_k plus item k.

Algorithm

- Running time: O(nW).
- Not a polynomial-time algorithm if W is large
- This is a pseudo-polynomial time algorithm

Algorithm ***01Knapsack*(*S, W*)**:

Input: set S of items w/ benefit b_i and weight w_i; max. weight W

Output: benefit of best subset with weight at most W

for $w \leftarrow 0$ **to** W **do**

$B[w] \leftarrow 0$

- for i = 1 to n
- B[i,0] = 0

for $k \leftarrow 1$ **to** n **do**

for $w \leftarrow W$ **downto** w_k **do**

if $B[w-w_k]+b_k > B[w]$ **then**

$B[w] \leftarrow B[w-w_k]+b_k$

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	20	20	20	20	20	20
2	0	0	3	3	20	20	23	23	23	23
3	0	0	6	6	20	20	26	26	29	29
4	0	0	6	6	20	20	26	26	31	31
5	0	0	80	80	86	86	100	100	106	106

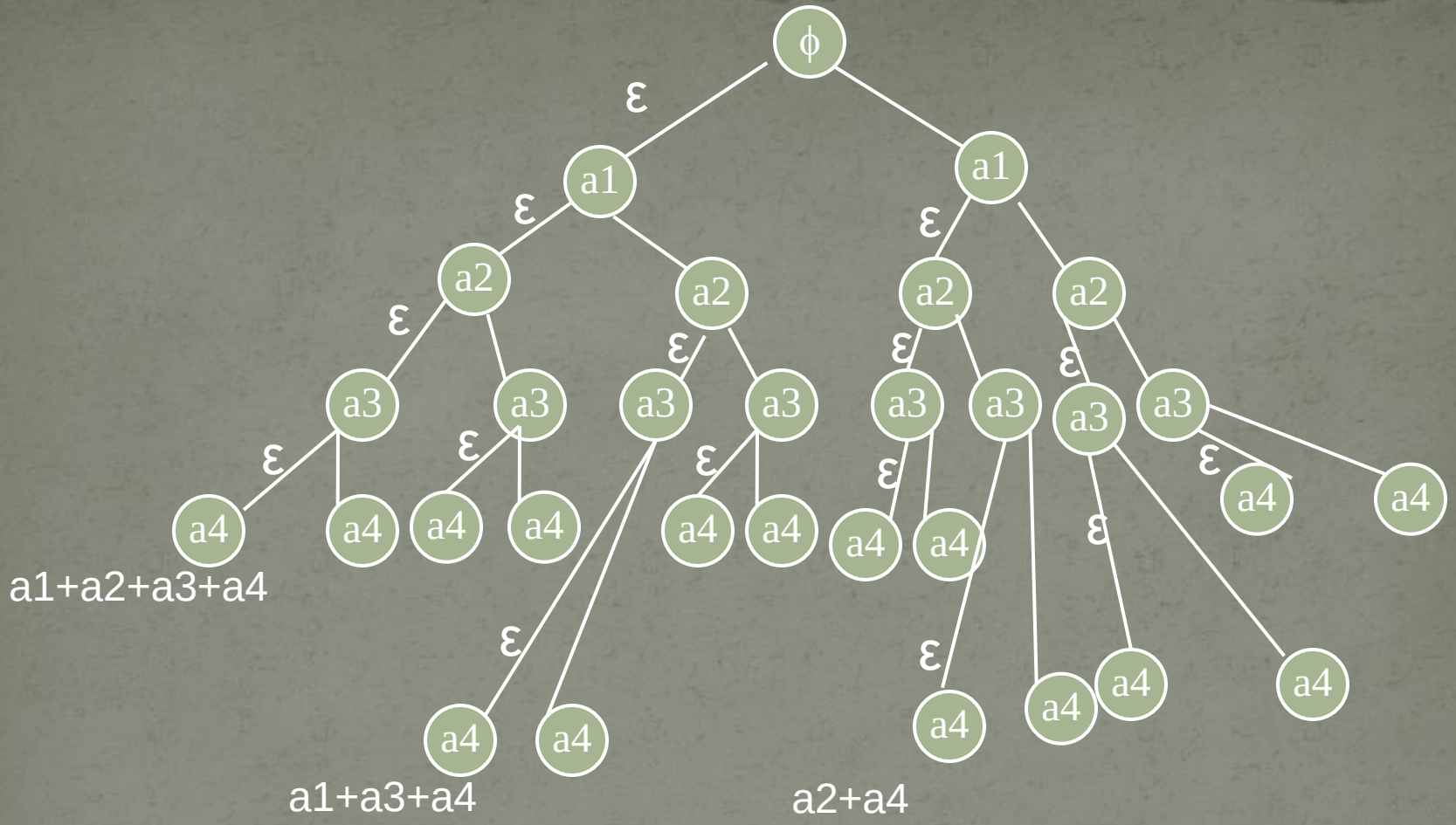
Item:W,P
1 : 4,20
2: 2,3
3 : 2,6
4 : 6,25
5 :2,80

Pruning strategy

- Suitable paradigm where output is a subset of input
- not proper for sorting or computing a formula
- Raman Maharishi said “Who am I” Use pruning
- Binary Search always pruning 50%

sum of subset problem

- S= {14,10,24,7,9,1} ={a1,a2,a3,a4,a5,a6}
- Output all subsets who add upto 31 or return empty
- 31= {14,10,7} = {a1,a2,a4}
- = {24,7} ={a3,a4}
- ={14,9,1,7} {a1,a4,a5,a6}
- Total 2^n subsets


ϕ
ε
a1
a1
ε
ε
a2
a2
a2
a2
ε
ε
ε
ε
a3
a3
a3
a3
a3
a3
a3
a3
ε
ε
ε
ε
ε
ε
ε
ε
a4
a4
a4
a4
a4
a4
a4
a4
a4
a4
a4
a4
a1+a2+a3+a4
ε
ε
ε
a4
a4
a4
a4
a4
a4
a1+a3+a4
a2+a4

pruning

- Partial sum at a particular level
- A[1]=a1…an
- a[2]=a2…an
- a[i]=ai…an
- a[n]=an
- PS+ai ……+an <M

Branch and Bound

- In this method we search all state space methods before any other node can become the live node. The search for a new node can not begin until the current node is not fully explored, As in case of backtracking, bounding functions are used to avoid the generation of subtrees that don’t contain an answer node.

Travelling Salesman problem

- Using Dynamic programming Problem we get an $O(n^2 2^n)$ algorithm for travelling salesman problem.
- TSP is that given a graph where nodes are cities and edge weights are the cost of travelling from one city to another city. We have to find a minimum cost tour if one salesman starts from a city and can come back to the same city in minimum cost.

Initial cost matrix

	1	2	3	4	5	
1	∞	20	30	10	11	
2	15	∞	16	4	2	
3	3	5	∞	2	4	
4	19	6	18	∞	3	
5	16	4	7	16	∞	

we take C(i,i)= ∞ so as to avoid any path to the same city in our tour

	1	2	3	4	5	Row reduction
1	∞	20	30	10	11	10
2	15	∞	16	4	2	2
3	3	5	∞	2	4	2
4	19	6	18	∞	3	3
5	16	4	7	16	∞	4
						21

	1	2	3	4	5	
1	∞	10	20	0	1	
2	13	∞	14	2	0	
3	1	3	∞	0	2	
4	16	3	15	∞	0	
5	12	0	3	12	∞	
column	1	0	3	0	0	

	1	2	3	4	5	
1	∞	10	17	0	1	
2	12	∞	11	2	0	
3	0	3	∞	0	2	
4	15	3	12	∞	0	
5	11	0	0	12	∞	
column	1	0	3	0	0	4

Process to make the solution tree

- So we got the reduced cost matrix and all the tours in the original graph will have at least cost of 25. Now we are looking for a reduced cost matrix where every tour starts at one and ends at one. We have to make a solution space tree. Let R is the resulting matrix we got above then (R,S) means including node S in our tour. The reduced cost matrix for that can be obtained by Changing all entries in row i and column j to ∞. This prevents any more edges leaving vertex i or entering vertex j and also Set A(j,1) to ∞. This prevents the use of edge (j,1). Reduce all rows and columns in the resulting matrix except for rows and columns containing only ∞.

	1	2	3	4	5	
1	∞	10	17	0	1	
2	12	∞	11	2	0	
3	0	3	∞	0	2	
4	15	3	12	∞	0	
5	11	0	0	12	∞	

1,2	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	∞	∞	11	2	0	0
3	0	∞	∞	0	2	0
4	15	∞	12	∞	0	0
5	11	∞	0	12	∞	0
	0	0	0	0	0	Total cost =25+10=35

	1	2	3	4	5	
1	∞	10	17	0	1	
2	12	∞	11	2	0	
3	0	3	∞	0	2	
4	15	3	12	∞	0	
5	11	0	0	12	∞	

1,3	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	1	∞	∞	2	0	0
3	∞	3	∞	0	2	0
4	4	3	∞	∞	0	0
5	0	0	∞	12	∞	0
	11	0	0	0	0	Total cost =25+11+17=53

	1	2	3	4	5	
1	∞	10	17	0	1	
2	12	∞	11	2	0	
3	0	3	∞	0	2	
4	15	3	12	∞	0	
5	11	0	0	12	∞	

1,4	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	12	∞	11	∞	0	0
3	0	3	∞	∞	2	0
4	∞	3	12	∞	0	0
5	11	0	0	∞	∞	0
	0	0	0	0	0	Total cost =25+0=25

	1	2	3	4	5	
1	∞	10	17	0	1	
2	12	∞	11	2	0	
3	0	3	∞	0	2	
4	15	3	12	∞	0	
5	11	0	0	12	∞	

1,5	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	10	∞	9	0	∞	2
3	0	3	∞	0	∞	0
4	12	0	9	∞	∞	3
5	∞	0	0	12	∞	0
	0	0	0	0	0	Total cost =25+5+1=31

1,4	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	12	∞	11	∞	0	0
3	0	3	∞	∞	2	0
4	∞	3	12	∞	0	0
5	11	0	0	∞	∞	0
	0	0	0	0	0	Total cost =25+0=25

1,4,2	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	∞	∞	11	∞	0	0
3	0	∞	∞	∞	2	0
4	∞	∞	∞	∞	∞	0
5	11	∞	0	∞	∞	0
	0	0	0	0	0	Total cost =25+3=28

1,4	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	12	∞	11	∞	0	0
3	0	3	∞	∞	2	0
4	∞	3	12	∞	0	0
5	11	0	0	∞	∞	0
	0	0	0	0	0	Total cost =25+0=25

1,4,3	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	1	∞	∞	∞	0	0
3	∞	1	∞	∞	0	2
4	∞	∞	∞	∞	∞	0
5	0	0	∞	∞	∞	0
	11	0	0	0	0	Total cost =25+13+12=50

1,4	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	12	∞	11	∞	0	0
3	0	3	∞	∞	2	0
4	∞	3	12	∞	0	0
5	11	0	0	∞	∞	0
	0	0	0	0	0	Total cost =25+0=25

1,4,5	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	1	∞	0	∞	∞	11
3	0	3	∞	∞	∞	0
4	∞	∞	∞	∞	∞	0
5	∞	0	0	∞	∞	0
	0	0	0	0	0	Total cost =25+11+0=36

1,4,2	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	∞	∞	11	∞	0	0
3	0	∞	∞	∞	2	0
4	∞	∞	∞	∞	∞	0
5	11	∞	0	∞	∞	0
	0	0	0	0	0	Total cost =25+3=28

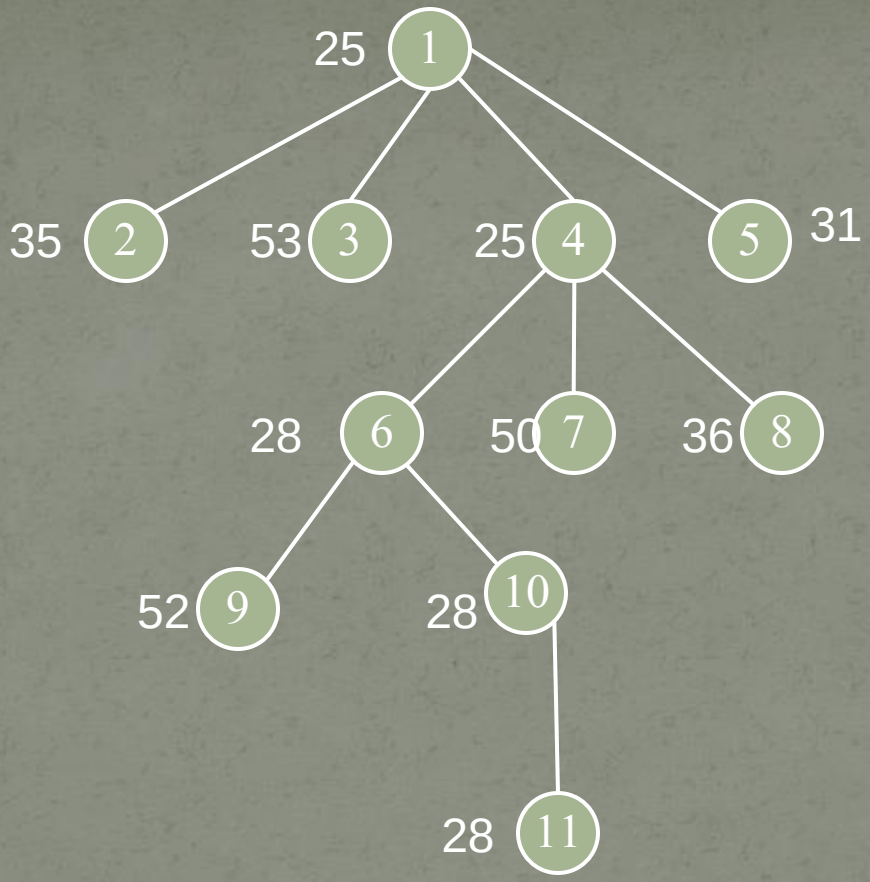
1,4,2,3	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	∞	∞	∞	∞	∞	0
3	∞	∞	∞	∞	0	2
4	∞	∞	∞	∞	∞	0
5	0	∞	∞	∞	∞	11
	0	0	0	0	0	Total cost =28+13+11=52

1,4,2	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	∞	∞	11	∞	0	0
3	0	∞	∞	∞	2	0
4	∞	∞	∞	∞	∞	0
5	11	∞	0	∞	∞	0
	0	0	0	0	0	Total cost =25+3=28

1,4,2,5	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	∞	∞	∞	∞	∞	0
3	0	∞	∞	∞	∞	0
4	∞	∞	∞	∞	∞	0
5	∞	∞	0	∞	∞	0
	0	0	0	0	0	Total cost =28+0=28

1,4,2,5	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	∞	∞	∞	∞	∞	0
3	0	∞	∞	∞	∞	0
4	∞	∞	∞	∞	∞	0
5	∞	∞	0	∞	∞	0
	0	0	0	0	0	Total cost =28+0=28

1,4,2,5,3	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	∞	∞	∞	∞	∞	0
3	∞	∞	∞	∞	∞	0
4	∞	∞	∞	∞	∞	0
5	∞	∞	∞	∞	∞	0
	0	0	0	0	0	Total cost =28+0=28


25
1
35
2
53
3
25
4
5
31
28
6
50
7
36
8
52
9
28
10
28
11

All pair Shortest path problem

- The output will be a matrix A such that A(i,j) is the length of the shortest path from I to j.
- One of the methods is to run dijkstra V times with each v ϵ V, but there should not be any negative weight edges in E & time will be O(n3)
- We will use an dynamic algorithm called Floyd-Warshall algorithm that allows negative edge weight but no negative weight cycles.

APSP Problem

- If k is an intermediate vertex on this shortest path from i to j then the subpaths from i to k and k to j must be shortest paths from I to k & k to j respectively, but we don't know k.

Using $d^{(k)}_{(i,j)}$ to represent the length of a shortest path from i to j going through {1,2,……K}

Floyd Warshall Algorithm

$D^{(0)}$ is original adjacency matrix

$D^{(n)}$ is the matrix to be computed

$D^{(k)}$ is $D^{(k)}[i,j] = d^{(k)}_{ij} = \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik}, d^{(k-1)}_{kj})$

The idea is to find all the vertices reachable using intermediate nodes in the range 1. ($D^{(1)}$), save the matrix & use it to find all vertices reachable using intermediate vertices in the range 1..2 ($D^{(2)}$) and so on upto $D^{(n)}$

Floyd Warshall Algorithm

n = rows[A]

$D^{(0)} = A$

For k = 1 to n do

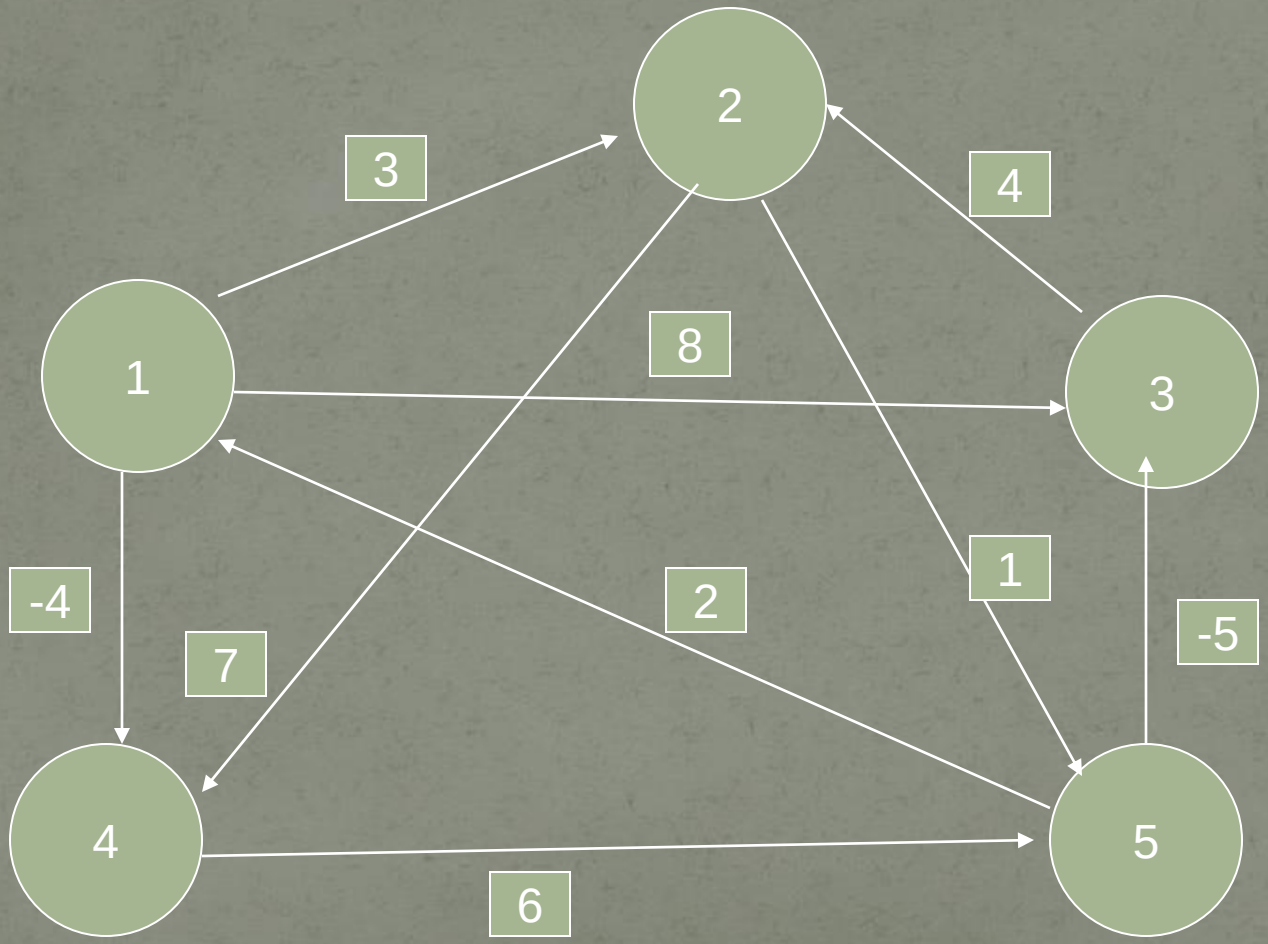
For i = 1 to n do

For j = 1 to n do

$d^{(k)}_{ij} = \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik}, d^{(k-1)}_{kj})$

Return $D^{(n)}$

Sample problem



D(1)

D1	1	2	3	4	**5**
1	0	3	8	-4	∞
2	∞	0	∞	7	1
3	∞	4	0	∞	∞
4	∞	∞	∞	0	6
5	2	∞	-5	∞	0

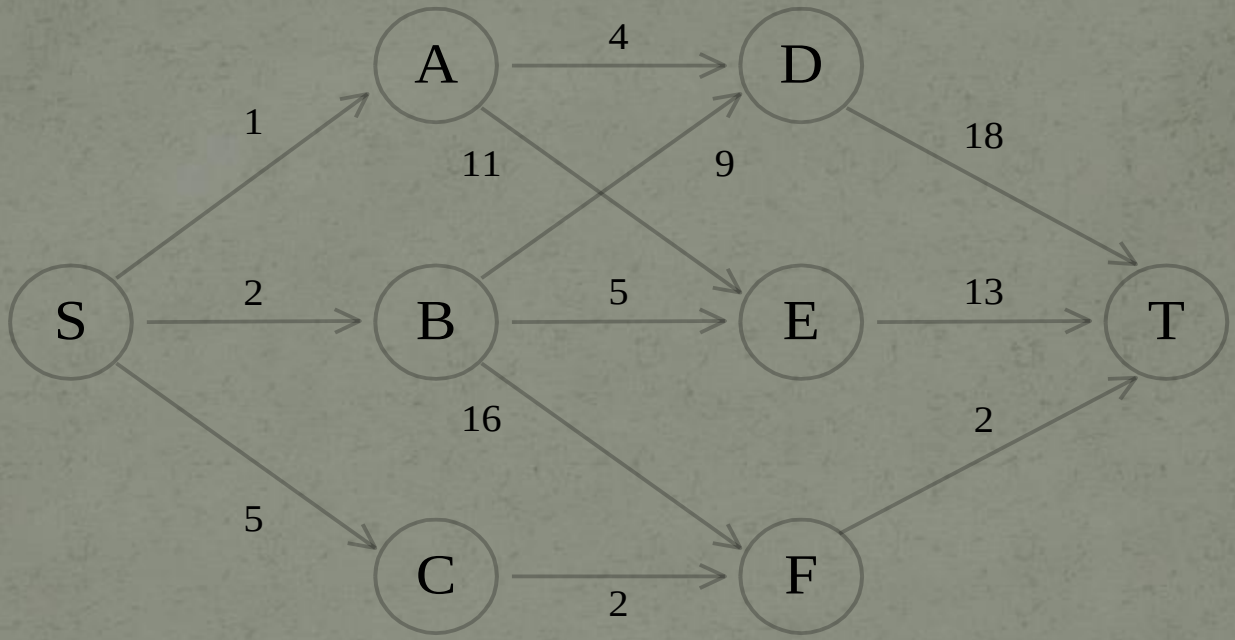
d2	1	2	3	4	5
1	0	3	8	-4	2
2	3	0	-4	7	1
3	∞	4	0	11	5
4	8	∞	1	0	6
5	2	-1	-5	-2	0

D4	1	2	3	4	5
1	0	1	-3	-4	2
2	3	0	-4	-1	1
3	7	4	0	3	5
4	8	5	1	0	6
5	2	-1	-5	2	0

Multi Stage Graph

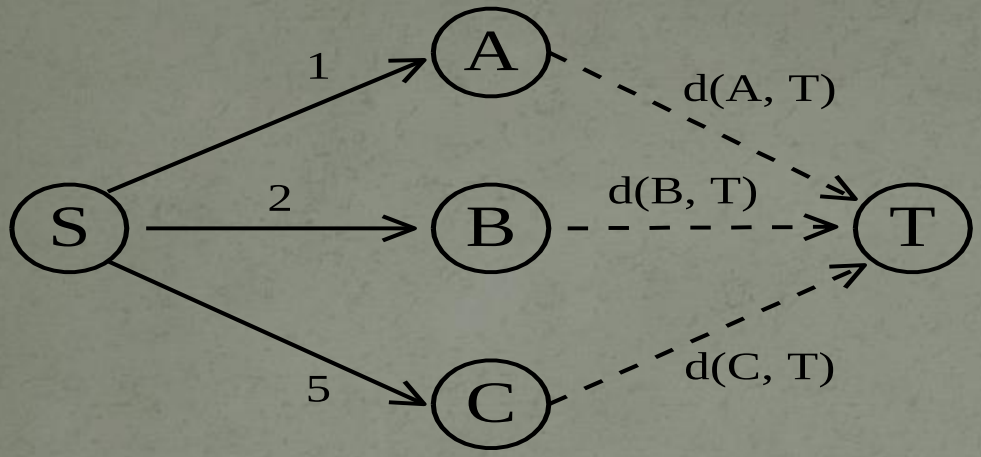
- These are the kind of problems which have various steps and one option to be selected in every step. Many problems can be converted to multi stage graph.
- If any problem can be converted to a multi stage graph then that can also be solved with dynamic programming.

Multi Stage Graph

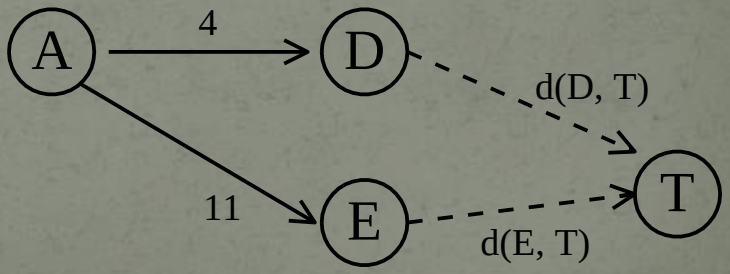


The greedy method can not be applied to this case:
(S, A, D, T) 1+4+18 = 23.
The real shortest path is: (S, C, F, T) 5+2+2 = 9

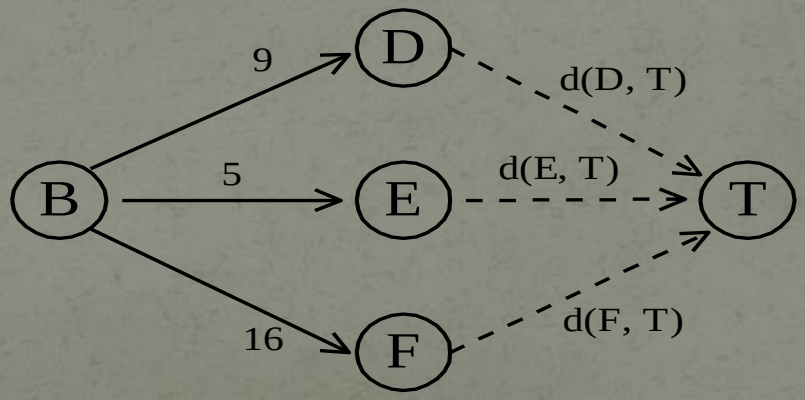
- d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}



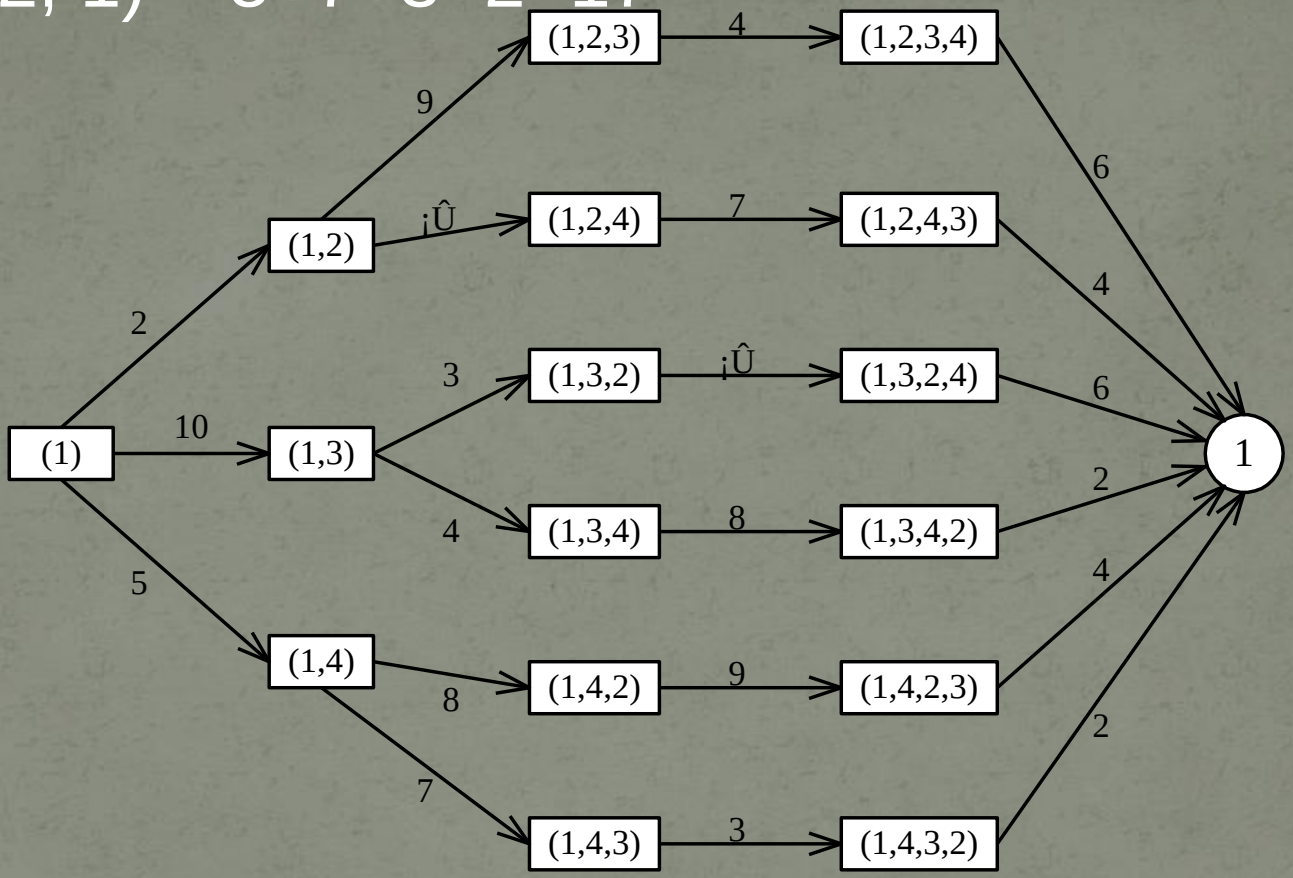
d(A,T) = min{4+d(D,T), 11+d(E,T)}
= min{4+18, 11+13} = 22



- d(B, T) = min{9+d(D, T), 5+d(E, T), 16+d(F, T)}
  = min{9+18, 5+13, 16+2} = 18.
- d(C, T) = min{ 2+d(F, T) } = 2+2 = 4
- d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}
  = min{1+22, 2+18, 5+4} = 9

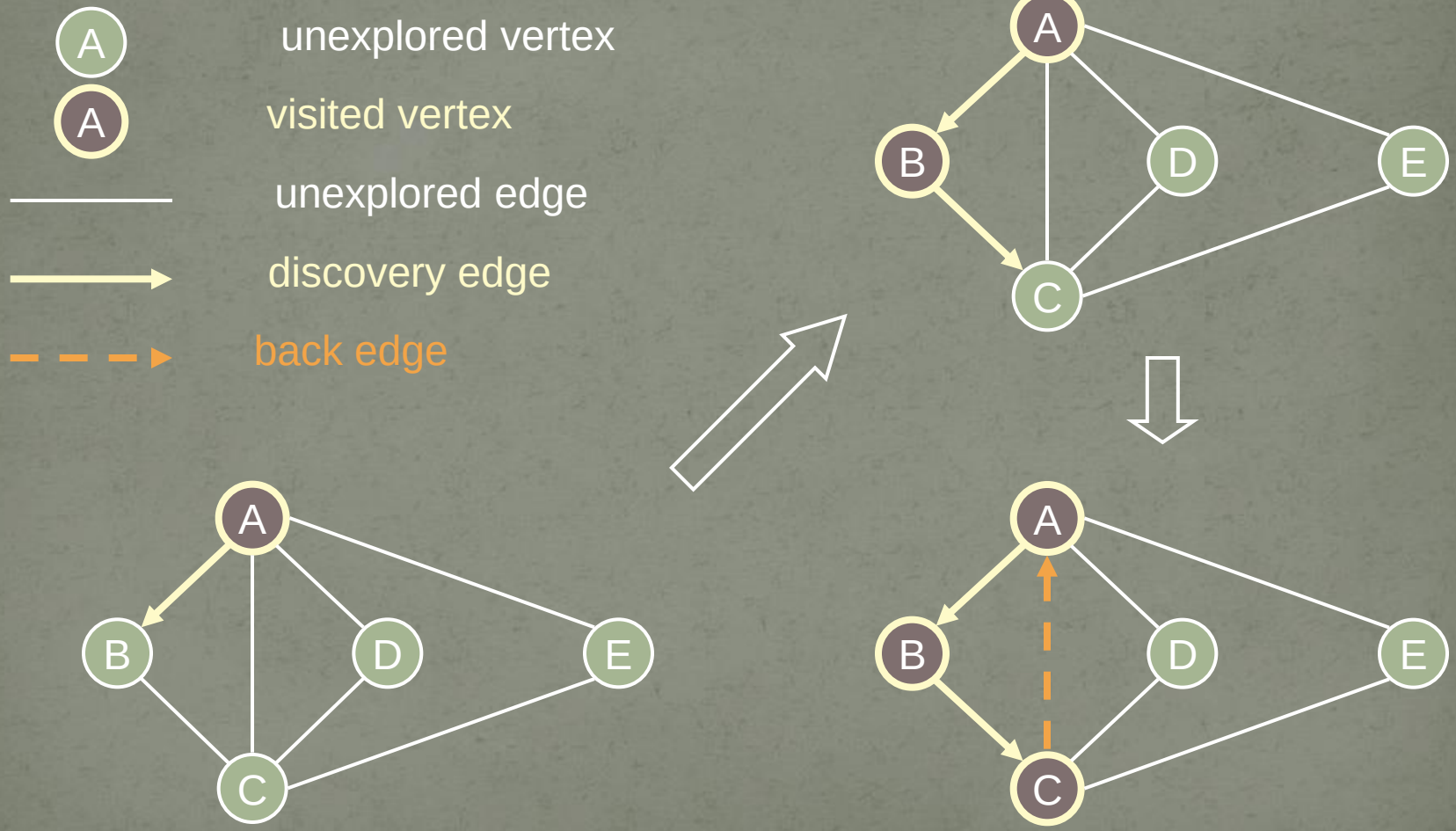


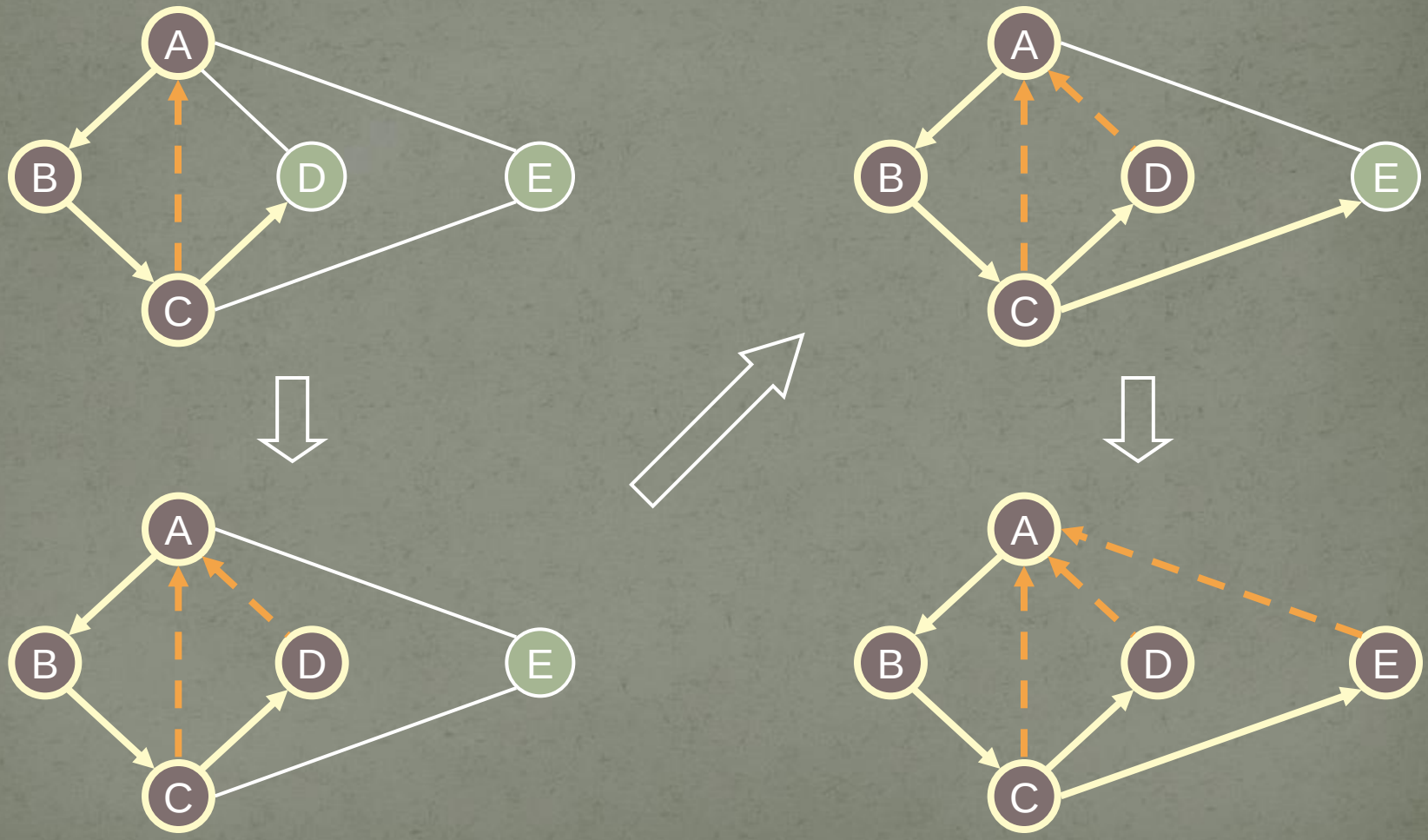
A multistage graph can describe all possible tours of directed graph. Find the shortest path (1, 4, 3, 2, 1) 5+7+3+2=17



DFS

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems


A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
back edge
A
B
D
E
C
A
B
D
E
C
A
B
D
E
C


A
B
D
E
C
A
B
D
E
C
A
B
D
E
C
A
B
D
E
C

Properties

- Property 1
- DFS(G, v) visits all the vertices and edges in the connected component of v
- Property 2
- The discovery edges labeled by DFS(G, v) form a spanning tree of the connected component of v
- Setting/getting a vertex/edge label takes O(1) time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in O(n + m) time provided the graph is represented by the adjacency list structure

BFS

- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- BFS on a graph with n vertices and m edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

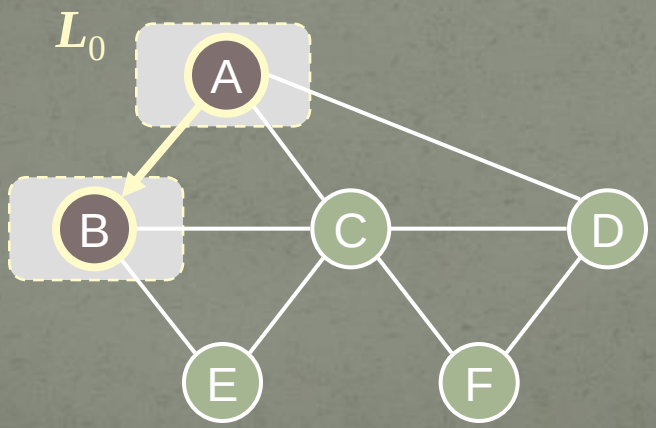
unexplored vertex

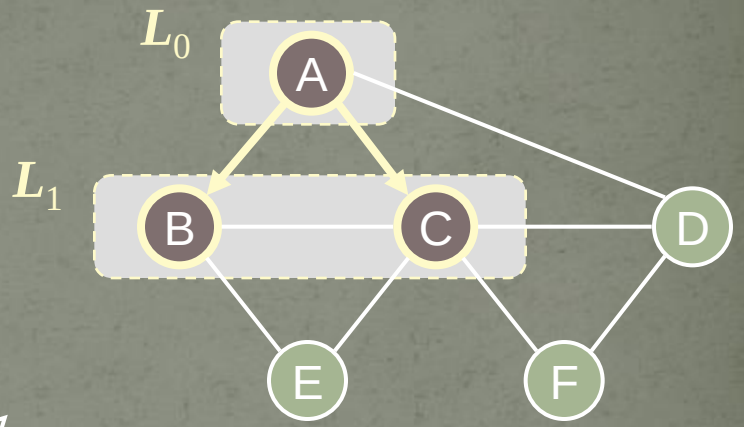
visited vertex

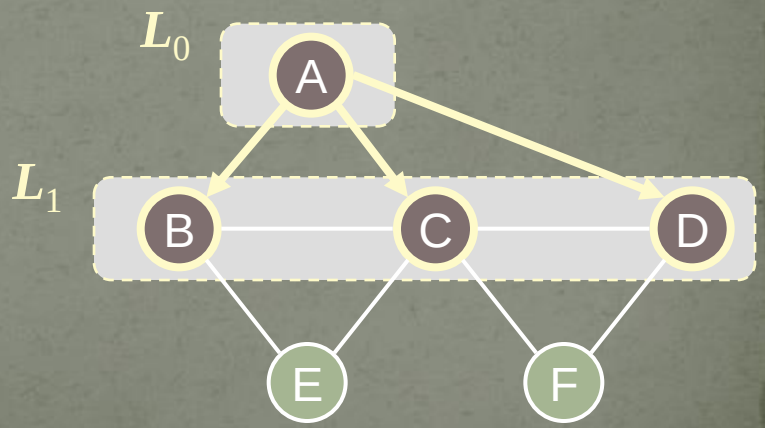
unexplored edge

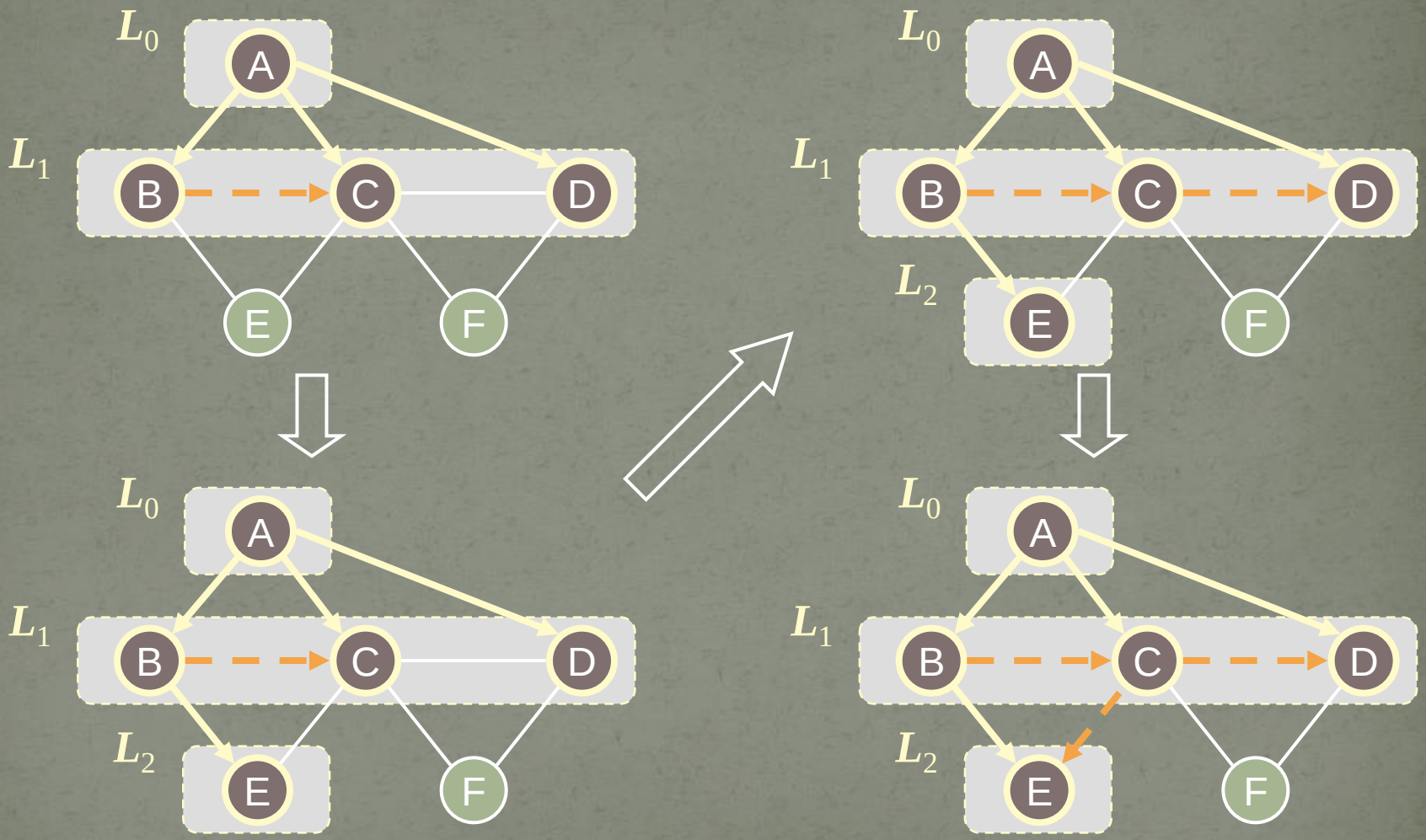
discovery edge

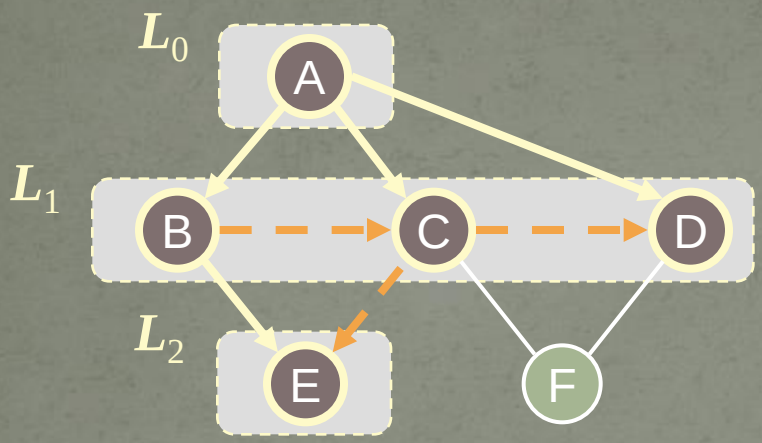
cross edge

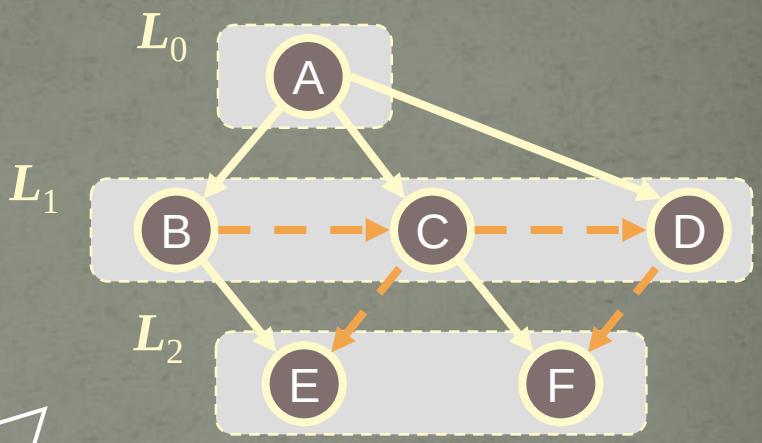


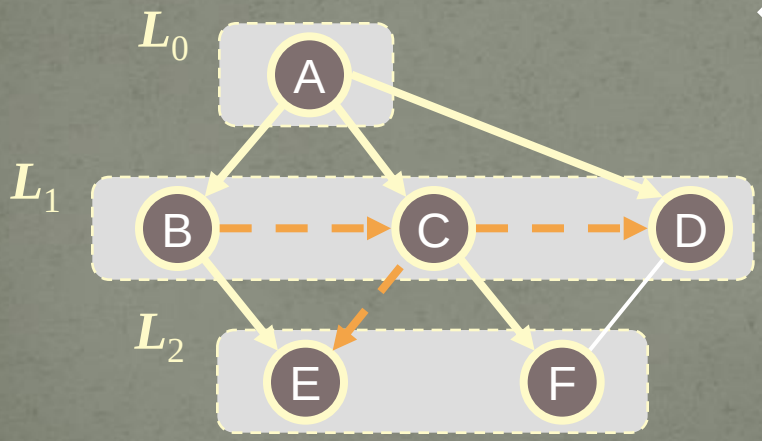





L_0
A
L_1
B
C
D
E
F
L_2


L_0
A
L_1
B
C
D
L_2
E
F


L_0
A
L_1
B
C
D
L_2
E
F


L_0
A
L_1
B
C
D
L_2
E
F

Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges() is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure

Strings

- A string is a sequence of characters
- Examples of strings:
  - C++ program
  - HTML document
  - DNA sequence
  - Digitized image
- An alphabet Σ is the set of possible characters for a family of strings
- Example of alphabets:
  - ASCII (used by C and C++)
  - Unicode (used by Java)
  - {0, 1}
  - {A, C, G, T}

String Matching

- Let P be a string of size m
  - A substring $P[i \,..\, j]$ of P is the subsequence of P consisting of the characters with ranks between i and j
  - A prefix of P is a substring of the type $P[0 \,..\, i]$
  - A suffix of P is a substring of the type $P[i \,..\, m-1]$
- Given strings T (text) and P (pattern), the pattern matching problem consists of finding a substring of T equal to P
- Applications:
  - Text editors
  - Search engines
  - Biological research

Brute force matching

- The brute-force pattern matching algorithm compares the pattern P with the text T for each possible shift of P relative to T, until either
  - a match is found, or
  - all placements of the pattern have been tried
- Brute-force pattern matching runs in time $O(nm)$
- Example of worst case:
  - $T = aaa \ldots ah$
  - $P = aaah$
  - may occur in images and DNA sequences
  - unlikely in English text

Brute force algorithm

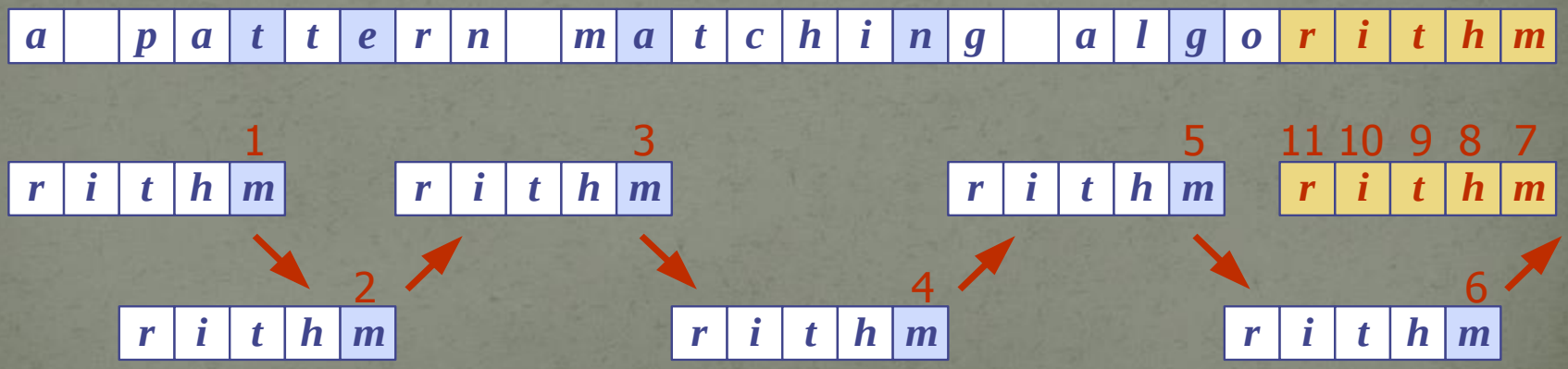
- Algorithm BruteForceMatch(T, P)
- Input text T of size n and pattern P of size m
- Output starting index of a substring of T equal to P or -1 if no such substring exists
  - for $i \leftarrow 0$ to n - m
  - { test shift i of the pattern }
  - $j \leftarrow 0$
  - while $j < m \wedge T[i + j] = P[j]$
  - $j \leftarrow j + 1$
  - if $j = m$
  - return i {match at i}
  - else
  - break while loop {mismatch}
  - return -1 {no match anywhere}

Boyer-Moore Heuristics

- The Boyer-Moore's pattern matching algorithm is based on two heuristics

  Looking-glass heuristic: Compare P with a subsequence of T moving backwards

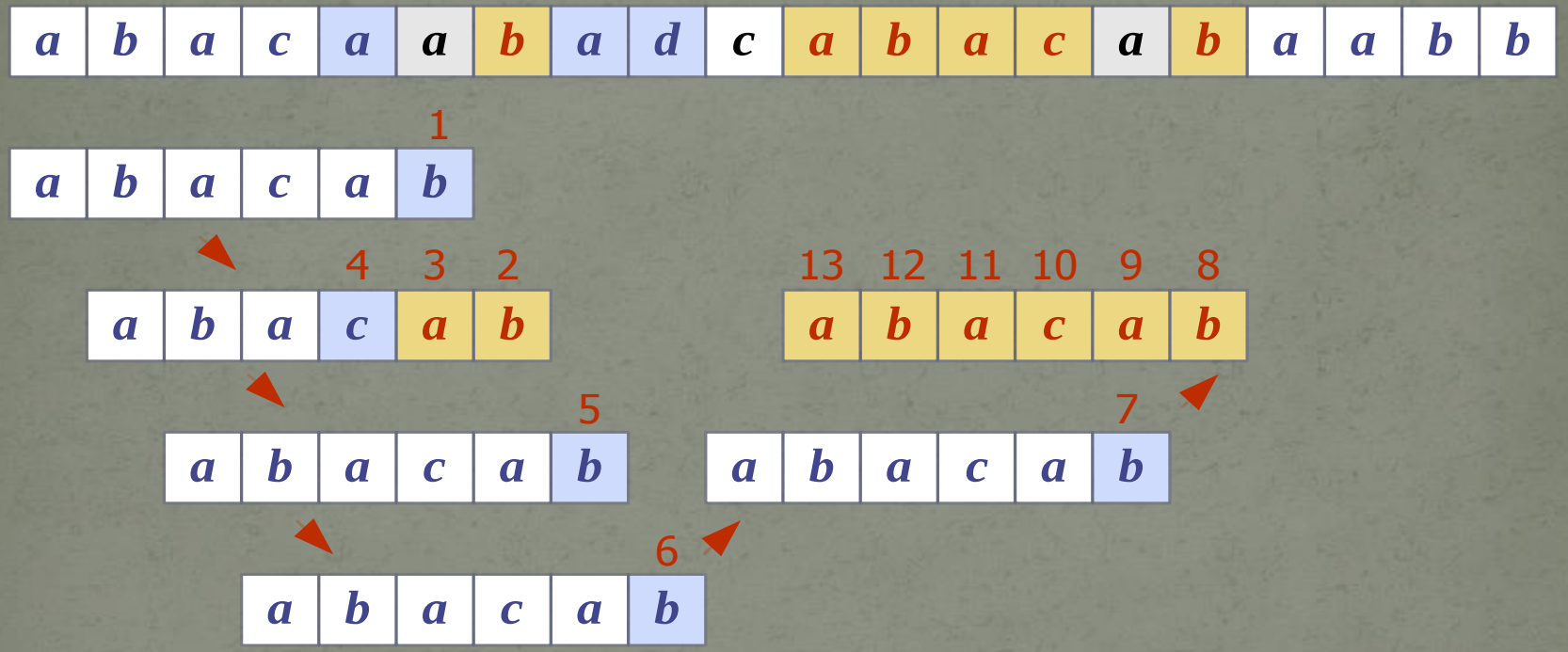
  Character-jump heuristic: When a mismatch occurs at $T[i] = c$
  - If P contains c, shift P to align the last occurrence of c in P with $T[i]$
  - Else, shift P to align $P[0]$ with $T[i+1]$
- Example



Algorithm BM

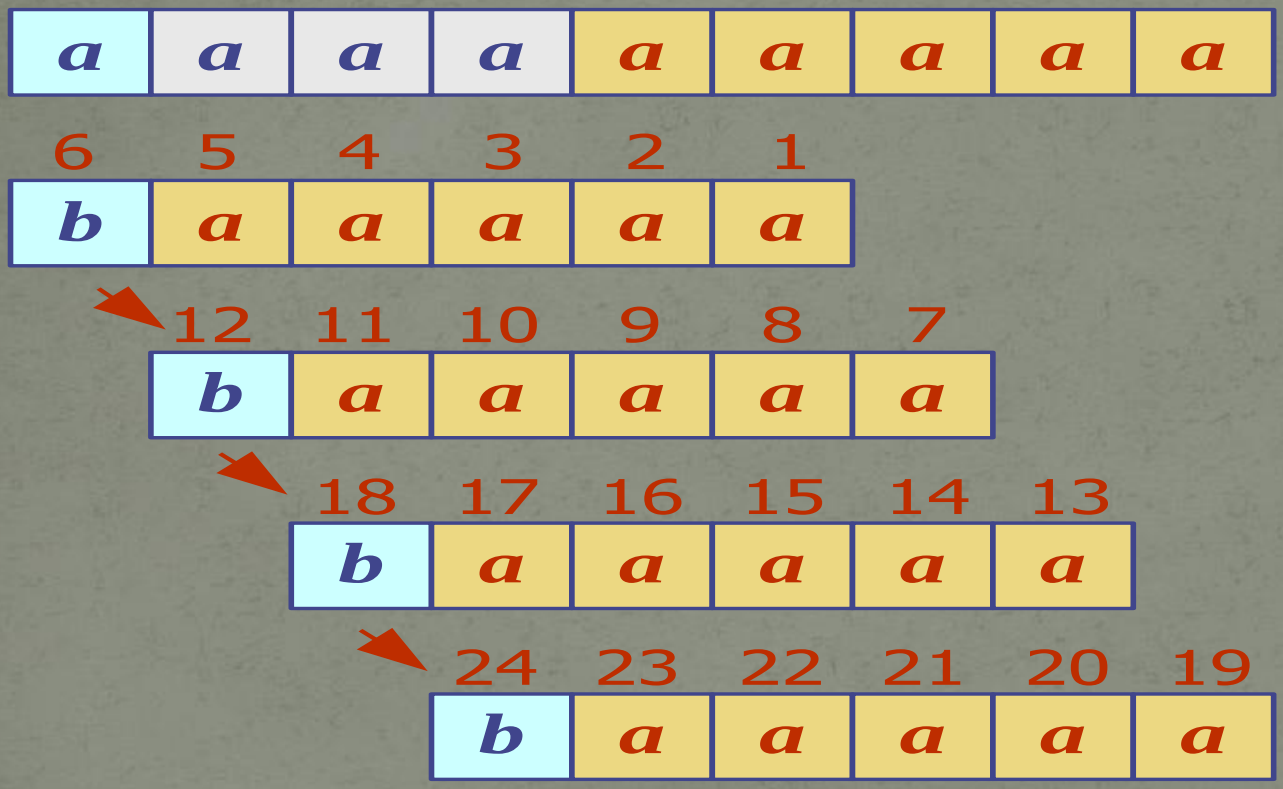
```
Algorithm BoyerMooreMatch(T, P, S)
        L ← lastOccurenceFunction(P, S )
        i ← m - 1
        j ← m - 1
        repeat
        if T[i] = P[j]
                if  j = 0
                        return  i  { match at i }
                else
                        i ← i - 1
                        j ← j - 1
        else
                { character-jump }
                l ← L[T[i]]
                i ← i + m – min(j, 1 + l)
                j ← m - 1
    until  i > n - 1
    return  -1 { no match }
```

Example


a b a c a a b a d c a b a c a b a a b b
1
a b a c a b
4 3 2
a b a c a b
13 12 11 10 9 8
a b a c a b
5
a b a c a b
7
a b a c a b
6
a b a c a b

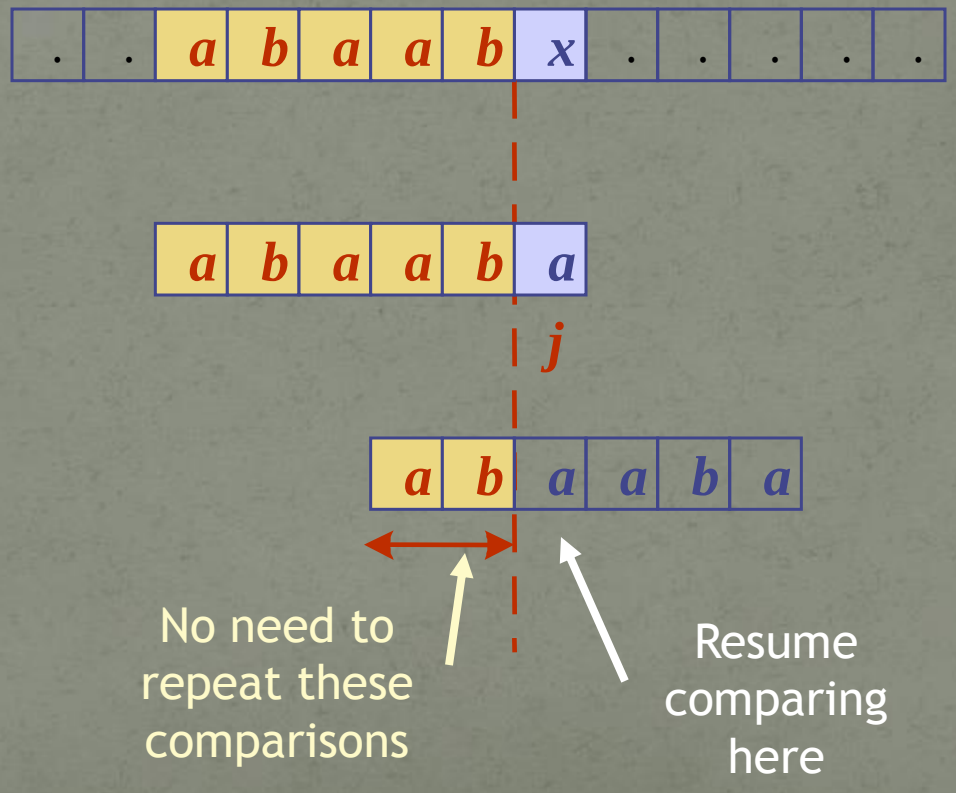
Analysis

- Boyer-Moore's algorithm runs in time $O(nm + s)$
- Example of worst case:
  - $T = aaa \ldots a$
  - $P = baaa$
- The worst case may occur in images and DNA sequences but is unlikely in English text
- Boyer-Moore's algorithm is significantly faster than the brute-force algorithm on English text


a a a a a a a a a
6 5 4 3 2 1
b a a a a a
12 11 10 9 8 7
b a a a a a
18 17 16 15 14 13
b a a a a a
24 23 22 21 20 19
b a a a a a

KMP Algorithm

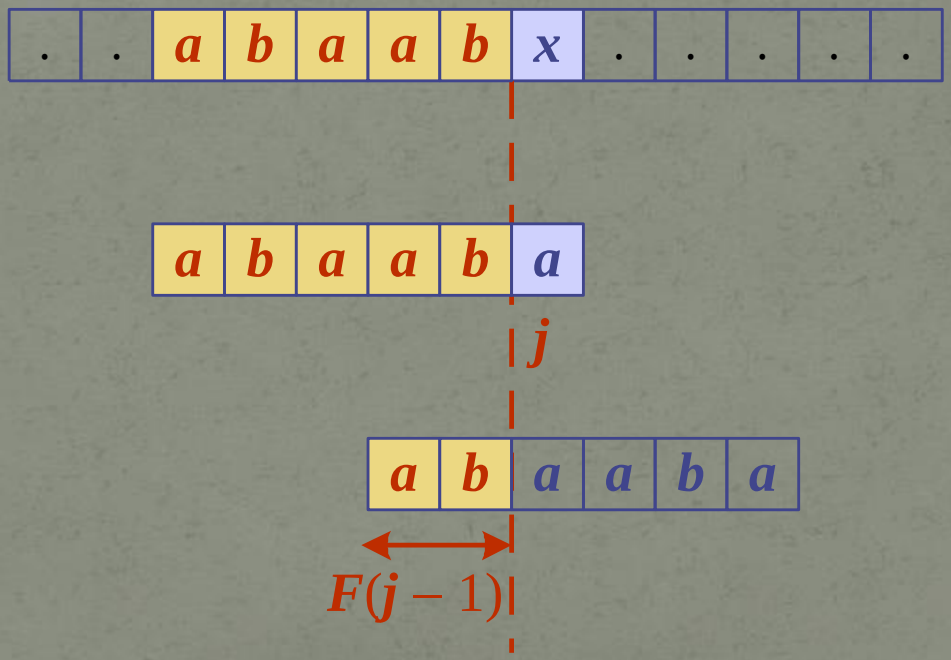
- Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$


. . a b a a b x . . . . .
a b a a b a
j
a b a a b a
No need to repeat these comparisons
Resume comparing here

KMP Algorithm

- Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself
- The **failure function** $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F(j-1)$

j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3

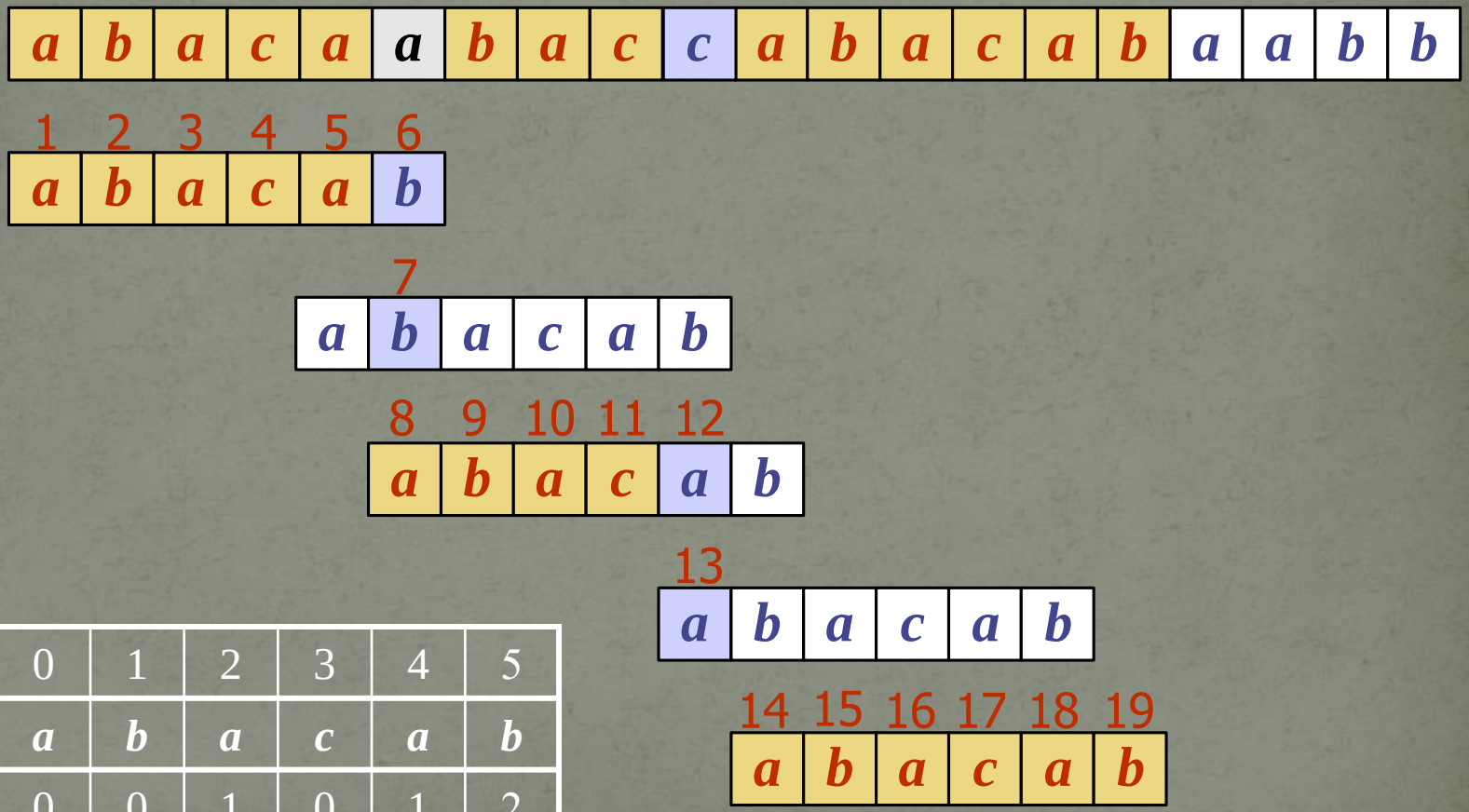


Failure function

- The failure function can be represented by an array and can be computed in $O(m)$ time
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j-1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop
- Thus, KMP's algorithm runs in optimal time $O(m+n)$

Algorithm KMP

```
Algorithm KMPMatch(T, P)
        F ← failureFunction(P)
        i ← 0
        j ← 0
        while i < n
        if T[i] = P[j]
                if  j = m - 1
                        return  i - j { match }
                else
                        i ← i + 1
                        j ← j + 1
        else
                if  j > 0
                        j ← F[j - 1]
                else
                        i ← i + 1
    return  -1 { no match }
```



j	0	1	2	3	4	5
$P[j]$	a	b	a	c	a	b
$F(j)$	0	0	1	0	1	2

genetic Programming

- Genetic programming (GP) is an evolutionary algorithm based methodology inspired by biological evolution to find computer programs that perform a user-defined task. It is a specialization of genetic algorithms where each individual is a computer program. Therefore it is a machine learning technique used to optimize a population of computer programs according to a fitness landscape determined by a program's ability to perform a given computational task.

steps

1. Generate random “programs”
2. Evaluate programs using training data
3. Modify population of programs using cross-over and mutation
4. If a good program is found, finish, else go to 2

terminology

- Gene : A section of genetic code that represents a single parameter to a solution
- Allele : The value stored in a gene
- Chromosome : The collection of genes sufficient to completely describe a possible solution to a program
- Genome : all the chromosomes of an individual organism
- genotype : the values of a genome
- problem space: the area of all possible problem instances
- Solution space: similar to the problem space but defined by the solution parameters

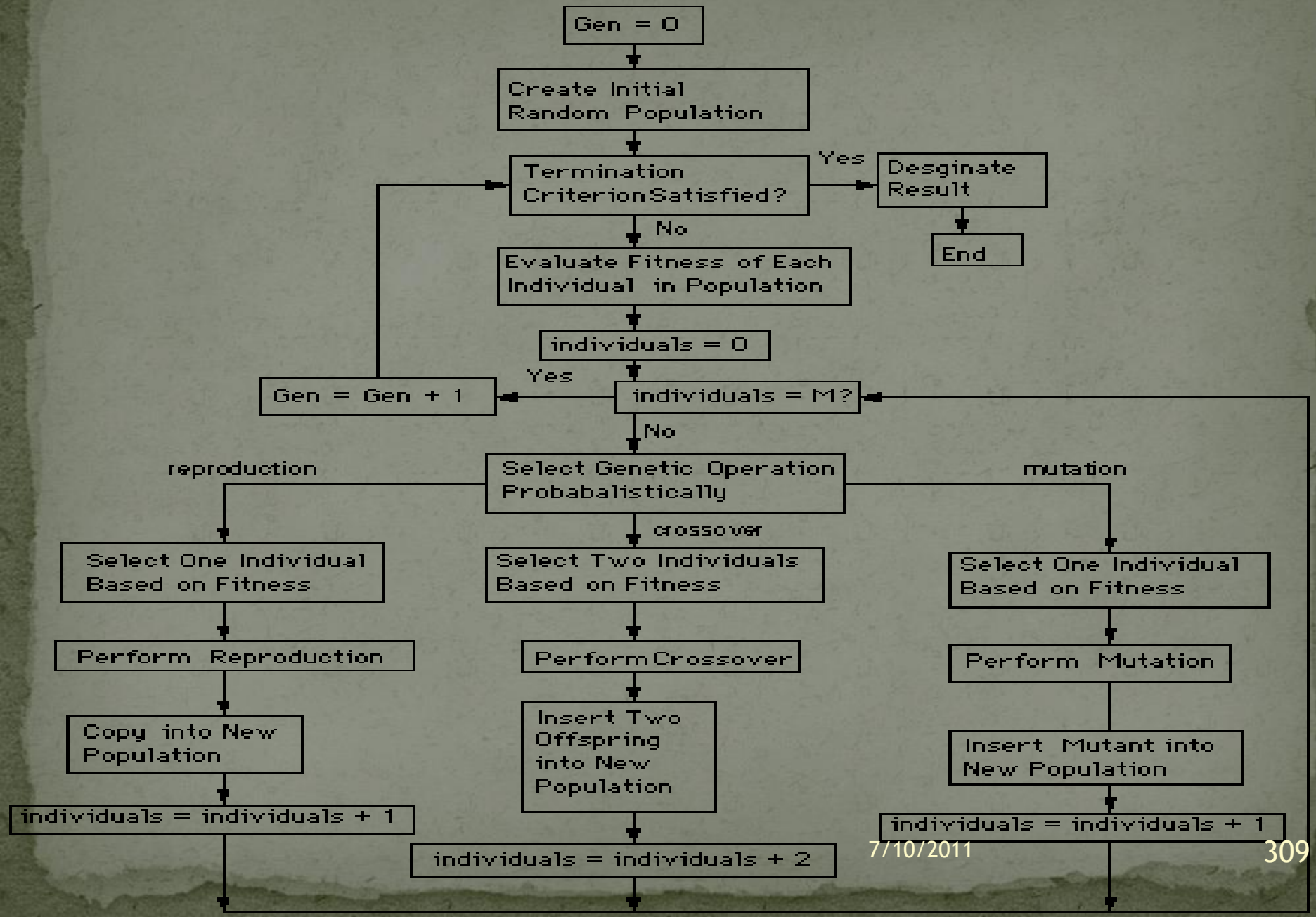
mapping

- Individual (Chromosome) = Possible solution
- Population = A collection of possible solutions
- Fitness = Goodness of solutions
- Selection (Reproduction) = Survival of the fittest
- Crossover = Recombination of partial solutions
- Mutation = Alteration of an existing solution

operators

- **Reproduction:** make copies of chromosome (the fitter the chromosome, the more copies)
- 10000100• 10000100 10000100
- **Crossover:** exchange subparts of two chromosomes
- 100| 00100 10011111
- 111|11111 11100100
- **Mutation:** randomly flip some bits
- 00000100 00000000

Flowchart for Genetic Programming



Fitness function

- Measure the performance of the candidate programs on the training data
- For example: Fitness is the number of output values generated by the program which come within 20% of the correct values.

Genetic Programming

- Robot motion
- Image recognition
- Facility layout problems
- Data mining
- bio-computing

Working with real numbers

- A computer is an approximate device when working with real numbers.
- To begin with, the real number is represented only approximately. Calculations on real numbers have truncation and roundoff errors which creep in due to finite precision.
- The rules of arithmetic – distributivity, associativity, etc may not be exactly met. There are specific properties which finite precision implementations like those following the IEEE-754 and IEEE-854 standard have to meet.
- Two different formulae which look the same in exact arithmetic can have considerable different error properties when in finite precision arithmetic.

root cause

- Decimal Numbers cannot be exactly represented in the current floating-point standard implemented on most
- machines (IEEE754)
- •0.1
- •*We cannot enter this value exactly on a Form on a page!*
- •*It is taken to be 0.099999999999999500399638918679556809365749359130859375*
- •*Will cause errors, even if all subsequent calculations exact!*

Comparing Floating point numbers

- Do not rely on exact comparisons between two floating point variables, since there will typically be errors in
- calculating the value of the variables, making the comparison inexact.
- If EPSILON is the tolerance,
- to compare two floating point values x and y, do
- if (abs(x - y) <= EPSILON) then
- x=y
- else
- x!=y
- end

Business applications

- **The stock market trades 10,000's Crores every day!**
- **The total funds in should match the total funds + fee out very accurately (1 part in 10^{16} or better),**
- IEEE754 accuracy, with 15/16 decimal digits may not be sufficient for certain classes of financial calculations, and we may have to use more precise arithmetic – e.g. variants of IEEE754 and/or IEEE854

Banks and financial Applications

- Large Corporate Deposit Rs 10,000 Crores
- – 5% Interest P/A, Rs 500 Crores

Even 0.01% Inaccuracy in calculation

- – Loss of Rs 5 Lakhs to Corporation (Can be software hacker's gain!)

Example: Sales tax calculation

- *– 5% tax on a short ATM transaction priced at US $0.70, round to cents*
- *– Binary, double precision is 0.7349999999999999- which is* rounded to $0.73
- – Exact Decimal calculation: $0.735 => Round upto $0.74!
- – 1.37% error, completely unacceptable!

Solutions

- Financial Calculations require some of the highest precision computations. This is because of the huge volumes of money which modern financial institutions like banks/ stock exchanges handle. Even small relative errors in calculation can lead to large errors, with possible legal consequences. Again, small errors in calculating parameters like interest rates, etc. can have major impact on a bank's balance sheet.
- Currently special hardware at 30+ digit accuracy is being developed for demanding financial applications. (e.g. IBM is implementing a FP ALU with large number of digits).
- Binary Representations are not very accurate enough for financial calculations. Even the small errors which IEEE-754 makes are significant for demanding financial applications

issues

- A large number of classical graph algorithms depend on the associativity and commutativity properties of the metric used. In simple terms, the cost or the total length should be the same, no matter in what order you add the edges in.
- Unfortunately, with finite precision arithmetic, these properties DO NOT exactly hold, and the algorithms assuming them cannot guarantee the best solution. They will typically get a good close-to-optimal solution. For some applications in finance (e.g. arbitrage, multicurrency
- large volume electronic stock exchanges), with the highest accuracy requirements, this may not be good enough, and modifications/alternatives have to be devised.

issues

- An example of these issues in in balancing books in finance (flow conservation), using inaccurate arithmetic.
- Detecting large violations of flow conservation is simply performed by comparing total inflow versus total outflow at all nodes.
- If the money outflow is less than the inflow, there is either an account not accounted for, or electronic-money is vanishing (which is not acceptable in any entity other than the Central bank)
- If the money outflow is greater than the inflow, then some unaccounted capital reserve is being depleted, or else money is being generated (not acceptable in any entity other than the Central Bank)
- Detecting small violations is much more difficult. Basically, the output flow may not be equal to the input flow exactly, and may be slightly less than, greater than, or equal to the input.
- The difference may also change with changes in the input. Detecting systematic violations of flow conservation requires statistical tests in general

Power conscious algos

- Power is an important criteria to be considered while designing embedded systems, sensor networks, etc
- • A majority of the digital logic IC's today are made from CMOS technology.
- This technology dissipates energy primarily while transitioning from one state to another (bit flip), and very little energy is required to just maintain the state.
- As such, if for an algorithm, the changes of state of the registers and/or memory units are reduced, power efficiency is improved.

Linear Indexing

- Address (# of bit changes)
- 000
- 001 (1)
- 010 (2)
- 011 (1)
- 100 (3)
- 101 (1)
- 110 (2)
- 111 (1)

Gray Code Indexing

- 000
- 001 (1)
- 011 (1)
- 010 (1)
- 110 (1)
- 111 (1)
- 101 (1)
- 100 (1)

Resiliency

- Throughout the IT industry, fault tolerance and data resiliency is becoming increasingly important. Some of the dimensions of this very important area are:
- Systems have to be up 24 hours a day, 7 days a week, 365 days a year (ideally). Many banks, financial institutions, e-commerce sites, sports/news sites, etc require this.
- Clustering/hot-standby systems are some of the techniques for this purpose.
- Data is precious. It has to be accurately maintained, irrespective of hardware/software failures of the IT infrastructure. Accounts in a bank have to be maintained accurately for many years, maybe decades.
- Various forms of RAID (redundant arrays of independent disks) arrays, replication, mirroring to remote sites, etc are used.
- The techniques used to build fault tolerant systems typically introduce communication and/or synchronization, which can have a major impact on algorithm performance (it can be the dominant effect).

backup and replication

- Two copies of data are maintained, in physically diverse locations. Typically, due to the overhead of communication and maintaining synchronization between these copies, the locations can be at most 100-200 Km from each other. This system is resilient to outages at one site, but not to area wide outages, e.g. a flood affecting the whole city

Resiliency

- For more resiliency, data has to be copied to locations 1000's of Km distant. Maintaining exact synchronism of the copies is generally not cost-effective in this situation. The remote copy is typically a few transactions behind the master and the local copy (some transactions have been committed and
- written to the master and the local copy, but not the remote copy). Recovery of data from the remote site may not be complete. Minimizing communication bandwidth to the remote copy is a major/dominant algorithmic issue, here, since each remote access is very expensive.
- **Conclusion:**
- Algorithms designed resiliency have to work with and keep multiple copies of data. Consistency amongst these copies is an issue.
- The analysis of algorithms taking into account data resiliency due to distribution of the data at multiple places has to take communication and/or synchronization time into account.

Concurrency

- Two or more users access a database concurrently
- Problems associated with concurrent execution:
- Lost update
- Dirty read
- Non repeatable read
- Phantom records

Lost Update

	Murugan's Deposit	Balance	Narsimhulu's Deposit
5.22	Read Balance(1500)	1500	
5.23	Balance=1500+2000		
5.24			Read Balance(1500)
5.25	Write New Balance(3500)	3500	
5.26	Commit		
5.27			Balance=1500+1800
5.28		3300	Write new balance(3300)
5.29			Commit

Dirty Read

	Jahfar's Deposit	Balance	Riji's Deposit
5.22	Read Balance(1500)	1500	
5.23	Balance=1500+2000		
5.24	Write New Balance(3500)	3500	
5.25			Read Balance(1500)
5.26	Rollback		
5.27			Balance=1500+3500
5.28		5300	Write new balance(5300)
5.29			Commit

Incorrect Summary

	Shailesh's Transfer	Balance	
5.22	Read Shailesh Balance(1500)	1500	Sum=0
5.23	Balance=1500-500		Read Shailesh's Balance
5.24	Write New Balance(1000)	1000	
5.25			Sum=Sum+ Balance(1500)
5.26	Read Sidharth Balance(1500)	1500	
5.27	Balance=1500+500		
5.28	Write new balance(2000)	2000	
5.29	Commit	2000	
5.30			Read Sidharth's Balance(2000)
			Sum=sum+balance(3500)
		3000	Write sum(3500)
			Commit

Phantom Record

	Jahfar's Deposit	Balance	Riji's Deposit
5.22			
5.23			Read total number of accounts in the bank
5.24			
5.25	Create account for Arjan		
5.26	Create Account for Nisarg		
5.27	Create Account for Jainul		
5.28	Commit		
5.29			Write Total
5.30			Commit

Solution

- To make every transaction follow each other.
- Achieved by setting following rules on transactions:
- If any row is being modified, then do not allow any other transaction either to read or write that row until the first transaction completes.
- If a transaction is reading a particular row, prevent other transactions from making any changes to that row until the first transaction completes.
- If a transaction is reading some data, do not allow any other transaction to insert new rows into the same table until the first transaction completes. This will avoid problems like phantom records.

Shared Intent Exclusive(SIX) locking

	S	X	IS	IX	SIX
S	Y	N	Y	N	N
X	N	N	N	N	N
IS	Y	N	Y	Y	Y
IX	N	N	Y	Y	N
SIX	N	N	Y	N	N

Deadlock

	Transaction ATM update	Transaction Loan update
5.22	Lock ATM Data table	Lock Loan data table
5.23	Update ATM details	Update loan details
5.24	Try lock on loan table	Try lock on atm table
5.25	Wait for lock	Wait for lock
5.26	Wait for lock	Wait for lock
5.27	Wait for lock	Wait for lock
5.28	Wait for lock	Wait for lock
5.29	Wait for lock	Wait for lock
5.30	Wait for lock	Wait for lock

Timestamping

- X W_X=3.40 p.m Rx= 3.20 p.m
- Now transaction Ti starts at 3.35 p.m who's action to be performed is Read(x). Ti wanted to read x at 3.35. at that time the value could have been 20 (assume) so Ti actually wanted this value 20. But, Ti could not get a chance at that time. In the meantime, another transaction Tj has updated the value of X at 3.40 p.m . When Ti is getting the chance, it is already 3.42 p.m .. If Ti reads the value of x, it is going to be the new value as updated by t_j and not the one ti actually wanted to read. So, there is no point in performing this read operation now. Ti is rejected and has to try again.
- Let us say Ti issues Write(Q)
- **Case 1: if(Ts(T_i) < Rx(x) it means ti is trying to update x whereas some other trans has already read**
- the value of x. so the updation done by ti is not required. So ti is rejected
- **Case 2: if Ts(T_i) < Wx(x) then it means before Ti could update x, some other transaction has updated**
- x with a latest value. So ti is rejected
- Case 3 : else ti is executed

P,NP, NP-Hard and NP-Complete

- We can categorize the problem space into two parts
- Solvable Problems
- Unsolvable problems

Halting Problem

- *Given a description of a program and a finite input, decide whether the program finishes running in a finite time or will run forever, given that input*
- The halting problem is famous because it was one of the first problems proved undecidable, which means there is no computer program capable of correctly answering the question for all possible inputs

Post Correspondence Problem

- U1 U2 U3 U4
- aba bbb aab bb

- V1 v2 v3 v4
- a aaa abab babba

- A solution to this problem would be the sequence 1, 4, 3, 1 because
- $u_1u_4u_3u_1 = aba + bb + aab + aba = ababbaababa = a + babba + abab + a = v_1v_4v_3v_1$
- *The problem is to find whether such a solution exists or not*

Moving towards solution

- From not solvable we come to the concept of how well we can solve the problems
- logn- logaritihmic time
- n- linear time
- nlogn – near linear time
- n^2 – square time
- n^3 cubic time
- n^4 --- n^k - higher polynomial or super polynomial

P Class of Problems

- All those algorithms that can be solved in polynomial time with worst case running time as $O(n^k)$. So these problems are regarded as tractable.
- A problem that can be solved polynomially in one model can also be solved polynomially in other model.

Determinism

- If the result of every operation is uniquely defined then we call that algorithm as deterministic algorithm.
- If we remove this notion and we say that the output of an operation is not unique but limited to a set of possibilities
- In theoretical computer science, a non-deterministic Turing machine (NTM) is a Turing machine (TM) in which state transitions are not uniquely defined.

Non determinism

- A Non deterministic algorithm for searching an element x in a given set of elements A[1..n]
- 1. j= choice(1,n)
- 2. if A[j]=x then write (j); Success
- 3. else write(0); Failure

Non determinsitic sorting

- Algorithm Nsort
- For i=1 to n do B[i]=0;
- for i=1to n do
- { j= choice(1,n);
- if b[j]<>0 then failure();
  - B[j] = a[i]
  - }
  - for i= 1 to n-1 do
  - if B[i] > B[i+1] then failure;
  - success();

Non-determininstic Polynomial (NP)

- It is the set of all problems that can be solved in polynomial time with non-deterministic concept.
- However this concept only exists in theory. There are no implementations what so ever.

Decision Problems

- A decision problem is that problem which can have only two options as its solution space i.e. yes or no.
- Whether x is a prime number is also a decision problem that will have answer yes or no.
- For a Travelling salesperson problem the question that whether a tour of cost < k exists is a decision problem.

Guess & verify

- So for many problems it will be like guess & verify situation.
- We will guess a particular solution and then verify that whether this is the solution (yes) or not (no)
- If this guessing and verifying can be done in polynomial time with the help of a non deterministic concept then we say that the problem is in NP

Satisfiability Problem

- Satisfiability is the problem of determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. Equally important is to determine whether no such assignments exist, which would imply that the function expressed by the formula is identically FALSE for all possible variable assignments. In this latter case, we would say that the function is unsatisfiable; otherwise it is satisfiable. To emphasize the binary nature of this problem, it is frequently referred to as Boolean or propositional satisfiability.

Satisfiability Problem

- if $f(x_1,x_2,\ldots\ldots x_n)$ = True
- x_i = 0 or 1
- we have to find $x_1,x_2,x_3\ldots x_n$
- such that $f(b_1,b_2,b_3\ldots\ldots b_n)$=1 where b1= 0 or 1,b2= 0 or 1…….. bn = 0 or 1
- 2^n different permutations, someone out of that can give answer as 1 (true)
- guess
- ?f(1,0,1,0,0,1 ……..,1,0) = 1
- verify (evaluate the function)
- ability to guess and verify in polynomial time

COOK theorem

- The complexity class P is contained in NP but the vice versa has not been proved and proving whether p= NP remains the biggest research question.
- Cook Formulated a question that is there any single problem in NP such that if we showed it to be in P, then that would imply P=NP

reducibility

- Given an instance x of a problem A, use a polynomial time reduction algorithm to transform it to instance y of problem B.
- Run the polynomial time algorithm for B on instance y
- use the answer for y as the answer for x.

NP Hard

- This problem A can be decision problem for some problem or a decision version for optimization problem. It may be or may not be part of NP class of problems. We call such problems as NP hard problems if they can be reduced to another problem B in polynomial time and then then that problem B can be solved in polynomial time and then that result of B can be used for giving solution of A in polynomial time.

NP complete

- if Problem is NP-hard and problem belongs to NP then it is NP-complete.
- Problems are designated "NP-complete" if their solutions can be quickly checked for correctness, and if the same solving algorithm used can solve all other NP problems. They are the most difficult problems in NP in the sense that a deterministic, polynomial-time solution to any NP-complete problem would provide a solution to every other problem in NP

Max Clique problems

- Given a graph , what is the max clique size of the graph.
- Decision problem: Does G have a clique size < K (yes or No)

Reduction

- In Satisfiability problem we have a specific problem called 3 satisfiability problem. It uses conjuctive normal form(3 CNF)
- Let us try to convert 3SAT CNF→ Clique Problem
- (I,P) → (I',G)

- $f=(x1+x2+\bar{x3})$ and $(\bar{x1}+\bar{x2}+x3)$ and $(x2+\bar{x3})$

- Take x1=1,x2=0 and x3=1

proof

- Each Column corresponds to each clause. (x,y) are not connected if they are in the same column or if

- $x = \bar{y}$

- G has a clique of size k iff this is satisfiable
- So we are going to provide edges except inconsistent assignments
- Function f will be satisfiable iff all clauses are one and all clauses will be one iff one of the constituents is one.
- Assume any of them to be one and then take corresponding elements in the graph, they will form a clique because they are in different columns and do not complement each other.

Final judgement

- So we are able to correlate one problem which is NP hard with the another problem which is also NP hard and so we can say that if one of them can be solved in polynomial time then another can also be solved in polynomial time. Similarly it can be proved for all problems in NP.
- But the problem is that we don't have any polynomial solution for any of the problems in NP included both discussed. And also there is almost no hope. Only thing we can say here is that it has not been proved that P=NP and neither it has been proved that P<> NP so that can give us a little hope to work on and on.