Algorithms

Slides compiled from the internet (google wikipedia, books).

Textbook and references

- Cormen, 3rd edition.

  http://www.flipkart.com/introduction-algorithms-8120340078

References

- Design and Analysis of Computer Algorithms,

  by Ullman and Hopcroft

  http://www.flipkart.com/design-analysis-computer-algorithms-8131702057

- Algorithm Design Manual, 2nd ed, by Skiena

  http://www.flipkart.com/algorithm-design-manual-8184898657

Google, Wikipedia

- Mathematics for CS by Meyer, from MIT.
- MIT courseware video lectures

References (advanced)

Combinatorial Optimization: Algorithms and Complexity, by Papadimitriou and Steiglitz.
Computer and Intractability, by Garey and Johnson.
Approximation Algorithms, by Vazirani.
Randomized Algorithms, by Motwani and Raghavan.

Programming textbooks

- K&R: C Programming Language, by Kernighan and Ritchie, 2nd ed (not older ed).
- Unix Programing Environment, by Kernighan and Pike.
- C++ Language, by Stroustrup, 3rd ed (not older ed).
- Learning Python (or docs.python.org)
- Core Java, Vol 1 & 2, by Horstmann

Lab

- C programs using gcc/g++ IDE from http://www.codeblocks.org or MS-VC, not dev-cpp. recent gcc 4.
- On Linux or Windows (install cygwin for bash, make, configure, vim, ctags).
- Sometimes we will use Python, Java, C++,js
- Demonstration packages:
  - Maple, Mathematica
  - LP: glpk (gnu linear programming),Excel solver.
  - Optimization: Concorde TSP Solvers, Coin-OR
  - Number Theory: gnupg, gmp, pari, js.
  - graphviz for drawing graphs

Topics

- Algorithm running times
- Searching & Sorting: insertion, bubble, merge, quick, heap, radix.
- Data structures: stacks, queues, lists, hashing, heaps, trees, graphs.
- Searching: lists, binary, dfs, bfs.
- Greedy algorithms, minimum spanning tree.
- Dynamic programming: matrix chain multiplication, longest common subsequence,
- Graph algorithms: shortest path, matchings, max-flow.
- Matrix operations, LUP decomposition, Strassen matrix mult.
- Polynomial multiplication and fft.
- String matching: kmp, rabin karp, boyer moore, suffix trees.
- Linear programming, simplex.
- Number theory: gcd, ext-gcd, power-mod, crt, rsa,gpg.
- NP completeness, SAT, vertex cover, tsp, hamiltonian, partition…
- Approximation algorithms

Website and discussions

- Code https://sites.google.com/site/algorithms2013/
- IT-235 https://www.facebook.com/groups/algorithms.nitk
- CS-830 https://www.facebook.com/groups/applied.algorithms.nitk
- To ask questions, reply to questions, discussions, schedules.

Grading scheme

- 50% endsem
- 25% midsem exam
- 10+10% quizzes/handwritten assignments
- 5% attendance

0 marks to all copied assignments/exams.

What is an algorithm?

Algorithm is a step-by-step procedure for calculations. More precisely, an algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function

A program is an implementation of a algorithm.

Examples

- Compute area of circle, given the radius?
- Driving Directions from NITK to Manipal?
- Recipe to bake a cake?
- Predict tomorrow's weather?
- Autofocus a camera?
- Recognize face? Voice? Songs?
- Oldest algorithm? Euclid's GCD.

Reactive systems

- Examples: Traffic lights, lift, factory alarms
- Properties:
  - Always running, realtime input/output
  - No beginning, no end.
  - Correctness
  - Liveliness
  - Deadlock free
  - Fairness

What is computable?

- Alonzo Church created a method for defining functions called the λ-calculus
- Alan Turing created a theoretical model for a machine, now called a universal Turing machine
- Church–Turing **thesis** states that a function is algorithmically computable if and only if it is computable by a Turing machine.

Turing machine (TM)

- **Turing machine** is a hypothetical device that manipulates symbols on a strip of tape according to a table of rules.
- Despite its simplicity, a TM can be adapted to simulate the logic of any computer algorithm
- Used for explaining real computations.
- A universal Turing machine (UTM) is able to simulate any other Turing machine.
- Church–Turing thesis states that Turing machines capture the informal notion of effective method in logic and mathematics, and provide a precise definition of an algorithm or a 'mechanical procedure'.

Lambda Calculus (λ)

- Formulated by Alonzo Church as a way to formalize mathematics through the notion of functions
- (λ var body) ..Lambda expression/function
- Application:

(λ x y (* (+ 3 x) (- y 4))) 10 20

→ (* (+ 3 10) (- 20 4)) → (* 13 16) → 208

Church–Turing Thesis, the untyped lambda **calculus** is claimed to be capable of computing all effectively calculable functions.

The typed lambda **calculus** is a variety that restricts function application, so that functions can only be applied if they are capable of accepting the given input's "type" of data.

Combinatory Logic

- **SKI combinator calculus** is a computational system that may be perceived as a reduced version of untyped lambda calculus.
- Used in mathematical theory of algorithms because it is an extremely simple Turing complete language.
- Only two operators or combinators: **K***xy* = *x and* **S***xyz* = *xz*(*yz*).

- **K** := λ*x*.λ*y*.*x*
- **S** := λ*x*.λ*y*.λ*z*.*x z* (*y z*)
- **I** := λ*x*.*x*
- **B** := λ*x*.λ*y*.λ*z*.*x* (*y z*)
- **C** := λ*x*.λ*y*.λ*z*.*x z y*
- **W** := λ*x*.λ*y*.*x y y*
- **ω** := λ*x*.*x x*
- **Ω** := **ω ω**
- **Y** := λ*g*.(λ*x*.*g* (*x x*)) (λ*x*.*g* (*x x*))

Example computation or reduction: SKKX ==> KX(KX) ==> X

Functional programming

- FP is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.
- Based on just application of functions without side effects.
- Compare to the imperative programming style, which emphasizes changes in state.
- Functional programming has its roots in lambda calculus
- Languages: Haskell, Ocaml, ML, Lisp, Scala, Clojure,..
- Lisp example: **`(defun fibonacci (n &optional (a 0) (b 1)) (if (= n 0) a (fib (- n 1) b (+ a b))))`**
- Python Example: >>> (lambda x: x*2)(3)
  6

Languages

Recursively Enumerable Languages
"computable functions"
Turing machines
Recursive Languages
"algorithms"
"decision problems"
TMs that always halt
Context-Sensitive Languages
Context-Free Languages
Regular Languages
Some use in parsing.
- context-sensitive grammar
- linear bounded automata
Useful for parsing.
- context-free grammar
- pushdown automata
Useful for pattern matching.
- finite automata
- regular exps.
- regular grammar

What are these?

Oracle of Delphi

- A **Turing** machine with a black box, called an oracle, which is able to decide certain decision problems in a single operation.
- The problem can be of any complexity class.
- Even undecidable problems, like the halting problem can be answered by an oracle.

Cray super computer

Data center = Millions of PCs

FFT algorithm is used in audio/video/signal processing

Algorithms Part 2

- Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, will proceed through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state.
- The transition from one state to the next is not necessarily deterministic
- some algorithms, known as randomized algorithms, incorporate random input.

Some algorithms are harder than others

- Some algorithms are easy
  - Finding the max (largest or smallest) value in a list
  - Searching for a specific value in a list
- Some algorithms are a bit harder
  - Sorting a list
- Some algorithms are very hard
  - Finding the cheapest road tour of n cities
- Some algorithms are practically impossible
  - Factoring large numbers
- Some problems are undecidable (halting problem).

Algorithm: Maximum element

- Given a list, how do we find the maximum element in the list?
- To express the algorithm, we'll use pseudocode. Pseudocode is like a programming language, without worrying about syntax.

Finding the Maximum

```
procedure max (a[1..n] : array of integers)
    max := a_1
    for i := 2 to n
      if max < a_i then
        max := a_i
      {max is largest of a[1..i} // invariant
    endfor
    {max is the largest of a[1..n]} // invariant
    return max
end procedure
```

Maximum element running time

- How long does this take?
- If the list has *n* elements
- Worst case?
- Best case?
- Average case?

Properties of algorithms

- Algorithms generally share a set of properties:
  - Input
  - Output
  - Definiteness: the steps are defined precisely
  - Correctness: should produce the correct output
  - Finiteness: the steps required should be finite
  - Effectiveness: each step must be able to be performed in a finite amount of time
  - Generality: the algorithm *should* be applicable to all problems of a similar form.

Types of algorithms

- Deterministic
- Randomized, answer maybe different each time, but correct.
- Approximation, for hard problems answer is close to the best (optimal).
- Online, solve items as they come, without seeing all the items (choices). E.g. Hiring.

Algorithm Design Techniques:

- **Incremental Technique:** *Insertion sort, Selection sort*.
- **Divide and Conquer Technique:** *Merge sort, Quick sort, Binary Search.*
- **Dynamic Technique:** *Fibonacci numbers, Matrix chain multiplication, Knapsack problem.*
- **Greedy Technique:** *Shortest path problem, knapsack problem, spanning tree.*
- **Graph Traversal:** *Depth-first search, Breadth-first search, Traveling salesman.*

Algorithm Complexity

Time complexity : Specifies how the running time depends on the size of the input. A function mapping “size” n of input to “time” T(n) executed. (Independent of the type of computer used).

Space complexity : Function specifying mapping “size” n of input to space used.

Complexity

- Asymptotic Performance: *How does the algorithm behave as the problem size gets very large?*
  - Running time
  - Memory/storage requirements
- asymptotic (big-O) notation : *What does O(n) running time mean? $O(n^2)$? (n lg n)?*

Design and Analysis of Algorithms IT-235

Timings and location

- Tue 10-12pm
- Thu 11-12 and 1-2pm
- Classroom L601
- Office hours: After class
- Lab: Thu 2-5pm

Applied Algorithms
CS-830

- Timings and location

Tue 2-4pm

Thu 2-3pm

Classroom: Seminar Room

Office hours: After class

Marking scheme

- Final 50 %
- Midterm 25 %
- In semester 25%
  - Quizzes 10 %
  - Homework 10%
  - Attendance 5%

Website and discussions

- Code https://sites.google.com/site/algorithms2013/
- IT-235 https://www.facebook.com/groups/algorithms.nitk
- CS-830 https://www.facebook.com/groups/applied.algorithms.nitk
- Ask Class Representative to add you to the group
- To ask questions, reply to questions, discussions, schedules.
- Email: moshahmed/at/gmail

Growth of Functions

(and computers algorithms)

Pocket calculators

Cray super computer

Data center = Million PCs

Moore's Law - 2005
Transistors
Per Die
10^{10}
10^{9}
10^{8}
10^{7}
10^{6}
10^{5}
10^{4}
10^{3}
10^{2}
10^{1}
10^{0}
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005
2010
1K
4K
16K
64K
256K
1M
4M
16M
64M
128M
256M
512M
1G
2G
4004
8008
8080
8086
80286
386™ Processor
486™ Processor
Pentium® Processor
Pentium® II Processor
Pentium® III Processor
Pentium® 4 Processor
Itanium™ Processor
Itanium™ 2 Processor
1965 Data (Moore)
Memory
Microprocessor
Source: Intel

Microprocessor Transistor Counts 1971-2011 & Moore's Law

How does one measure algorithms?

- We can time how long it takes a computer
  - What if the computer is doing other things?
  - And what happens if you get a faster computer?
    - A 3 Ghz Windows machine will run an algorithm at a different speed than a 3 Ghz Linux

Step

- We can loosely define a “step” as a single computer operation
  - A comparison, an assignment, etc.
  - Regardless of how many machine instructions it translates into

- This allows us to put algorithms into broad categories of efficient-ness
  - An efficient algorithm on a slow computer will *always* beat an inefficient algorithm on a fast computer on a large problem.

Asymptotic

- Asymptotic efficiency of algorithms
  - How does the running time of an algorithm increase as the input size (n) increases infinitely $n \rightarrow \infty$
- Asymptotic notation ("the order of")
  - Define sets of functions that satisfy certain criteria and use these to characterize time and space complexity of algorithms

Function growth rates

- For input size n = 1000

$O(1)$	1	
$O(\log n)$	≈ 10	
$O(n)$	10^3	
$O(n \log n)$	$\approx 10^4$	
$O(n^2)$	10^6	
$O(n^3)$	10^9	
$O(n^4)$	10^{12}	
$O(n^c)$	10^{3*c}	*c* is a consant
2^n	$\approx 10^{301}$	
$n!$	$\approx 10^{2568}$	
n^n	10^{3000}	

Complexity	Term
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \lg n)$	n log n
$O(n^b)$	polynomial
$O(b^n)$ $b > 1$	exponential
$O(n!)$	factorial

5000
4000
3000
2000
1000
0
1
3
5
7
9
11
13
15
17
19
f(n) = n
f(n) = log(n)
f(n) = n log(n)
f(n) = n^2
f(n) = n^3
f(n) = 2^n

Exponential versus Polynomial

Logarithmic scale!

FIGURE 3 A Display of the Growth of Functions Commonly Used in Big-*O* Estimates.

Algorithm Running Time

Given a size n problem, an algorithm runs $O(f(n))$ time:

1. $O(f(n))$: upper bound. (Ω :lower θ: equal)

2.Polynomial: $f(n)=1$(constant), n(linear), n^2(quadratic), n^k.

3.Exponential: $f(n)=2^n$, $n!$, n^n.

Time	$n=1$	$n=10$	$n=100$	$n=1000$
n	1	10	10^2	10^3
n^2	1	10^2	10^4	10^6
n^{10}	1	10^{10}	10^{20}	10^{30}
2^n	2	$>10^3$	$>10^{30}$	$>10^{300}$
$n!$	1	$>10^6$	$>10^{150}$	$>10^{2500}$

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3 nsec	0.01 μ	0.02 μ	0.06 μ	0.51 μ	0.26 μ
16	4 nsec	0.02 μ	0.06 μ	0.26 μ	4.10 μ	65.5 μ
32	5 nsec	0.03 μ	0.16 μ	1.02 μ	32.7 μ	4.29 sec
64	6 nsec	0.06 μ	0.38 μ	4.10 μ	262 μ	5.85 cent
128	0.01 μ	0.13 μ	0.90 μ	16.38 μ	0.01 sec	10^{20} cent
256	0.01 μ	0.26 μ	2.05 μ	65.54 μ	0.02 sec	10^{58} cent
512	0.01 μ	0.51 μ	4.61 μ	262.14 μ	0.13 sec	10^{135} cent
2048	0.01 μ	2.05 μ	22.53 μ	0.01 sec	1.07 sec	10^{598} cent
4096	0.01 μ	4.10 μ	49.15 μ	0.02 sec	8.40 sec	10^{1214} cent
8192	0.01 μ	8.19 μ	106.50 μ	0.07 sec	1.15 min	10^{2447} cent
16384	0.01 μ	16.38 μ	229.38 μ	0.27 sec	1.22 hrs	10^{4913} cent
32768	0.02 μ	32.77 μ	491.52 μ	1.07 sec	9.77 hrs	10^{9845} cent
65536	0.02 μ	65.54 μ	1048.6 μ	0.07 min	3.3 days	10^{19709} cent
131072	0.02 μ	131.07 μ	2228.2 μ	0.29 min	26 days	10^{39438} cent
262144	0.02 μ	262.14 μ	4718.6 μ	1.15 min	7 mnths	10^{78894} cent
524288	0.02 μ	524.29 μ	9961.5 μ	4.58 min	4.6 years	10^{157808} cent
1048576	0.02 μ	1048.60 μ	20972 μ	18.3 min	37 years	10^{315634} cent

Table of common time complexities

Further information: Computational complexity of mathematical operations

The following **table** summarises some classes of commonly encountered time complexities. In the **table**, poly(x) = $x^{O(1)}$, i.e., polynomial in x.

Name	Complexity class	Running time ($T(n)$)	Examples of running times	Example algorithms
constant time		$O(1)$	10	Determining if an integer (represented in binary) is even or odd
inverse Ackermann time		$O(\alpha(n))$		Amortized time per operation using a disjoint set
iterated logarithmic time		$O(\mathbf{log}^* n)$		Distributed coloring of cycles
log-logarithmic		$O(\mathbf{log}\ \mathbf{log}\ n)$		Amortized time per operation using a bounded

				priority queue[2]
logarithmic time	DLOGTIME	$O(\log n)$	$\log n$, $\log(n^2)$	Binary search
polylogarithmic time		$\text{poly}(\log n)$	$(\log n)^2$	
fractional power		$O(n^c)$ where $0 < c < 1$	$n^{1/2}$, $n^{2/3}$	Searching in a kd-tree
linear time		$O(n)$	n	Finding the smallest item in an unsorted array
"n log star n" time		$O(n \log^* n)$		Seidel's polygon triangulation algorithm.
linearithmic time		$O(n \log n)$	$n \log n$, $\log n!$	Fastest possible comparison sort
quadratic time		$O(n^2)$	n^2	Bubble sort; Insertion sort
cubic time		$O(n^3)$	n^3	Naive multiplication of two $n \times n$ matrices. Calculating partial correlation.
polynomial time	P	$2^{O(\log n)} = \text{poly}(n)$	n, $n \log n$, n^{10}	Karmarkar's algorithm for linear programming; AKS primality test
quasi-polynomial time	QP	$2^{\text{poly}(\log n)}$	$n^{\log \log n}$, $n^{\log n}$	Best-known $O(\log^2 n)$-approximation algorithm for the directed Steiner tree problem.

sub-exponential time (first definition)	SUBEXP	$O(2^{n^{\varepsilon}})$ for all $\varepsilon > 0$	$O(2^{\log n^{\log \log n}})$	Assuming complexity theoretic conjectures, BPP is contained in SUBEXP.[3]
sub-exponential time (second definition)		$2^{o(n)}$	$2^{n^{1/3}}$	Best-known algorithm for integer factorization and graph isomorphism
exponential time	E	$2^{O(n)}$	1.1^n, 10^n	Solving the traveling salesman problem using dynamic programming
factorial time		$O(n!)$	$n!$	Solving the traveling salesman problem via brute-force search
exponential time	EXPTIME	$2^{\text{poly}(n)}$	2^n, 2^{n^2}	
double exponential time	2-EXPTIME	$2^{2^{\text{poly}(n)}}$	2^{2^n}	Deciding the truth of a given statement in Presburger arithmetic

From http://en.wikipedia.org/wiki/Time_complexity

Complexity

- **Worst-Case Complexity:** the maximum number of steps taken on an instance of size n.
- **Best-Case Complexity** the minimum number of steps taken on an instance of size n.
- **Average-Case Complexity** the average number of steps taken on any instance
- of size n.

Example of difficult problem

- Factoring a composite number into it's component primes is $O(2^n)$
  - Where n is the number of bits in the number, ie. The length of the number in binary bits.
- This, if we choose 2048 bit numbers (as in RSA keys), it takes 2^{2048} (10^{617}) steps to factor it.

NP Hard problems

"I CAN'T SOLVE IT - BUT NEITHER CAN ALL THESE FAMOUS PEOPLE!"

Solving hard problems in practice

- SIMPLEX - Optimization using simplex method can take exponential steps in worst case, but in practice it finishes quickly.
- Hard problems can be solved quickly using fast approximation algorithms, if you don’t want the very best answer.

Knapsack and bin packing are NP complete problems

Moving a drill minimal amount to make a circuit board

Soldering on a circuit board

Beyond Polynomial

- Polynomial: Any expression of the form n^c, where c is a constant Thus, any function that is $O(n^k)$ is a polynomial-time function, for a fixed k.
- 2^n, $n!$, n^n are exponential, not polynomial functions
- Elementary: +,-,*,/,log,e(n),n!
- Non-elementary: Ackerman

Ackerman function

the fastest growing function

```
ack 0  n = n + 1                    - - haskell
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))
```

Values of $A(m, n)$

m\n	0	1	2	3	4	n
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2=2+(n+3)-3$
2	3	5	7	9	11	$2n+3=2\cdot(n+3)-3$
3	5	13	29	61	125	$2^{(n+3)}-3$
4	13 $=2^{2^2}-3$	65533 $=2^{2^{2^2}}-3$	$2^{65536}-3$ $=2^{2^{2^{2^2}}}-3$	$2^{2^{65536}}-3$ $=2^{2^{2^{2^{2^2}}}}-3$	$2^{2^{2^{65536}}}-3$ $=2^{2^{2^{2^{2^{2^2}}}}}-3$	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3}-3$

Inverse-Ackermann function

- $\alpha(n)$, slowest growing function.
- As slow as ackermann is fast, not inverse
- Algorithm 'Union-Find' runtime is $O(m\ \alpha(n) + n)$, where n is the number of elements and m is the total number of Union and Find operations performed.

The following table shows the first values of $\alpha_k(n)$:

n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	...	n
$\alpha_1(n)$ =	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10	11	11	12	12	13	...	$[n / 2]$
$\alpha_2(n)$ =	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	...	$[\log_2 n]$
$\alpha_3(n)$ =	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4	4	...	$\log^* n$
$\alpha_4(n)$ =	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	...	
...																											

Install following software

Windows:

- Cygwin with gcc, g++, xwindows
- Codeblocks (gcc and g++ with ide)
- python
- Java jdk

Linux

- Codeblocks

PYTHON
THIS IS PLAGIARISM. YOU CAN'T JUST "IMPORT ESSAY."
JAVA
I'M TWO PAGES IN AND I STILL HAVE NO IDEA WHAT YOU'RE SAYING.
C++
I ASKED FOR ONE COPY, NOT FOUR HUNDRED.
UNIX SHELL
I DON'T HAVE PERMISSION TO READ THIS.
ASSEMBLY
DID YOU REALLY HAVE TO REDEFINE EVERY WORD IN THE ENGLISH LANGUAGE?
C
THIS IS GREAT, BUT YOU FORGOT TO ADD A NULL TERMINATOR. NOW I'M JUST READING GARBAGE.
LATEX
YOUR PAPER MAKES NO GODDAMN SENSE, BUT IT'S THE MOST BEAUTIFUL THING I HAVE EVER LAID EYES ON.
HTML
THIS IS A FLOWER POT.
2010-2011 SOMETHINGOFTHATILK.COM

Home work

- Write a program to print the table of
- Ackermann[0..3, 0..4] in C, Python and Java.

Solutions: Ackerman in python, C, Java

```python
# naïve ackermann function
def ackermann(m, n):
    global calls
    calls += 1
    if m == 0:
        return n + 1
    elif n == 0:
        return ackermann(m - 1, 1)
    else:
        return ackermann(m - 1, ackermann(m, n - 1))
```

```c
int A (int m, int n)  {
    if (m == 0) return n + 1;
    if (n == 0) return A(m - 1, 1);
    return A(m - 1, A(m, n - 1));
}
```

```java
public class Ackermann {
    public static long ackermann(long m, long n) {
        if (m == 0) return n + 1;
        if (n == 0) return ackermann(m - 1, 1);
        return ackermann(m - 1, ackermann(m, n - 1));
    }
    public static void main(String[] args) {
        long M = Long.parseLong(args[0]);
        long N = Long.parseLong(args[1]);
        System.out.println(ackermann(M, N));
    }
}
```

Questions

Q. Which has higher complexity:

1. O(n!) or O(exp(n))?
2. O(n log n) or O(n^1.5)?
3. O(log n) or O(sqrt(n))?
4. O(n log n) or O(n)?
5. O(n log n) or O((log n)^2) ?

O-notation

Growth of functions

Big O notation (Upper bound, Worst case)

$$O(g(n)) = \{\ f(n) : \exists\ c, m > 0 : 0 \le f(n) \le c\, g(n),\ \forall\ n \ge m\ \}$$

f(n) always lies
on or below c.g(n).

f(n) = O(g(n))

Commonly used
notation

f(n) = O(g(n))

Correct notation

f(n) in O(g(n))

Big Omega Ω-notation (Lower bound, Best case)

$\Omega(g(n)) = \{ f(n)$: ∃ positive constants c and m such that $0 \leq c\, g(n) \leq f(n), \forall\, n \geq m \}$

f(n) always lies on or above c. g(n).

f(n) = Ω(g(n)

Think of the equality as meaning *in the set of functions*.

Big Theta Θ-notation (Tight bound)

$$\Theta(g(n)) = \{f(n): \exists \text{ positive constants } c_1, c_2 \text{ and } m : 0 \le c_1\, g(n) \le f(n) \le c_2\, g(n)\ \forall\ n \ge m\ \}$$

f(n) always lies
between c1.g(n) and c2.g(n).

f(n) = Θ(g(n)

Other Asymptotic Notations

- **Little o**
  - A function f(n) is o(g(n)) if $\exists$ positive constants *c* and *m* such that
    $f(n) < c\, g(n)\ \forall\ n \geq m$
- **little-omega**
  - A function f(n) is ω(g(n)) if $\exists$ positive constants *c* and *m* such that
    $c\, g(n) < f(n)\ \forall\ n \geq m$

- $f(n) = O(g(n)) \approx a \leq b$ f no faster than **g**
- $f(n) = \Omega(g(n)) \approx a \geq b$ f no slower than **g**
- $f(n) = \Theta(g(n)) \approx a = b$ f about as fast as **g**
- $f(n) = o(g(n)) \approx a < b$ f slower than **g**
- $f(n) = w(g(n)) \approx a > b$ f faster than **g**

Space Complexity

- Number of memory cells (or words) needed to carry out the computational steps required to solve an instance of the problem excluding the space allocated to hold the input. ***Only the work space.***
- All previous asymptotic notation definitions are also applied to space complexity.
- Naturally, in many problems there is a time-space tradeoff: The more space we allocate for the algorithm the faster it runs, and vice versa

Complexity class	Model of computation	Resource constraint
DTIME($f(n)$)	Deterministic Turing machine	Time $f(n)$
P	Deterministic Turing machine	Time poly(n)
EXPTIME	Deterministic Turing machine	Time $2^{\text{poly}(n)}$
NTIME($f(n)$)	Non-deterministic Turing machine	Time $f(n)$
NP	Non-deterministic Turing machine	Time poly(n)
NEXPTIME	Non-deterministic Turing machine	Time $2^{\text{poly}(n)}$
DSPACE($f(n)$)	Deterministic Turing machine	**Space** $f(n)$
L	Deterministic Turing machine	**Space** O(**log** n)
PSPACE	Deterministic Turing machine	**Space** poly(n)
EXPSPACE	Deterministic Turing machine	**Space** $2^{\text{poly}(n)}$
NSPACE($f(n)$)	Non-deterministic Turing machine	**Space** $f(n)$
NL	Non-deterministic Turing machine	**Space** O(**log** n)
NPSPACE	Non-deterministic Turing machine	**Space** poly(n)
NEXPSPACE	Non-deterministic Turing machine	**Space** $2^{\text{poly}(n)}$

It turns out that PSPACE = NPSPACE and EXPSPACE = NEXPSPACE by Savitch's theorem.

Space time hierarchy

EXPSPACE
EXPTIME
PSPACE = NPSPACE = IP
NP
co-NP
RP
ZPP
co-RP
P
NL=coNL
L
BPP

Complexity of Traveling salesman

xkcd.com

Basic math for algorithms

Logarithmic spiral

Logarithms – notation, simple facts

1. $\lg n = \log_2 n$ (logarithm of base 2)

2. $\ln n = \log_e n$ (natural logarithm to base e=2.7182818284590451)

3. $\log n = \log_{10} n$ (common logarithm to base 10)

$\lg^k n = (\lg n)^k$

$\lg \lg n = \lg (\lg n)$ ..right associative.

Logarithms – notation, simple facts

$a = b^{\log_b a}$

$\log_c (ab) = \log_c a + \log_c b$

$\log_c a^n = n \log_c a$

$\log_b (1/a) = -\log_b a$

Slide rule multiplies numbers by adding their logs, e.g log(2)+log(3)=log(2*3)

Changing base of log

- $\log_a n$
- $= \log_b n * \log_a b$
- $= \log_b n / \log_b a$
- $\log_a b$ is a constant when a, b are constants.
- $\log_{10} n = \ln n / \ln 10 = \ln n * 0.4343$

```
C:\> c:\python33\python.exe
>>> from numpy import *
>>> log10(e)
0.43429448190325182
```

Iterated log*

- lg* *n* = number of times lg function must be applied to make n ≤ 1.

x	$\lg^* x$
$(-\infty, 1]$	0
$(1, 2]$	1
$(2, 4]$	2
$(4, 16]$	3
$(16, 65536]$	4
$(65536, 2^{65536}]$	5

Table showing range of x, and the corresponding value of lg*(x), it is a very very slowly growing function

Floor and ceiling

- Floor(x) = int(x) = $\lfloor x \rfloor$
- Ceiling(x)= int(x+0.5)= $\lceil x \rceil$
- Eg. $\lfloor 2.5 \rfloor = 2$ and $\lceil 2.5 \rceil = 3$
- Mod(x,y) = remainder(x/y) = x % y
- E.g. 7 mod 4 = 3, $7 \equiv_4 3$, $7 \equiv 3 \pmod 4$
- -2 mod 3 = 1, 7 mod 0 = undefined.
- x div y = int(x/y), e.g. 7 div 3 = 2,
- x div 0 = undefined
- x mod y = x – (x div y) y

Exponents

- $A^0 = 1$
- $A^1 = A$
- $A^{-1} = 1/A$
- (A^m)^n = A^(mn) != A^(m^n) [non-associative]
- A^m*A^n = A^(m+n)
- e^(i*pi) = -1 .. from complex numbers.
- n! = n*(n-1)*…*1 ✍ n^n

Fibonacci sequence

- F(0) = 0, F(1)=1,
- F(n)=F(n-1)+F(n-2)
- 0,1,1,2,3,5,8,13,21,34,55
- Golden ratio:

  ϕ = (1 + sqrt(5))/2 = 1.618

  ϕ' = (1 - sqrt(5))/2 = 0.618

  F(n) = (ϕ^n - ϕ'^n) / sqrt(5) $\approx$ int(ϕ^n / sqrt(5))
- Fibonacci has exponential growth

Fibonacci numbers in nature

Master Method (CLR)

for Solving Recurrences about complexity of algorithmS

Example: Mergesort

- DIVIDE the input sequence in half
- RECURSIVELY sort the two halves
  - basis of the recursion is sequence with 1 key
- COMBINE the two sorted subsequences by merging them

Mergesort Example

Recurrence Relation for Mergesort

- Let $T(n)$ be worst case time on a sequence of n keys
- If $n = 1$, then $T(n) = \Theta(1)$ (constant)
- If $n > 1$, then $T(n) = 2\,T(n/2) + \Theta(n)$
  - two subproblems of size $n/2$ each that are solved recursively
  - $\Theta(n)$ time to do the merge

Recurrence Relations

How To Solve Recurrences

- Ad hoc method:
  - expand several times
  - guess the pattern
  - can verify with proof by induction
- Master theorem
  - general formula that works if recurrence has the form $T(n)$

    $= aT(n/b) + f(n)$
    - a is number of sub-problems
    - n / b is size of each subproblem
    - $f(n)$ is cost of non-recursive part

Master Theorem

Consider a recurrence of the form

$$T(n) = a\,T(n/b) + f(n)$$

with $a \geq 1$, $b > 1$, and $f(n)$ eventually positive.

a) If $f(n) = O(n^{\log_b(a)-\varepsilon})$, then $T(n)=\Theta(n^{\log_b(a)})$.

b) If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n)=\Theta(n^{\log_b(a)} \log(n))$.

c) If $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ and $f(n)$ is regular, then $T(n)=\Theta(f(n))$

$f(n)$ is regular iff eventually $a*f(n/b) \leq c*f(n)$ for some constant $c<1$

Master Theorem

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Master method details

Essentially, the Master theorem compares the function $f(n)$ with $g(n) = n^{\log_b(a)}$.

Roughly, the theorem says:

a)If $f(n) \ll g(n)$ then $T(n)=\Theta(g(n))$.

b)If $f(n) \approx g(n)$ then $T(n)=\Theta(g(n)\log(n))$.

c)If $f(n) \gg g(n)$ then $T(n)=\Theta(f(n))$.

Master method details (CLR)

Before applying the master theorem to some examples, let's spend a moment trying to understand what it says. In each of the three cases, we compare the function $f(n)$ with the function $n^{\log_b a}$. Intuitively, the larger of the two functions determines the solution to the recurrence. If, as in case 1, the function $n^{\log_b a}$ is the larger, then the solution is $T(n) = \Theta(n^{\log_b a})$. If, as in case 3, the function $f(n)$ is the larger, then the solution is $T(n) = \Theta(f(n))$. If, as in case 2, the two functions are the same size, we multiply by a logarithmic factor, and the solution is $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Beyond this intuition, you need to be aware of some technicalities. In the first case, not only must $f(n)$ be smaller than $n^{\log_b a}$, it must be *polynomially* smaller.

Nothing is perfect…

The Master theorem does not cover all possible cases. For example, if

$$f(n) = \Theta(n^{\log_b(a)} \log n),$$

then we lie between cases 2) and 3), but the theorem does not apply.

There exist better versions of the Master theorem that cover more cases, but these are even harder to memorize.

Idea of the Proof

Let us iteratively substitute the recurrence:

$$
\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a(aT(n/b^2)) + f(n/b)) + bn \\
&= a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^3T(n/b^3) + a^2 f(n/b^2) + af(n/b) + f(n) \\
&= \ldots \\
&= a^{\log_b n}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i) \\
&= n^{\log_b a}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i)
\end{aligned}
$$

Idea of the Proof

Thus, we obtained

$$T(n) = n^{\log_b(a)}\, T(1) + \sum a^i\, f(n/b^i)$$

The proof proceeds by distinguishing three cases:

1) The first term in dominant: $f(n) = O(n^{\log_b(a)-\varepsilon})$
2) Each part of the summation is equally dominant: $f(n) = \Theta(n^{\log_b(a)})$
3) The summation can be bounded by a geometric series: $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ and the regularity of f is key to make the argument work.

Example 2. Strassen's Matrix Multiplication

- Strassen found a way to get all the required information with only 7 matrix multiplications, instead of 8, using divide and conquer.
- Recurrence for new algorithm is
  - $T(\mathrm{n}) = 7T(n/2) + \Theta(n^2)$

Solving the Recurrence Relation

Applying the Master Theorem to

$T(n) = a\,T(n/b) + f(n) = 7T(n/2) + \Theta(n^2)$

We have: $a=7$, $b=2$, and $f(n)=\Theta(n^2)$.

Since $f(n) = O(n^{\log_b(a)-\varepsilon}) = O(n^{\log_2(7)-\varepsilon})$,

case a) applies and we get

$T(n)= \Theta(n^{\log_b(a)}) = \Theta(n^{\log_2(7)}) = O(n^{2.81})$.

Example 3

- $T(n) = 9\,T(n/3)+n$,
- We have $a=9$, $b=3$, $f(n)=n$
- $n^{(\log_b a)}=n^{(\log_3 9)} = n^2$
- $f(n)=n= n^1=n^{(2-e)}= n^{((\log_b a)-e)}$, $e=1$
- Case a) of master theorem
- $T(n) = \Theta(n^2)$.

Example 3

- $T(n)=T(2n/3)+1$,
- with $a=1$, $b=3/2$, $f(n)=1$
- $n^{(\log_b a)}=n^{(\log_{3/2} 1)} = n^0=1$
- Case 2) $f(n)= \Theta(n^{(\log_b a)})= \Theta(1)$
- Hence $T(n)= \Theta(\lg n)$

Example 4

- $T(n)=3T(n/4)+n \lg n$,
- with $a=3$, $b=4$, $f(n)=n \lg n$.
- $n^{(\log_b a)}=n^{(\log_4 3)} =O(n^{0.793})$
- $f(n)= \Omega(n^{(\log_4 3+e)})$, $e \geq 0.2$
- For suff. large n, $3(n/4)\lg(n/4) <= \frac{3}{4} * n * \lg(n) = c\, f(n)$, with $c=3/4$
- By case 3) $T(n)= \Theta(n \lg n)$

Example 5

- $T(n)=2T(n/2)+n \lg n$,
- with $a=2$, $b=2$, $f(n)=n \lg n$.
- $n^{(\log_b a)}=n^{(\log_2 2)} =n^1=n$
- $f(n)= n \lg n$, but case 3 does NOT apply, since $n \lg(n)$ is not polynomially larger than n
- $f(n)/n^{(\log_b a)}=n \lg n / n = \lg n$ is less than $n^{e,}$ for all constant e.

Data Structures

Linked lists, Stacks, Queues

Linked list

List Handle
Datum
Link
Datum
Link
Datum
Link
Datum
Link
Datum
Null

Circular doubly linked list

Properties of linked lists

- Advantages?
- Disadvantages?

Linked Lists

- A linked list consists of a finite sequence of elements or nodes that contain information plus (except possibly the last one) a link to another node.
- If node *x* points to node *y*, then *x* is called the *predecessor* of *y* and *y* the *successor* of *x*. There is a link to the first element called the *head* of the list.
- If there is a link from the last element to the first, the list is called *circular*.
- If in a linked list each node (except possibly the first one) points also to its *predecessor*, then the list is called a *doubly linked list*.
- If the first and the last nodes are also connected by a pair of links, then we have a *circular doubly linked list*.

The two primary operations on linked lists are insertion and deletion.

a) **Single linked list**
b) **Circular singly linked list**
c) **doubly linked list**
d) **Doubly circular linked list**

Insertion in doubly linked list

Stack

Stack pointer push and pop

Stacks

- A ***stack*** is a linked list in which insertions and deletions are permitted only at one end, called the ***top*** of the stack. The stack pointer (sp) points to the top of the stack.
- ***stack*** supports two basic operations:
  - ***pop(S)*** returns the top of the stack and removes it from S.
  - ***push(S, x)*** adds *x* to top of *S* and updates the sp to point to *x*.

Queues

In a queue, all operations take place at one end of the queue or the other. The "enqueue" operation adds an item to the "back" of the queue. The "dequeue" operation removes the item at the "front" of the queue and returns it.

Queue

- **A *queue* is a list in which insertions are permitted only at one end of the list called its *rear,* and all deletions are from the other end called the *front* of the queue.**
- **The operations supported by queues are**
  - ***Add(Q,x)*** **adds x at the rear of the queue Q.**
  - ***Delete(Q,x)*** **delete x from the front of the queue Q.**

Circular queue

Graphs

Data Structure

Graph Theory - History

Leonhard Euler's paper on
"*Seven Bridges of Königsberg*" ,
published in 1736.

Famous problems

- Euler circuit: travel on every edge once
- Hamiltonian path: visit every vertex once
  - NP complete
- Traveling salesman problem
  - Shortest hamiltonian path (Visit every vertex once and travel least amount of edges).
- Graph Isomorphism: Are two graphs identical?
- Graph coloring: Color the vertices, so that adjacent vertices have different colors
- 4 Color problem: Every map can be coloured with just 4 colours, and adjacent countries have different colors

4 color problem

In 1852 Francis Guthrie posed the “four color problem” which asks if it is possible to color, using only four colors, any map of countries in such a way as to prevent two bordering countries from having the same color.

Proved in 1976 by Kenneth Appel and Wolfgang Haken, with heavy use of computers.

Uses

- Transportation: travel and routing
- Arranging wires on circuit boards
- Maximizing flow of liquids in network.
- Routing network packets
- Any more?

What is a Graph?

- G*raph* is a set of vertices (nodes) joined by a set of edges (lines or arrows).

Directed graph

Graph

- **Graph.** A graph $G=(V,E)$ is
- Vertices V (nodes) and Edges (links) E.
- Edge e connects two nodes
- Edge e can have a weight (distance), or 1.
- Directed Graph if edges have direction.
- Size of Graph is measured by size of V and E

Example

- V:={1,2,3,4,5,6}
- E:={{1,2},{1,5},{2,3},{2,5},{3,4},{4,5},{4,6}}

Vertex degree

- ***Vertex degree*** **is**

  Undirected G:

  number of vertices adjacent to the vertex**.**

  Directed G:

  In-degree*:* number of edges directed to *vi*

  Out-degree: number of edges directed out of to *vi*

Degree

The degree of B is 2.

- Number of edges incident on a node

Degree (Directed Graphs)

The in degree of 2 is 2 and the out degree of 2 is 3.

- In degree: Number of edges entering
- Out degree: Number of edges leaving

- Degree = indegree + outdegree

- **<u>Path</u>** ***in G from vertex v_1 to vertex v_k is a*** *sequence of vertices <v_1, v_2, …,v_k> with k-1 edges between adjacent v_i and v_i+1.*
  - ***<u>length</u>*** of a path is the number of edges in the path.
  - ***<u>Simple path</u>*** **if all its vertices are distinct.**
  - ***<u>Cycle in a path</u>*** **if *v*1 = *v*k.**

Path

- A *path* is a sequence of vertices such that there is an edge from each vertex to its successor.
- A path is ***simple*** if each vertex is distinct.

Simple path from 1 to 5
= [1, 2, 4, 5]

Our text's alternates the vertices and edges.

If there is path *p* from *u* to *v* then we say *v* is **reachable** from *u* via *p*.

Connectivity

- G is ***connected*** if
  - you can get from any node to any other by following a sequence of edges, i.e.
  - any two nodes are connected by a path.
- ***G is strongly connected*** if
  - there is a *directed* path from any node to any other node.

Sparse/Dense Graphs

- A graph is ***sparse*** if $|E| \approx |V|$
- A graph is ***dense*** if $|E| \approx |V|^2$.

Complete Graph

- Denoted K_n
- Every pair of vertices are adjacent
- N vertices, has n(n-1) edges

Bipartite graph

- V can be partitioned into 2 sets V_1 and V_2 such that $(u,v) \in E$ implies

  either

  $u \in V_1$ and $v \in V_2$

  OR

  $v \in V_1$ and $u \in V_2$.

Complete Bipartite Graph

- Bipartite Variation of Complete Graph
- Every node of one set is connected to every other node on the other set

Planar Graph

K4

- Can be drawn on a plane such that no two edges intersect
- K_4 is the largest complete graph that is planar
- K5, K6, .. are not planar

Isomorphism Problem

- Determining whether two graphs are isomorphic
- Although these graphs look very different, they are isomorphic
- One isomorphism between them is:

  f(a) = 1 f(b) = 6

  f(c) = 8 f(d) = 3

  f(g) = 5 f(h) = 2

  f(i) = 4 f(j) = 7

Tree

- Connected Acyclic Graph
- Two nodes have *exactly* one path between them

Graph Data Structures

Representations of graphs

Adjacency list adjacency matrix

(a)

(b)

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

(c)

(a)

(b)

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

(c)

Adjacency lists

- $G = (V, E)$ an array of adjacencies: for each vertex $v \in V$, store a list of vertices adjacent to v
- *Storage requirements:*
  - directed graph, the sum of the lengths of all the adjacency lists is $|E|$,
  - undirected graph, the sum of the lengths of all the adjacency lists is $2\,|E|$,
- *Storage requirements: For both directed and undirected graphs*, amount of memory it requires is $\Theta(|V| + |E|)$.

Adjacency lists

- Advantage:
  - Saves space for sparse graphs. Most graphs are sparse.
  - Traverse all the edges that start at v, in θ(degree(v))
- Disadvantage:
  - Check for existence of an edge (v, u) in worst case time θ(degree(v))

Problem:
Show the adjacency-list of

1

2

3

4

5

Problem: adjacency-list 2

Problem: adjacency-list 3

Adjacency matrix

- Represents the graph as a *n* x *n* matrix A:
  - A[*i*, *j*] = 1 if edge (*i*, *j*) $\in$ E (*or weight of edge)*
    = 0 if edge (*i*, *j*) $\notin$ E
- Storage requirements: $O(V^2)$
- Efficient for small graphs
  - Especially if store just one bit/edge
  - Undirected graph: only need one diagonal of matrix

Problem: ***Adjacency matrix***

	1	2	3	4	5
1	x	x	x	x	x
2	x	x	x	x	x
3	x	x	x	x	x
4	x	x	x	x	x
5	x	x	x	x	x

Adjacency matrix 2

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

Large graphs

Graph Problems

- Bipartite, matchings
- Shortest path
- Max flow
- Hamitonian path
- TSP
- Vertex cover
- Colouring
- Isomorphism

Searching algorithms

- Given a list, find a specific element in the list
- We will see two types
  - Linear search
    - a.k.a. sequential search
  - Binary search

Linear search

- Given a list, find a specific element in the list
  - List does NOT have to be sorted!

procedure linear_search (x: integer; a_1, a_2, …, a_n: integers)
i := 1
while ($i \leq n$ and $x \neq a_i$)
  i := i + 1
if $i \leq n$ **then** *location* := i
else *location* := 0

Algorithm 2: Linear search, take 1

procedure linear_search (x: integer; a_1, a_2, …, a_n: integers)

$i := 1$

while ($i \leq n$ and $x \neq a_i$)

  $i := i + 1$

if $i \leq n$ **then** *location* := i

else *location* := 0

x	3
location	8

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
4	1	7	0	5	2	9	3	6	8

i	[redacted]

Algorithm 2: Linear search, take 2

procedure linear_search (x: integer; a_1, a_2, …, a_n: integers)

$i := 1$

while ($i \leq n$ and $x \neq a_i$)

  $i := i + 1$

if $i \leq n$ **then** *location* := i

else *location* := 0

x	11
location	0

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
4	1	7	0	5	2	9	3	6	8

i [redacted]

Linear search running time

- How long does this take?

- If the list has *n* elements, worst case scenario is that it takes *n* “steps”
  - Here, a step is considered a single step through the list

Algorithm 3: Binary search

- Given a list, find a specific element in the list
  - List MUST be sorted!
- Each time it iterates through, it cuts the list in half

procedure binary_search (x: integer; a_1, a_2, …, a_n: increasing integers)
i := 1 { i is left endpoint of search interval }
j := n { j is right endpoint of search interval }
while $i < j$
begin
  m := $\lfloor (i+j)/2 \rfloor$ { m is the point in the middle }
  if x > a_m **then** i := m+1
  else j := m
end
if $x = a_i$ **then** *location* := i
else *location* := 0

{*location* is the subscript of the term that equals x, or it is 0 if x is not found}

Algorithm 3: Binary search, take 1

procedure binary_search (x: integer; a_1, a_2, …, a_n: increasing integers)

```
i := 1
j := n
while i < j
begin
    m := ⌊(i+j)/2⌋
    if x > a_m then i := m+1
    else j := m
end
if x = a_i then location := i
else location := 0
```

x	14
location	7

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
2	4	6	8	10	12	14	16	18	20

i 1 2 3 4 5 6 7 8 9 10 m 5 6 7 8 j 10 9 8 7 6

Binary Search

- Assumes list is sorted
- Example of a recursive algorithm as algorithm applied to smaller list

Binary search in worst case takes

$O(\lg_2 n)$ steps, where n = size of the list.

Binary search running time

- How long does this take (worst case)?

- If the list has 8 elements
  - It takes 3 steps
- If the list has 16 elements
  - It takes 4 steps
- If the list has 64 elements
  - It takes 6 steps

- If the list has *n* elements
  - It takes $\log_2 n$ steps

Comparing Sequential and Binary Search

Sequential search is worst case and average case **O(n)**

Binary search is worst case **O(log n)**

n	log n
$2 = 2^1$	1
$16 = 2^4$	4
$1024 = 2^{10}$	10
1 million $= 2^{20}$	20
1 billion $= 2^{30}$	30

Median in Linear Time

Selection in Linear time (divide and conquer)

Problem: Given an array *A* of *n* numbers *A*[*1*..*n*] and a number *k* (*1*<*k*<*n*), find the *k*th smallest number in *A*.

The median is the $\lceil n/2 \rceil$ th smallest element

Example: If A=(7, 4, 8, 2, 4); then |A| = 5 and ceil(5/ 2)= 3rd smallest element (and median) is 4.

Solutions for our problem:

1. Naive approach:
   1. Sort the array (heap,merge) → O(n log n)
   2. return the *k* th element. → O(1)
   3. Total Execution time →$Q(n \log n)$
2. There is a linear-time selection algorithm

Strategy1: Partition-based (quick sort) selection

- Choose one element *p as pivot* from array A.
- Split input into three sets:
  - **LESS: elements from A that are smaller than p**
  - **EQUAL: elements from A that are equal to p**
  - **MORE: elements from A that are greater than p**
- We have three cases:
  - **i < |L| → the element we are looking for is also the *i*th smallest number in L**
  - **|L| < i < |L| + |E| →the element we are looking for is p.**
  - **|L| + |E| < i→the element we are looking for is the (i - |L| - |E|)th smallest element in M.**
- This takes O(n) in the best and average case, Unfortunatelly the running time can grow to $O(n^2)$ in the worst case.

Linear Time selection algorithm

Procedure *select* (*A, low…high, k*)

1. If n is small, for example n<6, then, sort and return the kth smallest element .(bounded by 7 steps)
2. If n>5, then partition the numbers into groups of 5. (time n/5)
3. Sort the elements in each group. Select the middle elements (medians). (time- 7n/5)
4. median of the medians is mm
5. Use mm to Partition the (n-1) elements into 3 lists (L,R,M).
   1. L → A[i] < mm
   2. R → A[i] > mm
   3. M → A[i] = mm
6. Find the k-th median recursively:
   - k=r →return m
   - k<r →return k th smallest of set L. ***select* (L, *1..|L|, k*)**
   - k>r→ return k-r th smallest of set R. ***select* (R, *r+1..|R|*, k-|*L*|+*1*)**

Example

8, 33, 17, 51, 57, 49, 35, 11, 25, 37,14, 3,
2, 13, 52, 12, 6, 29, 32, 54, 5, 16, 22, 23, 7

The median is the kth element, k = 25/ 2 = 13 in sorted list

divide into 5 groups	Sort each group
(8, 33, 17, 51, 57)	(8, 17, 33, 51, 57)
(49, 35, 11, 25, 37)	(11, 25, 35, 37, 49)
(14, 3, 2, 13, 52)	(2, 3, 13, 14, 52)
(12, 6, 29, 32, 54)	(6, 12, 29, 32, 54)
(5, 16, 22, 23, 7).	(5, 7, 16, 22, 23).

Note: sorting smaller array takes less time than sorting whole array.
Extract the median of each group M = {33, 35, 13, 29, 16}, sort M,
median of medians mm = 29:

Partition A into three sequences with mm as pivot:

L = {8, 17, 11, 25, 14, 3, 2, 13, 12, 6, 5, 16, 22, 23, 7} size=15

M = {29},

R = {33, 51, 57, 49, 35, 37, 52, 32, 54} size=9

Example continued

L = {8, 17, 11, 25, 14, 3, 2, 13, 12, 6, 5, 16, 22, 23, 7}

- We repeat the same procedure above with L as A.
- select (L, 1..|L|, k)
  - We divide the elements into 3 groups of 5 elements each: (8,17,11,25,14), (3,2,13,12,6), (5,16,22,23,7).
  - Sort each of the group, and find the new set of medians: *M* = {14, 6, 16}.
  - the new median of medians *mm* is 14.
  - Next, partition A into three sequences with pivot mm=14

    L1 = {*8, 11, 3, 2, 13, 12, 6, 5,* 7},

    M1 = {14}

    R1 = {*17, 25, 16, 22,* 23} .
    - Since k=25/2=13 > *(9+1)* = $|L1| + |M1|$, we set *A* = R1 and find the 3rd (3 = 13 – *(9+1)*) element in R1.
    - The algorithm will return 3rd element of R1[3] = 22.
- Thus, the median of the numbers in the given sequence is 22.

Hashing

Hash Tables

- Hashing is used for storing relatively large amounts of data in a table called a hash table.
- Hash table is usually fixed as H-size, which is larger than the amount of data that we want to store.
- We define the load factor (λ) to be the ratio of data to the size of the hash table.
- Hash function maps an item into an

Hash Tables (2)

- Hashing is a technique used to perform insertions, deletions, and finds in constant average time.
- To insert or find a certain data, we assign a key to the elements and use a function to determine the location of the element within the table called hash function.
- Hash tables are arrays of cells with fixed size containing data or keys corresponding to data.
- For each key, we use the hashing function to map key into some number in the range 0 to H-size-1 using hashing function.

Hash Function

- Hashing function should have the following features:
  - Easy to compute.
  - Two distinct key map to two different cells in array (Not true in general) - why?.
  - This can be achieved by using direct-address table where universal set of keys is reasonably small.
  - Distributes the keys evenly among cells.
- One simple hashing function is to use mod function with a prime number.
- Any manipulation of digits, with least complexity and good distribution can be used.

Hash function

- A **hash function** is any **well-defined procedure or mathematical function** that converts a large, possibly variable-sized amount of data into a small data (usually an integer that may serve as an index to an array).

Example of hash function in Java

```
int hash(String key, int tableSize) {
    int hashVal = 0;
    for (int i=0; i < key.length(); i++)
        hashVal += key.charAt(i)
    return hashVal % tableSize;
}
```

Choosing a hash function

- A good has function should satisfy two criteria:
  1. It should be quick to compute
  2. It should minimize the number of collisions

Hashing

Hash tables are used to store data as {<key,value> pairs }, accessible by the key.

Hashing is a method of inserting data into a table.

Tables can be implemented in many ways.

Hashing is used for faster data retrieval : O(1) on average for **insert, lookup, and remove**

Examples

Examples include

- a fixed array (limiting number of elements),
- array of linked lists (potentially unlimited number of elements)

Hash table

- **hash table** (also **hash map**) is a data structure used to implement an associative array, a structure that can map keys to values.
- A hash table uses a hash function to compute an *index* into an array of *buckets* or *slots*, from which the correct value can be found.

Collisions

- Ideally, the hash function should assign to each possible key to a unique bucket, but this ideal is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after it is created).
- Instead, most hash table designs assume that *hash collisions*—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way.

Collision resolution

- When two keys map into the same cell, we get a collision.
- We may have collision in insertion, and need to set a procedure (collision resolution) to resolve it.

Closed Hashing

- If collision, try to find alternative cells within table.
- Closed hashing also known as open addressing.
- For insertion, we try cells in sequence by using incremented function like:
  - $h_i(x) = (hash(x) + f(i)) \bmod H\text{-size}$ $\quad f(0) = 0$
- Function f is used as collision resolution strategy.
- The table is bigger than the number of data.
- Different method to choose function f :
  - Linear probing
  - Quadratic probing
  - Double hashing

Issues

- Other issues common to all closed hashing resolutions:
  - Confusing after deletion.
  - Simpler than open hashing function
  - Good if we do not expect too many collisions.
  - If search is unsuccessful, we may have to search the whole table.
  - Use of large table compare to number of data expected.

- Table size is usually prime to avoid aliasing
- large table size means wasted space
- small table size means more collisions.

Example:

table_size = 12

145 mod 12 =1

92 mod 12 = 8

241 mod 12 = 1 // collision

127 mod 12 = 7

73 mod 12 =1 // collision

145 mod 12 = 10

Hash table of size=12

Most used

- In many situations, hash **tables** turn out to be more efficient than search trees or any other table lookup structure.
- For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets.

Hash Functions

- A *collision* is defined when multiple keys map onto the same table index.
- A good hash function has the following characteristics
  - avoids collisions
  - Spreads keys evenly in the array
  - Inexpensive to compute - must be O(1)
- Hash Functions for Integers

  hash(int key) =abs(key) % table_size
- table_size should be a prime

Universal hashing

- **Universal hashing** (in a randomized algorithm or data structure) refers to selecting a hash function at random from a family of hash functions with a certain mathematical property (see definition below). This guarantees a low number of collisions in expectation, even if the data is chosen by an adversary.

Perfect hashing

- If all keys are known ahead of time, a perfect hash function can be used to create a perfect hash table that has no collisions.
- minimal perfect hashing means every location in the hash table is used.
- Load_factor = data_size /hash_table_size
- Eg. Used by Db, compiler keyword table,

Collision resolution

- *separate chaining*, each bucket is independent, and has some sort of list of entries with the same index.

Separate chaining

- Some chaining implementations store the first record of each chain in the slot array itself

Open addressing

- open addressing, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found
- Deletion is harder

Probing sequence for empty slot

Well-known probe sequences include:

- Linear probing, in which the interval between probes is fixed (usually 1)
- Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
- Double hashing, in which the interval between probes is computed by another hash function

hash table in practice

- **Associative array**, **hash**, **map**, or **dictionary** is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.
- Operations associated with this data type allow:
  - the addition of pairs to the collection
  - the removal of pairs from the collection
  - the modification of the values of existing pairs
  - the lookup of the value associated with a particular key

hash maps in languages

C++:

```
#include <map>
std::map<std::string, std::string> mymap; mymap[“mosh"] = “nitk";
```

C: see glibc, next slide

Java:

```
Map<String, String> mymap = new HashMap<String, String>();
mymap.put(“mosh", “nitk");
```

Perl: $mymap{“mosh”}=“nitk”;

Php: $mymap[“mosh”]=“nitk”;

Python:

```
mymap = {}
mymap["mosh"]="nitk“
```

See http://rosettacode.org/wiki/Associative_arrays

Hashing in C with gcc

- #include <search.h>
- First a hash table must be created using **hcreate**()
- The function **hdestroy**() frees the memory occupied by the hash table that was created by **hcreate**(). After calling **hdestroy**() a new hash table can be created using **hcreate**().
- The **hsearch**() function searches the hash table for an item with the same key as *item*
- On success, **hsearch**() returns a pointer to an entry in the hash table. **hsearch**() returns NULL on error, that is, if *action* is **ENTER** and the hash table is full, or *action* is **FIND** and *item* cannot be found in the hash table.
- Disadvantage: single global hash table
- See http://linux.die.net/man/3/hsearch

Hashing in g++

```
$ g++  -std=c++0x  hash_map4.cpp
$ a.exe
september -> 30
february  -> 28
```

```
1.  #include <iostream>
2.  #include <string>
3.  #include <unordered_map>
4.  using namespace std;
5.  int main() {
6.      unordered_map<string, int> months;
7.      months["february"] = 28;
8.      months["september"] = 30;
9.      cout << "september -> " << months["september"] << endl;
10.     cout << "february  -> " << months["february"] << endl;
11.     return 0;
12. }
```

Custom hashing function in g++

```
#include <iostream>
#include <unordered_map>
#include <string>
#include <functional>
using namespace std;

typedef pair<string,string> Name;
struct hash_name {
    size_t operator( )(const Name &name) const {
        return hash<string>( )(name.first)
                ^ hash<string>( )(name.second);
    }
} ;
```

```
int main(int argc, char* argv[]) {
    unordered_map<Name,int,hash_name> ids;
    ids[Name("Amit", "Gupta")] = 234;
    ids[Name("Alok","Raj")] =  567;
    for ( auto ii = ids.begin() ; ii != ids.end() ; ii++ )
        cout << ii->first.first << " "
            << ii->first.second << " : "
            << ii->second << endl;
    return 0;
}
```

```
$ g++  -std=c++0x  hash_map5.cpp
$ ./a.exe
Amit Gupta : 234
Alok Raj : 567
```

Perl hash tables

- Perl:

```
my %grade;     # declare grade to be a hash table
grade{“Ajay”} = ‘B’; grade{“Ann”}=90;
# Sort by name and print the grades
for my name in sort keys %grade {
  print $name, $grade;
}
grade{“Ann”}++;        # add one to grade of Ann.
delete grade{“Ajay”}; # delete one entry.
```

Bloom filter

- A **Bloom filter** (1970) is a space-efficient probabilistic data structure to test whether an element is a member of a set:
  - Yes, maybe in the set.
    - False positive: could be mistaken.
  - No, not in the set, 100% sure.
- Like a hash-table, with no deletion.

FILTER
STORAGE
Do you have 'key1'?
Filter:
No
Storage:
No
No
Do you have 'key2'?
Filter:
Yes
necessary
disk access
Storage:
Yes
Yes: here is key2
Yes: here is key2
Do you have 'key3'?
False Positive
Filter:
Yes
unnecessary
disk access
Storage:
No
No
No

Bloom filter

- Bloom filter used to speed up answers in a key-value storage system: databases, webservers
- Values are stored on a disk which has slow access times.
- Wastage of disk accesses on a false positive report.
- Overall are much faster.
- Need more memory.

Sorting

Sorting

Input: a list of n elements

Output: the same list of n elements rearranges in anscending (non decreasing) or decending (non increasing) order.

Why sort?

- Sorting is used by many applications
- Sorting is initial step of many algorithms
- Many techniques can be illustrated by studying sorting.
- Sorted items are easier to search.

Sorting

1. Selection sort
2. Insertion sort
3. Bubble sort
4. Merge sort
5. Heap sort
6. Quick sort
7. Radix sort

Ideal sorting algorithm

- Stable: Equal keys aren't reordered.
- Operates in place, requiring O(1) extra space.
- Worst-case O(n·log(n)) key comparisons.
- Worst-case O(n) swaps.
- Adaptive: Speeds up to O(n) when data is nearly sorted or when there are few unique keys.

There is no algorithm that has all of these properties, and so the choice of sorting algorithm depends on the application.

Which is the best sort?

- See how each algorithm operates.
- There is no best sorting algorithm for all inputs.
- Advantages and disadvantages of each algorithm.
- Worse-case asymptotic behavior is not always the deciding factor in choosing an algorithm.
- Show that the initial condition (input order and key distribution) affects performance as much as the algorithm choice.

Comparison of running times

- Searches
  - Linear: n steps
  - Binary: $\log_2 n$ steps
  - Binary search is about as fast as you can get
- Sorts
  - Bubble: n^2 steps
  - Insertion: n^2 steps
- There are other, more efficient, sorting techniques
  - The fastest are heapsort, quicksort, and mergesort
  - These each take take $n * \log_2 n$ steps
  - In practice, quicksort is the fastest, followed by merge sort. qsort is part of standard C.

Bubble sort

- A **bubble sort**, a sorting algorithm that continuously steps through a list, swapping items until they appear in the correct order.
- Note that the largest end gets sorted first, with smaller elements taking longer to move to their correct positions.

Bubble sort complexity

- **Bubble sort** has worst-case and average complexity both $O(n2)$, where n is the number of items being sorted.
- There exist many sorting algorithms with substantially better worst-case or average complexity of $O(n \log n)$.
- Even other $O(n2)$ sorting algorithms, such as insertion **sort**, tend to have better performance than **bubble sort**.
- Therefore, **bubble sort** is not a practical sorting algorithm when n is large.

Bubble sort animation

Bubble sort

- Bubble sort has many of the same properties as insertion sort, but has slightly higher overhead.
- In the case of nearly sorted data, bubble sort takes O(n) time, but requires at least 2 passes through the data (whereas insertion sort requires something more like 1 pass).
- * Stable
- * O(1) extra space
- * O(n2) comparisons and swaps
- * Adaptive: O(n) when nearly sorted
- * ability to detect that the list is sorted

Bubble sort

```
procedure bubbleSort( A : list of sortable items )
   repeat
     swapped = false
     for i = 1 to length(A) - 1 inclusive do:
       if A[i-1] > A[i] then
           swap( A[i-1], A[i] );
           swapped = true
       end if
     end for
   until not swapped
end procedure
```

Bubble sorting example

6 5 3 1 8 7 2 4

INSERTION SORT

- Insertion sort keeps making the **left side** of the array sorted until the whole array is sorted.
- An example of an insertion sort occurs in everyday life while playing cards. To sort the cards in your hand you extract a card, shift the remaining cards, and then insert the extracted card in the correct place.

- Every repetition of insertion sort removes an element from the input data, inserting it into the correct position in the already-sorted list, until no input elements remain. The choice of which element to remove from the input is arbitrary, and can be made using almost any choice algorithm.
- Sorting is typically done in-place. The resulting array after *k* iterations has the property where the first *k* + 1 entries are sorted. In each iteration the first remaining entry of the input is removed, inserted into the result at the correct position, thus extending the result:

Insertion sort animation

Insertion sort

```
void insertSort(int a[], int length) {
    int i, j, value;
    for(i = 1; i < length; i++)  {
        value = a[i];
        // find place for value
        for (j = i - 1;
          j >= 0 && a[j] > value;
          j--){
          a[j + 1] = a[j]; // move up
        }
        a[j + 1] = value;
    }
}
```

Animation, each element left to right is put in its place

Insertion sorting animation

6 5 3 1 8 7 2 4

Properties of Insertion sort

- Stable
- O(1) extra space
- O(n^2) comparisons and swaps
- Adaptive: O(n) time when nearly sorted
- Very low overhead

Insertion sort

- A[i] is inserted in its proper position in the ith iteration in the sorted subarray **`A[1 .. i-1]`**
- In the i-th step, the elements from index i-1 down to 1 are scanned, each time comparing A[i] with the element at the correct position.
- In each iteration an element is shifted one position up to a higher index.
- The process of comparison and shifting continues until:
  - Either an **`element ≤ A[i]`** is found or
  - When all the sorted sequence so far is scanned.
- Then A[i] is inserted in its proper position.

Insertion sort

- Although it is one of the elementary sorting algorithms with O(n^2) worst-case time, insertion sort is the algorithm of choice either when the data is nearly sorted (because it is adaptive) or when the problem size is small (because it has low overhead).

- For these reasons, and because it is also stable, insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort.

Analysis

- Let a_0, ..., a_{n-1} be the sequence to be sorted. At the beginning and after each iteration of the algorithm the sequence consists of two parts:
  - first part a_0, ..., a_{i-1} is already sorted,
  - second part a_i, ..., a_{n-1} is still unsorted (i in 0, ..., n).
- **worst case** occurs when in every step the proper position for the element that is inserted is found at the beginning of the sorted part of the sequence.

The minimum # of element comparisons (best case) occurs when the array is already sorted in non-decreasing order.

In this case, the # of element comparisons is exactly n - 1, as each element $A[i]$, $2 \le i \le n$, is compared with $A[i - 1]$ only.

The maximum # of element comparisons (Worst case) occurs if the array is already sorted in decreasing order and all elements are distinct. In this case, the number is

$$\sum_{i=2}^{n} (i - 1) = \sum_{i=1}^{n-1} (i - 1) = n(n-1)/2$$

This is because each element $A[i]$, $2 \le i \le n$ is compared with each entry in subarray A[1 .. i-1]

➔ **Pros:** Relatively simple and easy to implement.
Cons: Inefficient for large lists.

- **Best case**: $O(n)$. It occurs when the data is in sorted order. After making one pass through the data and making no insertions, insertion sort exits.
- Average case: $\theta(n^2)$ since there is a wide variation with the running time.
- **Worst case**: $O(n^2)$ if the numbers were sorted in reverse order.

Selection sort

- How does it work:
  - first find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.
- *How does it sort the list in a non-increasing order?*
- Selection sort is:
  - The simplest sorting techniques.
  - a good algorithm to sort a small number of elements
  - an incremental algorithm – induction method

- Selection sort is Inefficient for large lists.
- Its runtime is always quadratic

Incremental algorithms → *process the input elements one-by-one and maintain the solution for the elements processed so far.*

Selection Sort Algorithm

Input: An array *A*[1..*n*] of *n* elements.

Output: *A*[1..*n*] sorted in nondecreasing order.

```
for i ← 1 to n - 1
    k ← i
    for j ← i + 1 to n  {Find the i th smallest element.}
      if A[j] < A[k] then k ← j
    end for
  if k ≠ i then interchange A[i] and A[k]
end for
```

Example

-
original array
76 3 94 55 21 1
find the smallest number and swap with 76
1 3 94 55 21 76
A B
find the smallest number and swap with 3
sorted
1 3 94 55 21 76
A B
find the smallest number and swap with 94
sorted
1 3 21 55 94 76
A B
find the smallest number and swap with 55
sorted
1 3 21 55 94 76
A B
find the smallest number and swap with 94
sorted
1 3 21 55 76 94
A B
sorted array
1 3 21 55 76 94

Analysis of Algorithms

- *T*(*n*) is the total number of accesses made from the beginning of selection_sort until the end.
- selection_sort itself simply calls swap and find_min_index as *i* goes from 1 to *n*-1

$$T(n) = \sum_{i=1}^{n-1} find - \min - element + swap$$

= n-1 + n-2 + n-3 + … + 1 = n(n-1)/2

Or = ∑ (n - i) = n (n - 1) / 2 → O(n²)

Properties

- Not stable
- O(1) extra space
- Θ(n^2) comparisons
- Θ(n) swaps
- Not adaptive

Selection sort animation

Merge sort

- Merge sort (also commonly spelled mergesort) is an O(n log n) comparison-based sorting algorithm.
- Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output.
- Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945

How it works

- Conceptually, a merge sort works as follows
- 1. Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
- 2. Repeatedly merge sublists to produce new sublists until there is only 1 sublist remaining. This will be the sorted list.

Merge sort animated example

6 5 3 1 8 7 2 4

Algorithm: MERGE

```
// from wikipedia
function merge(left, right)
    var list result
    while length(left) > 0 or length(right) > 0
        if length(left) > 0 and length(right) > 0
            if first(left) <= first(right)
                append first(left) to result
                left = rest(left)
            else
                append first(right) to result
                right = rest(right)
        else if length(left) > 0
            append first(left) to result
            left = rest(left)
        else if length(right) > 0
            append first(right) to result
            right = rest(right)
    end while
    return result
```

Merge Sort divide-and-conquer

- **Divide:** divide the *n*-element sequence into two subproblems of *n*/2 elements each.
- **Conquer**: sort the two subsequences recursively using merge sort. If the length of a sequence is 1, do nothing since it is already in order.
- **Combine**: merge the two sorted subsequences to produce the sorted answer.

MERGE-SORT(A,*p*,*r*)

```
if lo < hi
then mid ← ⌊(lo+hi)/2⌋
          MERGE-SORT(A,lo,mid)
          MERGE-SORT(A,mid+1,hi)
          MERGE(A,lo,mid,hi)
```

Call MERGE-SORT(A,1,*n*) (assume *n*=length of list A)

A = {10, 5, 7, 6, 1, 4, 8, 3, 2, 9}

Merge sort EXAMPLE from wikipedia

Analysis of Merge Sort

```
1.  if lo < hi                                  ............  1
2.  then mid ← ⌊(lo+hi)/2⌋                      ............. 1
3.           MERGE-SORT(A,lo,mid)               ............. n/2
4.           MERGE-SORT(A,mid+1,hi)              ............ n/2
5.           MERGE(A,lo,mid,hi)                  .............n
```

- Described by recursive equation
- *Let* $T(n)$ be the running time on a problem of size n.
- $T(n) = \begin{cases} c & \text{if } n=1 \\ 2T(n/2)+cn & \text{if } n>1 \end{cases}$

$O(n \log n)$

Merge sort animation

Quick sort

- The quicksort algorithm was developed in 1960 by Tony Hoare while in the Soviet Union, as a visiting student at Moscow State University.
- At that time, Hoare worked in a project on machine translation for the National Physical Laboratory.
- He developed the algorithm in order to sort the words to be translated, to make them more easily matched to an already-sorted Russian-to-English dictionary that was stored on magnetic tape

Algorithm

- Quicksort is a divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.
- The steps are:
- Pick an element, called a *pivot*, from the list.
- Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
- Recursively sort the sub-list of lesser elements and the sub-list of greater elements.
- The base case of the recursion are lists of size zero or one, which never need to be sorted.

Psuedo code

```
function quicksort('array')
    if length('array') ≤ 1
        return 'array'  // an array of zero or one elements is already sorted
    select and remove a pivot value 'pivot' from 'array'
    create empty lists 'less' and 'greater'
    for each 'x' in 'array'
        if 'x' ≤ 'pivot' then append 'x' to 'less'
        else append 'x' to 'greater'
    return concatenate(quicksort('less'), 'pivot',
quicksort('greater')) // two recursive calls
```

6 5 3 1 8 7 2 4

Properties

- Not stable
- O(lg(n)) extra space
- O(n2) time, but typically O(n·lg(n)) time
- With both sub-sorts performed recursively, quick **sort** requires O(n) extra space for the recursion stack in the worst case when recursion is not balanced.
- To make sure at most O(log N) space is used, recurse first into the smaller half of the array, and use a tail call to recurse into the other
- **Quicksort with 3-way partitioning**.
- Use insertion **sort**, which has a smaller constant factor and is thus faster on small arrays.

Pivot selection

- In very early versions of quicksort, the leftmost element of the partition would often be chosen as the pivot element.
- Unfortunately, this causes worst-case behavior on already sorted arrays, which is a rather common use-case.
- The problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or (especially for longer partitions) choosing the **median** of the first, middle and last element of the partition for the pivot [Sedgewick].

Quick sort animation

Quick sorting example

Divide and Conquer

An algorithm design technique

1. **Divide: the instance (problem) into a number of subinstances (in most cases 2).**
2. **Conquer: the subinstance by solving them separately.**
3. **Combine: the solutions to the subinstances to obtain the solution to the original problem instance.**

Binary Search

Input: An array A[*1..*n] of *n* elements sorted in nondecreasing order and an element x.
Output: *j* if *x* = A[j], 1 ≤ *j* ≤ *n*, and 0 otherwise.

1*. binarysearch*(*1,* n)

Procedure *binarysearch*(*low, high*)

1. if *low* > *high* then return 0
2. else
3. *mid* ← (low + high)/2
4. if *x* = A[*mid*] then return *mid*
5. else if *x* < A[*mid*] then return *binarysearch*(*low,mid*-1)
6. else return *binarysearch*(*mid* + *1, high*)
7. end if

C(n) is the number of comparisons performed by Algorithm BINARYSEARCHREC in the worst case on an array of size n.

$$C(n) \le \begin{cases} 1 & \text{if } n=1 \\ 1+C(\lfloor n/2 \rfloor) & \text{if } n \ge 2. \end{cases}$$

Binary search is an example of an O(logn) algorithm. Thus, twenty comparisons suffice to find any name in the million-name list

MERGESORT

1. Divide the whole list into 2 sublists of equal size;
2. Recursively merge sort the 2 sublists;
3. Combine the 2 sorted sublists into a sorted list.

Algorithm MERGESORT

Input: **An array A[*1..n*] of *n* elements.**

Output: ***A*[1*..n*] sorted in nondecreasing order.**

***mergesort*(*A*, 1, *n*)**

Procedure *mergesort*(A, *low, high*)

1. if *low* < *high* then		
2.	***mid* ←(low + high) / 2**	**T(1)**
3.	***mergesort*(*A, low, mid*)**	**T(n/2)**
4.	***mergesort*(*A, mid + 1, high*)**	**T(n/2)**
5.	**MERGE *(A, low, mid, high*)**	**T(n)**
6. end if		

Use: *Linear-time merge subroutine.*

Analysis for Merge Sort

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

Quick sort

- Divide-and-conquer algorithm.
- Sorts “in place” (*like insertion sort, but not like merge sort).*

Quicksort an *n*-element array:

1. ***Divide:*** Partition the array into two subarrays around a ***pivot*** *x* such that elements in lower subarray ≤ *x* ≤ elements in upper subarray.
- ***2. Conquer:*** Recursively sort the two subarrays.
- ***3. Combine:*** Nothing

Use: *Linear-time partitioning subroutine.*

Partitioning places all the elements less than the pivot in the *left* part of the array, and all elements greater than the pivot in the *right* part of the array. The pivot fits in the slot between them.

Partitioning

Partition a list A[] into two non-empty parts.

Pick any value in the array, pivot.

left = the elements in A[] ≤ pivot

right = the elements in A[] > pivot

Place The pivot in the slot between them.

An illustration of one step partitioning

```
procedure Quicksort(A, p, r)
 if p < r then
     q ← Partition(A, p, r)
     Quicksort(A, p, q − 1)
     Quicksort(A, q +1, r)
```

```
Partition( A[ ], first, last)
//Define the pivot value as the contents of A[First]
P = A[first] ; Q=first;
//Initialize Up to First and Down to Last
Up = first ; down = last;
Repeat
    Repeat up++     until (A[up] > P)
    repeat down- -  until (A[down] ≤ P)
    if (up < Down)
            exchange (A[up] , A[down] )
until (up ≥ Down )
Exchange (A[First] , A[Down])
Q= Down
Return Q
```

The effects of the linear time partitioning step:

1. The pivot element ends up in the position it retains in the final sorted order.
2. After a partitioning, no element flops to the other side of the pivot in the final sorted order.

Thus *we can sort the elements to the left of the pivot and the right of the pivot independently!*

Quicksort– best case

The best case for *divide-and-conquer* algorithms comes when we split the input as evenly as possible. Thus in the best case, each subproblem is of size *n*/2.

The partition step on each subproblem is linear in its size. Thus the total effort in partitioning the problems of size is *O*(*n*).

The total partitioning on each level is *O*(*n*), and it take log n levels of partitions to get to single element subproblems.

When we are down to single elements, the problems are sorted. Thus the total time in the best case is O(n log n) .

The recursion tree for the best case looks like this

Quicksort - Worst Case

- Worst case: the pivot chosen is the largest or smallest value in the array. Partition creates one part of size 1 (containing only the pivot), the other of size n-1.
- Now we have *n-1* levels, instead of logn , for a worst case time of $O(n^2)$

Heap sort

Heaps and Priority Queues

What is Heap?

An array Representing an *almost complete binary tree:*

- All levels filled except possibly the last
- Last level filled from the left to a point

Let A be an array that implements the heap:

- **`Root of the tree is A[1].`**
- **`Node` *`i`* `is A[`*`i`*`]`**
- **`Parent(i) { return ⌊i/2⌋; }`**
- **`Left(i) { return 2*i; }`**
- **`right(i) { return 2*i + 1; }`**

Heap can be seen as a complete binary tree think of unfilled slots as null pointers

A=

16	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*

- *What is the height of an n-element heap? Why?*

Heap Property

$A[PARENT(i)] \geq A[i]$ for all nodes $i > 1$

- the value of a node is at most the value of its parent
- the largest element in the heap is stored in the root

- **<u>Operations on Heaps</u>**
  - ***delete-max*[H]:** Delete and return an item of maximum key from a nonempty heap **H.**
  - ***insert*[*H*,x]:** Insert item ***x*** into heap **H**.
  - ***delete*[*H*,i]:** Delete the **ith** item from heap **H.**
  - ***makeheap*[A]:** Transform array ***A*** into a heap.
  - **HEAPSORT(A):** sorts an array in place.

- Before describing the main heap operations, let us first present two secondary heap operations:
  - **Shift-up**
  - **Shift-down**
- *These are used subroutines in the algorithms that implement heap operations.*

Shift-up

➢**Suppose the key stored in the 10th position of the heap shown in is changed from 5 to 25.**

move H[i] up along the *unique* path from H[i] to the root until its proper location along this path is found.

At each step along the path, the key of H[i] is compared with the key of its parent H[ëi/2û].

Shift-down

➢**Suppose we <u>change the key 17</u> stored in the second position of the heap <u>shown in Fig. 1</u> to <u>3</u>.**

➢**At each step along the path, its key is compared with the maximum of the two keys stored in its children nodes (if any).**

Algorithm MAKEHEAP

Input: An array $A[1..n]$ of n elements.

Output: $A[1..n]$ is transformed into a heap

```
for i ← n/2 downto 1
    SHIFT-DOWN(A , i)
end for
```

Make heap animation

Let { 6, 5, 3, 1, 8, 7, 2, 4 } be the list that we want to sort from the smallest to the largest. (NOTE, for 'Building the Heap' step: Larger nodes don't stay below smaller node parents. They are swapped with parents, and then recursively checked if another swap is needed, to keep larger numbers above smaller numbers on the heap.

6 5 3 1 8 7 2 4

Heap sort Algorithm

- Heapsort is a two step algorithm.
- The first step is to build a heap out of the data.
- The second step begins with removing the largest element from the heap.
- We insert the removed element into the sorted array.
- For the first element, this would be position 0 of the array.
- Next we reconstruct the heap and remove the next largest item, and insert it into the array.
- After we have removed all the objects from the heap, we have a sorted array.

Heap sort properties

- Heapsort is not a stable **sort**; merge **sort** is stable.
- The heapsort algorithm itself has $O(n \log n)$ time complexity

Algorithm

```
function heapify(a,count) is
    // (end is assigned the index of the first (left) child of the root)
    end := 1

    while end < count
        // (sift up the node at index end to the proper place such that all nodes above
        // the end index are in heap order)
        siftUp(a, 0, end)
        end := end + 1
    // (after sifting up the last node all nodes are in heap order)

function siftUp(a, start, end) is
    input:  start represents the limit of how far up the heap to sift.
                end is the node to sift up.
    child := end
    while child > start
        parent := floor((child - 1) / 2)
        if a[parent] < a[child] then (out of max-heap order)
            swap(a[parent], a[child])
            child := parent (repeat to continue sifting up the parent now)
        else
            return
```

heap animation

Let { 6, 5, 3, 1, 8, 7, 2, 4 } be the list that we want to sort from the smallest to the largest. (NOTE, for 'Building the Heap' step: Larger nodes don't stay below smaller node parents. They are swapped with parents, and then recursively checked if another swap is needed, to keep larger numbers above smaller numbers on the heap.

6 5 3 1 8 7 2 4

heap sort animation

Radix sort

- **Radix sort** is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.
- A positional notation is required, but because integers can represent strings of characters (e.g., names or dates) and specially formatted floating point numbers.
- Radix **sort** is not limited to integers.
- Radix **sort** dates back as far as 1887 to the work of Herman Hollerith on tabulating machines

Radix sort is fast and stable

- Radix **sort** is a fast stable sorting algorithm which can be used to **sort** keys in integer representation order.
- The complexity of radix **sort** is linear, which in terms of omega means O(n).
- That is a great benefit in performance compared to O(n.log(n)) or even worse with O(n^2)

Radix sort is linear

Example of radix sorting

1. Unsorted list:

- 523
- 153
- 088
- 554
- 235

2. On Radix 0 (1st digit)

- 523
- 153
- 554
- 235
- 088
- ^

Sort on Radix 1 (2nd. digit)

- 523
- 235
- 153
- 554
- 088

Sort on Radix 2 (3rd digit)

- 088
- 153
- 235
- 523
- 554
- ^

Example of radix sort

1. Original, unsorted list:

   170, 45, 75, 90, 802, 24, 2, 66

2. Sorting by least significant digit (1s place) gives: [we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

   17<u>0</u>, 9<u>0</u>, 80<u>2</u>, <u>2</u>, 2<u>4</u>, 4<u>5</u>, 7<u>5</u>, 6<u>6</u>

continued

Example continued

3 Sorting by next digit (10s place) gives: [Note: 802 again comes before 2 as 802 comes before 2 in the previous list.]

: 8<u>0</u>2, 2, <u>2</u>4, <u>4</u>5, <u>6</u>6, 1<u>7</u>0, <u>7</u>5, <u>9</u>0

4. Sorting by most significant digit (100s place) gives:

: 2, 24, 45, 66, 75, 90, <u>1</u>70, <u>8</u>02

Radix sort is single pass

- It is important to realize that each of the above steps requires just a single pass over the data,
- since each item can be placed in its correct bucket without comparing with other items.

Topological Sort

- Sorting vertices in a graph
- Not unique
- Not usual meaning of sort
- Graph must be directed acyclic graph (DAG) (no cyles).
- Used in scheduling a sequence of jobs or tasks based on their dependencies

Introduction

- There are many problems involving a set of tasks in which some of the tasks must be done before others.
- For example, consider the problem of taking a course only after taking its prerequisites.
- Is there any systematic way of linearly arranging the courses in the order that they should be taken?

Yes! - Topological sort.

Definition of Topological Sort

- Topological sort is a method of arranging the vertices in a directed acyclic graph (DAG), as a sequence, such that no vertex appear in the sequence before its predecessor.

- The graph in (a) can be topologically sorted as in (b)

(a) (b)

Topological Sort is not unique

- Topological sort is not unique.
- The following are all topological sort of the graph below:

s1 = {a, b, c, d, e, f, g, h, i}

s2 = {a, c, b, f, e, d, h, g, i}

s3 = {a, b, d, c, e, g, f, h, i}

s4 = {a, c, f, b, e, h, d, g, i}
etc.

Topological sort algorithm

```
Input graph G
Result ← { }
S ← { Set of all nodes with no incoming edges }
while S is non-empty do
    Move n from S to Result
    for each child m of n do:
        delete edge e (n to m)
        if m is root (no more incoming edges) then
            add m to S
Finally
if G still has edges then
    return error (G was cyclic)
else
    return Result (a topologically sorted order)
```

DFS based topological sort

```
result = {  };
roots = {nodes with no parent}
for n in roots:
    dfs(n)
function visit(n)
    if n is not-visited then
        mark n as visited
        for child m of n do:
            visit(m)
        add n to result;
```

Topological Sort: DFS

S ← Set of all nodes with no incoming edges = { c }

Dfs(c)

Topological Sort: DFS

dfs(A)

Topological Sort: DFS

```
dfs(A)
  dfs(D)
```

Topological Sort: DFS

```
dfs(A)
  dfs(D)
    dfs(E)
```

Topological Sort: DFS

```
dfs(A)
  dfs(D)
    dfs(E)
      dfs(F)
```

Topological Sort: DFS

```
dfs(A)
  dfs(D)
    dfs(E)
      dfs(F)
        dfs(H)
```

Topological Sort: DFS

```
dfs(A)
  dfs(D)
    dfs(E)
      dfs(F)
```

Topological Sort: DFS

```
dfs(A)
  dfs(D)
    dfs(E)
```

Topological Sort: DFS

```
dfs(A)
  dfs(D)
```

Topological Sort: DFS

```
dfs(A)
  dfs(D)
```

Topological Sort: DFS

dfs(A)

Topological Sort: DFS

dfs(A)

Topological Sort: DFS

Topological Sort: DFS

dfs(B)

Topological Sort: DFS

dfs(B)

Topological Sort: DFS

Topological Sort: DFS

dfs(C)

Topological Sort: DFS

dfs(C)

Topological Sort: DFS

dfs(C)

Topological Sort: DFS

dfs(C)

Topological Sort: DFS

```
dfs(C)
  dfs(G)
```

Topological Sort: DFS

dfs(C)

Topological Sort: DFS

Topological Sort: DFS

Topological order: **C G B A D E F H**

Tree

Data Structure

Tree

- **Is a connected acylic graph**
- **Has a Root**
- **Each node may have subtrees**
- **Branching factor: number of children**

Tree vocabulary

- **Root:** A (only node with no parent)
- **Children**(B) = {E, F}
- **Siblings**(X) = {Nodes with the same parent as excluding X}
- **Siblings**(B) = {C, D}, Siblings(A) = { }
- **Descendants**(X) = {Nodes below X}
- **Descendants**(A) = {B,C,D,E,F,G}
- **Ancestors**(X) = {Nodes between X and the root}
- **Ancestors**(E) = {B, A}
- Nodes with no children are called
- **leaves** or **external or *terminal* nodes**: {C, E, F, G}
- **Internal nodes**: Nodes with children (vertices other than leaves {A, B, D})
- The **subtree rooted at X** is the tree of all descendents of X, including X.

Tree

- *Depth or Level of a vertex* is length of the path from the root to the vertex.
  - Root depth = 0
  - Depth(x) = number of ancestors of x.
  - Depth(x) = 1 + Depth(parent(x))
- Height of a node x is the length of the longest path from the vertex to a leaf.
- Height of a tree is Height of root

Path

Path to a node x is a sequence of nodes from root to x. How many paths are there to a node?

Height of a node is the length of the LONGEST path from the node to its leaves.

Height of a leaf is 0

Tree Traversals

- visit all nodes of a tree, starting from the root. *Use recursion*
  - each element of the tree is visited exactly once.
  - visit of an element, indicates ***action*** *(print value, evaluate the operator, etc.)*
- Pre-order traversal: (root ➔left ➔ right)
- In-order traversal: (left ➔ root ➔ right)
- Post-order traversal: (left ➔ right ➔ root)

Example of recursive function for preorder tree traversal.

```
preOrder(T) {
    if(! T) return;
    visit(T);
     preOrder(T.left);
     preOrder(T.right);
}
```

Pre-order traversal: (root ➔left ➔ right)
In-order traversal: (left ➔ root ➔ right)
Post-order traversal: (left ➔ right ➔ root)

Preorder: ABCDEFGHI

Postorder: CBFGEIHDA

Inorder: BCAFEGDIH

Traversal example

Pre-Order	ABDHIECFGJ
In-Order	HDIBEAFCJG
Post-Order	HIDEBFJGCA

Tree Traversals

Pre-Order: A-B-D-C-E-F **In-Order**: B-D-A-E-C-F **Post-Order**: D-B-E-F-C-A

Binary tree data structure

Binary Tree

- **Binary Tree** is a **tree data structure** in which each node has at most two **children**. *the child nodes are called left and right*

- **Full Binary Tree:** If each internal node has exactly two children.
- **Complete Binary Tree:** If it is full and all its leaves are on the same level (i.e. have the same depth)

Almost Complete Binary Tree (ACBT)

- complete binary tree.

Or

- complete binary tree and conditions:
  - All levels are complete except the lowest one.
  - In the last level empty spaces are towards the right.

- If Binary tree has height **k**, then it is between ***O(log n) and n-1***
- ***The <u>height of a complete or almost-complete</u> binary tree with n vertices is ⌊log n⌋.***
- ***In a <u>full binary tree</u>, the number of leaves is equal to the number of internal vertices plus one.***

Height of trees

Let n be the number of vertices in a binary tree T of height k.

If *T* is complete then:

$$n \leq \sum_{j=0..k} 2^j = 2^{(k+1)}-1.$$

If *T* is almost-complete then:

$$2^k \leq n \leq 2^{(k+1)} -1$$

Storage of Binary Trees in arrays

- **Resulting from sequentially numbering the nodes**
- Start with the root at **1**
  - **`Parent(i)  at int(i/2)`**
  - **`Left(i)  at 2*i`**
  - **`right(i)  at 2*i + 1`**

binary search tree

A binary search tree is a binary tree such that: If x is a node with key value key[x], If each node of a tree has the following *Binary Search Tree properties*:

- for all nodes y in left subtree of x, key[y] ≤ key[x]
- for all nodes y in right subtree of x, key[y] ≥ key[x]

Fig. 1

Special Trees

- Red black
- Kd trees
- B-trees
- 2-3 trees
- AVL trees
- Fibonacci

Type of Trees

Binary search tree (BST)

- BST also called an **ordered** or **sorted binary tree**, is a node-based binary **tree** data structure which has the following properties:
  - The left subtree of a node contains only nodes with keys less than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - Both the left and right subtrees must also be binary search trees.
  - There must be no duplicate nodes.

BST

- The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.
- Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

Optimal binary search ***tree***

- If we do not plan on modifying a search **tree**, and
- we know exactly how often each item will be accessed,
- we can construct an *optimal binary search* ***tree***, which is a search **tree**
- where the average cost of looking up an item (the *expected search cost*) is minimized.

Binary tree get unbalanced (O(n) to access nodes)

Balanced tree for faster (O(log n) to access nodes)

Balancing trees

- Recall that, for binary-search trees, although the average-case times for the lookup, insert, and delete methods are all O(log N), where N is the number of nodes in the **tree**, the worst-case time is O(N).
- We can **guarantee** O(log N) time for all three methods by using a **balanced tree** -- a **tree** that always has height O(log N)-- instead of a binary-search **tree**.
  - A number of different balanced trees have been defined, including **AVL trees**, **red-black trees**, and **B trees**.

Balanced trees

- Balanced search trees are found in many flavors and have been the major data structure used for structures called dictionaries, where insertion, deletion, and searching must take place.
- In 1970, John Hopcroft introduced 2-3 search trees as an improvement on existing balanced binary trees.
- Later they were generalized to B-trees by Bayer and McCreight.
- B-trees were then simplified by Bayer to form red-black trees.

Different types of trees

V · T · E	Trees in computer science
Binary trees	Binary search tree (BST) · Cartesian tree · MVP Tree · Top tree · T-tree
Self-balancing binary search trees	AA tree · AVL tree · LLRB tree · Red–black tree · Scapegoat tree · Splay tree · Treap
B-trees	B+ tree · B*-tree · B^X-tree · UB-tree · 2-3 tree · 2-3-4 tree · (a,b)-tree · Dancing tree · Htree
Tries	Suffix tree · Radix tree · Ternary search tree · X-fast trie · Y-fast trie
Binary space partitioning (BSP) trees	Quadtree · Octree · *k*-d tree · Implicit *k*-d tree · VP tree
Non-binary trees	Exponential tree · Fusion tree · Interval tree · PQ tree · Range tree · SPQR tree · Van Emde Boas tree
Spatial data partitioning trees	R-tree · R+ tree · R* tree · X-tree · M-tree · Segment tree · Hilbert R-tree · Priority R-tree
Other trees	Heap · Hash tree · Finger tree · Order statistic tree · Metric tree · Cover tree · BK-tree · Doubly chained tree · iDistance · Link-cut tree · Fenwick tree

Heap

A **heap** is a tree-based data structure that satisfies the *heap property:* If *A* is a parent node of *B* then key(*A*) is ordered with respect to key(*B*) with the same ordering applying recursively.

A *heap* data structure should not be confused with *the heap* which is a common name for dynamically allocated memory.

Max heap

Skip lists (a probabilistic alternative to balanced trees)

- A **skip list** is a data structure for storing a sorted **list** of items using a hierarchy of linked lists that connect increasingly sparse subsequences of the items. These auxiliary lists allow item lookup with efficiency comparable to balanced binary search trees (that is, with number of probes proportional to log *n* instead of *n*)

Skip list

- *Skip lists are a probabilistic data structure that seem likely to supplant balanced trees as the implementation method of choice for many applications.*
- *Skip **list** algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.*

Binomial tree

- A binomial heap is implemented as a collection of binomial trees. A **binomial tree** is defined recursively: A binomial tree of order 0 is a single node
  - A binomial tree of order *k* has a root node whose children are roots of binomial trees of orders *k*−1, *k*−2, ..., 2, 1, 0 (in this order).

Binomial heap

- Supports quick merging of two heaps.
- A binomial heap is implemented as a collection of binomial trees (compare with a binary heap single binary tree).
- A **binomial tree** is defined recursively:
  - A binomial tree of order 0 is a single node
  - A binomial tree of order *k* has a root node whose children are roots of binomial trees of orders *k*−1, *k*−2, ..., 2, 1, 0 (in this order).
- Properties
  - A binomial tree of order *k* has 2^k nodes, height *k*.
  - binomial heap with *n* nodes consists of at most log *n* + 1 binomial trees

Example of binomial trees

- Binomial trees of order 0 to 3:
- Each tree has a root node with subtrees of all lower ordered binomial trees, which have been highlighted.
- For example, the order 3 binomial tree is connected to an order 2, 1, and 0

  (highlighted as blue, green and red respectively) binomial tree.

Example of binomial heap

- *Example of a binomial heap containing 13 nodes with distinct keys.*
- *The heap consists of three binomial trees with orders 0, 2, and 3.*

Binomial heap

- A binomial heap is implemented as a set of binomial trees that satisfy the *binomial heap properties*:
- Each binomial tree in a heap obeys the *minimum-heap property*: the key of a node is greater than or equal to the key of its parent.
- There can only be either *one* or *zero* binomial trees for each order, including zero order.
- Because no operation requires random access to the root nodes of the binomial trees, the roots of the binomial trees can be stored in a linked list, ordered by increasing order of the tree.

Fibonacci heaps (FH)

- Invented by Fredman and Robert E. Tarjan in 1984 for use in graph algorithms.
- Using Fibonacci heaps for priority queues improves the asymptotic running time of important algorithms, such as Dijkstra's algorithm for computing the shortest path between two nodes in a graph.

Fibonacci heap

- FH is a collection of **trees** satisfying the minimum-heap property (the key of a child is always greater than or equal to the key of the parent).
- The minimum key is always at the root of one of the **trees**.

Lazy balancing of FH

- FH can be balanced lazily (postponing the work for later operations).
- As a result of a relaxed structure, some operations can take a long time while others are done very quickly.

Fibonacci heap properties

- every node has degree at most $O(\log n)$
- Size of subtree rooted in a node of degree k is at least $F[k + 2]$, where $F[k]$ is the kth Fibonacci number.

Uses of Fibobacci heaps

Fibonacci Heaps

Fibonacci heap history. Fredman and Tarjan (1986)

- Ingenious data structure and analysis.
- Original motivation: O(m + n log n) shortest path algorithm.
  - also led to faster algorithms for MST, weighted bipartite matching
- Still ahead of its time.

Fibonacci heap intuition.

- Similar to binomial heaps, but less structured.
- Decrease-key and union run in O(1) time.
- "Lazy" unions.

Cost of operations on heaps

Priority Queues

		Heaps		
Operation	Linked List	Binary	Binomial	Fibonacci †
make-heap	1	1	1	1
insert	1	log N	log N	1
find-min	N	1	log N	1
delete-min	N	log N	log N	log N
union	1	N	log N	1
decrease-key	1	log N	log N	1
delete	N	log N	log N	log N
is-empty	1	1	1	1

† amortized

Tarjan says

- Once you successfully write a complicated program, it always runs as is. The computer doesn’t have to understand the algorithm, it just follow the steps.
- E.g. Scientists are still using fast fortran programs from 1950s.

R. E. Tarjan

2-3 Tree

- **2–3 tree** is a type of data structure, a **tree** where every node with children (internal node) has either two children (2-node) and one data element or three children (3-nodes) and two data elements.

Rules for 2-3 search trees

- ***All data appears at the leaves.***
- ***Data elements are ordered from left (minimum) to right (maximum).***
- ***Every path through the tree is the same length.***
- ***Interior nodes have two or three subtrees.***

Example of 2-3 tree

Tree rotation

Rotate to balance

- A tree can be rebalanced using rotations.
- After a rotation, the side of the rotation increases its height by 1 whilst the side opposite the rotation decreases its height similarly.
- Therefore, one can strategically apply rotations to nodes whose left child and right child differ in height by more than 1.
- Self-balancing binary search trees apply this operation automatically.
- A type of tree which uses this rebalancing technique is the AVL tree, Red-Black, Splay trees.

Splay Trees

- A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again.
- It performs basic operations such as insertion, look-up and removal in O(log n) amortized time.
- For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.
- Invented by Daniel Dominic Sleator and Robert Endre Tarjan in 1985

Splay trees

- First perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top.
- Simple implementation—simpler than other self-balancing binary search trees, such as red-black trees or AVL trees.
- Perhaps the most significant disadvantage of splay trees is that the height of a splay tree can be linear.

AVL Tree

- Balance requirement for an AVL **tree**: the left and right sub-trees differ by at most 1 in height.

Rotation rebalance the trees

AVL tree

- **AVL tree** is a self-balancing binary search **tree**
- In an AVL **tree**, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.
- Lookup, insertion, and deletion all take O(log *n*) time in both the average and worst cases, where n is the number of nodes in the **tree** prior to the operation.
- Insertions and deletions may require the **tree** to be rebalanced by one or more **tree** rotations.

Rebalancing AVL by rotations

B-Tree

- **B-tree** is a tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
- The B-tree is a generalization of a binary search tree in that a node can have more than two children

B-tree of order 2

B-Tree

- In order to maintain the pre-defined range, internal nodes may be joined or split.
- Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees,
- but may waste some space, since nodes are not entirely full.

B-Tree

- Each internal node of a B-tree will contain a number of keys.
- The keys act as separation values which divide its subtrees

Example of a B-tree of order 5

- This means that (other that the root node) all internal nodes have at least ceil(5 / 2) = ceil(2.5) = 3 children (and hence at least 2 keys).
- The maximum number of children that a node can have is 5 (so that 4 is the maximum number of keys).
- Each leaf node must contain at least 2 keys.
- In practice B-trees usually have orders a lot bigger than 5.

B+ Trees

- The primary value of a B+ **tree** is in storing data for efficient retrieval in a block-oriented storage context—in particular, filesystems.
- The B+-**tree** is used as a (dynamic) indexing method in relational database management systems.
- This is primarily because unlike binary search trees, B+ trees have very high fanout (typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the **tree**.

B+ Tree example

B+ Tree

- variant of the original B-**tree** in which all records are stored in the leaves and all leaves are linked sequentially.

Red-black tree

- A **red–black tree** is a type of self-balancing binary search **tree**.
- The self-balancing is provided by painting each node with one of two colors (these are typically called 'red' and '**black**', hence the name of the trees) in such a way that the resulting painted **tree** satisfies certain properties that don't allow it to become significantly unbalanced.
- When the **tree** is modified, the new **tree** is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be made efficiently.

Example of RB tree

Rbtree

- Tracking the color of each node requires only 1 bit of information per node because there are only two colors.
- The **tree** does not contain any other data specific to it being the red–**black tree** so its memory footprint is almost identical to classic (uncolored) binary search **tree**.
- In many cases the additional bit of information can be stored at no additional memory cost.

Properties

- A node is either red or **black**.
- The root is **black**. (This rule is sometimes omitted. Since the root can always be changed from red to **black**, but not necessarily vice-versa, this rule has little effect on analysis.)
- All leaves (NIL) are **black**. (All leaves are same color as the root.)
- Both children of every red node are **black**.
- Every simple path from a given node to any of its descendant leaves contains the same number of **black** nodes.
- These constraints enforce a critical property of red–**black** trees: that the path from the root to the furthest leaf is no more than twice as long as the path from the root to the nearest leaf.

Uses

- C++ STL Library: Map, Multimap, Set, Multiset
- Java's TreeMap class
- IBM's ISAM (Indexed Sequential Access Method)

- Red–black trees offer worst-case guarantees for insertion time, deletion time, and search time:
- Real-time applications
- Computational geometry
- Completely Fair Scheduler in Linux kernel

Hash is 2x faster than STL map.

Hash is unordered set.

Map is ordered set.

Trie (**prefix tree**)

- A **trie**, (re**trie**val**)**, is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings.
- Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated.
- All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string.
- Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest.

Trie (**prefix tree**)

- A **trie**, (re**trie**val**)**, is an ordered tree data structure for associative array with strings keys.
- E.g. a trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn".

Suffix tree

- Used for fast string matching using Ukkonen's algorithm
- Constructing Suffix Tree(S) takes linear time and linear space in length(S).
- Once constructed, searching is fast.

Example: Suffix tree for “BANANA”

- The six paths from the root to a leaf (boxes) are the six suffixes A$, NA$, ANA$, NANA$, ANANA$ and BANANA$ of
- The numbers in the leaves give the start position of the corresponding suffix.
- Suffix links drawn dashed.

Uses of Suffix Tree

- String search, in *O(m)* complexity, where *m* is the length of the sub-string (but with initial *O(n)* time required to build the suffix tree for the string)
- Finding the longest repeated substring
- Finding the longest common substring
- Finding the longest palindrome in a string.
- Bio-Informatics – DNA matching.
- search efficiently with mismatches.

Suffix tree

- The paths from the root to the leaves have a one-to-one relationship with the suffixes of S.
- Edges spell non-empty strings,
- Internal nodes (except perhaps the root) have at least two children.
- S padded with ‘$’ to mark end of string. This ensures that no suffix is a prefix of another, and that there will be n leaf nodes, one for each of the n suffixes of S.

- Generalized suffix tree is a suffix tree made for a set of words instead of only for a single word:
  - It represents all suffixes from this set of words.
  - Each word must be terminated by a different termination symbol or word.

Splay Tree Demo

Splay Trees

- At the end of each operation a special step called **splaying** is done.
- The splay operation moves x to the root of the tree.
- The splay operation consists of sub-operations called zig-zig, zig-zag, and zig.
- Splaying ensures that all operations take O(lg n) amortized time.

Zig-Zig

(Symmetric case too)

<u>Note:</u> x's depth decreases by two.

Zig-Zag

(Symmetric case too)

<u>Note:</u> x's depth decreases by two.

Zig

x has no grandparent (so, y is the root)

Note: w could be NIL

(Symmetric case too)

Note: x's depth decreases by one.

Complete Example

Splay(78)

zig-zag

Complete Example

Splay(78)

zig-zag

Complete Example

Splay(78)

zig-zag

Complete Example

Splay(78)

zig-zag

Complete Example

Splay(78)

zig-zag

Complete Example

Splay(78)

zig-zag

Complete Example

Splay(78)

zig

Case 1. x has no grandparent (*zig*)
If *x* is left child of root *y*, then rotate (*xy*)R.
Else if *x* is right child of root *y*, then rotate (*yx*)L.

Complete Example

Splay(78)

zig

Case 1 of 3

1. *x* has no grandparent (*zig*)
   - If *x* is left child of root *y*, then rotate (*xy*)R.
   - Else if *x* is right child of root *y*, then rotate (*yx*)L.

Case 2 of 3

2. x is LL or RR grandchild (*zig-zig*)

- If *x* is left child of *y*, and *y* is left child of *z*, then rotate at grandfather (*yz*)R and then rotate at father (*xy*)R.
- Else if *x* is right child of *y*, and *y* is right child of *z*,
  then rotate at grandfather (*yz*)L and then rotate at father (*xy*)L.

- If *x* has not become the root, then continue splaying at *x*.

Case 3 of 3

3. *x* is LR or RL grandchild (*zig-zag*)

- If *x* is right child of *y*, and *y* is left child of *z*, then rotate at father (*yx*)L and then rotate at grandfather (*xz*)R.
- Else if *x* is left child of *y*, and *y* is right child of *z*,
  then rotate at father (*yx*)R and then rotate at grandfather (*xz*)L.

- If *x* has not become the root, then continue splaying at *x*.

Red Black Trees

Colored Nodes Definition

- Binary search tree.
- Each node is colored red or black.
- Root and all external nodes are black.
- No root-to-external-node path has two consecutive red nodes.
- All root-to-external-node paths have the same number of black nodes

Red Black Trees

Colored Edges Definition

- Binary search tree.
- Child pointers are colored red or black.
- Pointer to an external node is black.
- No root to external node path has two consecutive red pointers.
- Every root to external node path has the same number of black pointers.

Example Red-Black Tree

Height of RB Tree

- The height of a red black tree that has n (internal) nodes is between lg(n+1) and 2*lg(n+1).

Properties

- Start with a red black tree whose height is h; collapse all red nodes into their parent black nodes to get a tree whose node-degrees are between 2 and 4, height is >= h/2, and all external nodes are at the same level.

Properties

Properties

- Let $h' >= h/2$ be the height of the collapsed tree.
- In worst-case, all internal nodes of collapsed tree have degree 2.
- Number of internal nodes in collapsed tree $>= 2^{h'}-1$.
- So, $n >= 2^{h'}-1$
- So, $h <= 2 \log_2 (n + 1)$

Properties

- At most 1 rotation and O(log n) color flips per insert/delete.
- Priority search trees.
  - Two keys per element.
  - Search tree on one key, priority queue on other.
  - Color flip doesn't disturb priority queue property.
  - Rotation disturbs priority queue property.
  - O(log n) fix time per rotation => $O(\log^2 n)$ overall time.

Properties

- O(1) amortized complexity to restructure following an insert/delete.
- C++ STL implementation
- java.util.TreeMap => red black tree

Insert

- New pair is placed in a new node, which is inserted into the red-black tree.
- New node color options.
  - Black node => one root-to-external-node path has an extra black node (black pointer).
    - Hard to remedy.
  - Red node => one root-to-external-node path may have two consecutive red nodes (pointers).
    - May be remedied by color flips and/or a rotation.

Classification Of 2 Red Nodes/Pointers

LLb

- XYz
  - X => relationship between gp and pp.
    - pp left child of gp => X = L.
  - Y => relationship between pp and p.
    - p right child of pp => Y = R.
  - z = b (black) if d = null or a black node.
  - z = r (red) if d is a red node.

XYr

- Color flip.

- Move p, pp, and gp up two levels.
- Continue rebalancing if necessary.

LLb

- Rotate.

- Done!
- Same as LL rotation of AVL tree.

LRb

- Rotate.

- Done!
- Same as LR rotation of AVL tree.
- RRb and RLb are symmetric.

Delete

- Delete as for unbalanced binary search tree.
- If red node deleted, no rebalancing needed.
- If black node deleted, a subtree becomes one black pointer (node) deficient.

Delete A Black Leaf

- Delete 8.

Delete A Black Leaf

- y is root of deficient subtree.
- py is parent of y.

Delete A Black Degree 1 Node

- Delete 45.
- y is root of deficient subtree.

Delete A Black Degree 2 Node

- Not possible, degree 2 nodes are never deleted.

Rebalancing Strategy

- If y is a red node, make it black.

Rebalancing Strategy

- Now, no subtree is deficient. Done!

Rebalancing Strategy

- y is a black root (there is no py).
- Entire tree is deficient. Done!

Rebalancing Strategy

- y is black but not the root (there is a py).

- Xcn
  - y is right child of py => X = R.
  - Pointer to v is black => c = b.
  - v has 1 red child => n = 1.

Rb0 (case 1)

- Color change.
- Now, py is root of deficient subtree.
- Continue!

Rb0 (case 2)

- Color change.
- Deficiency eliminated.
- Done!

Rb1 (case 1)

- LL rotation.
- Deficiency eliminated.
- Done!

Rb1 (case 2)

- LR rotation.
- Deficiency eliminated.
- Done!

Rb2

- LR rotation.
- Deficiency eliminated.
- Done!

Rr(n)

- n = # of red children of v's right child w.

Rr(2)

Rr(0)

- LL rotation.
- Done!

Rr(1) (case 1)

- LR rotation.
- Deficiency eliminated.
- Done!

Rr(1) (case 2)

- Rotation.
- Deficiency eliminated.
- Done!

Rr(2)

- Rotation.
- Deficiency eliminated.
- Done!

Red-Black Trees

Douglas Wilhelm Harder, M.Math. LEL
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

Copyright © 2006-11 by Douglas Wilhelm Harder. All rights reserved.

Outline

- The idea behind a red-black tree
- Defining balance
- Insertions and deletions
- The benefits of red-black trees over AVL trees

Red-Black Trees

A red black tree “colours” each node within a tree either red or black

- This can be represented by a single bit
- In AVL trees, balancing restricts the difference in heights to at most one
- For red-black trees, we have a different set of rules related to the colours of the nodes

Red-Black Trees

Define a *null path* within a binary tree as any path starting from the root where the last node is not a full node

– Consider the following binary tree:

Red-Black Trees

All null paths include:

(H, C, **B**)	(H, C, F, **D**)	(H, L, J, **I**)	(H, L, **P**)
(H, C, B, **A**)	(H, C, F, D, **E**)	(H, L, J, **K**)	(H, L, P, N, **M**)
	(H, C, F, **G**)		(H, L, P, N, **O**)

Red-Black Trees

The three rules which define a red-black tree are

1. The root must be black,
2. If a node is red, its children must be black, and
3. Each null path must have the same number of black nodes

Red-Black Trees

These are two examples of red-black trees:

Red-Black Trees

Theorem:

Every red node must be either

- A full node (with two black children), or
- A leaf node

Proof:

Suppose node S has one child:

- The one child L must be black
- The null path ending at S has k black nodes
- Any null path containing the node L will therefore have at least $k + 1$ black nodes

Red-Black Trees

In our two examples, you will note that all red nodes are either full or leaf nodes

Red-Black Trees

Another consequence is that if a node P has exactly one child:

- The one child must be red,
- The one child must be a leaf node, and
- That node P must be black

Red-Black Trees

Suppose that the child is black

- then the null path ending in P will have one fewer black nodes than the null path passing through C
- this contradicts the requirement that each null path has the same number of black nodes
- therefore the one child must be red

Red-Black Trees

Suppose the child is not a leaf node:

- Since it is red, its children:
  - cannot be red because that would contradict the requirement that all children of a red node are black, and
  - cannot be black, as that would cause any leaf node under the child to have more black nodes than the null path ending in P
- Contradiction, therefore the child must be a leaf node

Red-Black Trees

Since the one child is red and it must be a leaf node, the parent P must be black

- otherwise, we would contradict the requirement that the child of a red node must be black

Red-Black Trees

Again, in our examples, the only nodes with a single child are black and the corresponding children are red

Red-Black Trees

All red-black trees with 1, 2, 3, and 4 nodes:

Red-Black Trees

All red-black trees with 5 and 6 nodes:

Red-Black Trees

All red-black trees with seven nodes—most are shallow:

Red-Black Trees

Every perfect tree is a red-black tree if each node is coloured black

A complete tree is a red-black tree if:

- each node at the lowest depth is coloured red, and
- all other nodes are coloured black

What is the worst case?

Red-Black Trees

Any worst-case red-black tree must have an alternating red-black pattern down one side

The following are the worst-case red-black trees with 1 and 2 black nodes per null path (*i.e.*, heights 1 and 3)

Red-Black Trees

To create the worst-case for paths with 3 black nodes per path, start with a black and red node and add the previous worst-case for paths with 2 nodes

Red-Black Trees

This, however, is not a red-black tree because the two top nodes do not have paths with three black nodes

- To solve this, add the optimal red-black trees with two black nodes per path

Red-Black Trees

That is, add two perfect trees with height 1 to each of the missing sub-trees

Red-Black Trees

Thus, we have the worst-case for a red-black tree with three black nodes per path (or a red-black tree of height 5)

Red-Black Trees

Note that the left sub-tree of the root has height 4 while the right has height 1

- Thus suggests that AVL trees may be better...

Red-Black Trees

To create the worst-case scenario for red-black trees with 4 black nodes per null path (*i.e.*, height 7), again, start with two nodes and add the previous worst-case tree:

Red-Black Trees

To satisfy the definition of the red-black tree, add two perfect trees of height 2:

Red-Black Trees

Thus, with 30 nodes, the worst case red-black tree has height 7

– The worst-case AVL tree with 33 nodes has height 6...

Red-Black Trees

Let k represent the number of black nodes per null path and $h = 2k - 1$ represent the height of the worst-case tree

To determine $\mathrm{F}(k)$, the number of nodes in the worst-case tree, we note that $\mathrm{F}(1) = 2$ and:

$$\mathrm{F}(k) = \mathrm{F}(k-1) + 2 + 2(2^{k-1} - 1)$$

the number of nodes in the worst-case red-black tree with $k - 1$ nodes per null path

the two extra nodes

two perfect trees of height $k - 2$

Red-Black Trees

Use Maple:

```
> rsolve( {F(1) = 2, F(k) = 2 + F(k-1) + 2*(2^(k-1)-1)}, F(k) );
```

$$2\,2^{k} - 2$$

```
> solve( {h = 2*k - 1}, k );    # solve equation for k
```

$$\left\{k = \frac{h}{2} + \frac{1}{2}\right\}$$

```
> eval( 2*2^k - 2, % );         # evaluate at k = h/2 - 1/2
```

$$2\,2^{\left(\frac{h}{2} + \frac{1}{2}\right)} - 2$$

```
> solve( {n = %}, h );          # solve for h
```

$$\left\{h = \frac{2\ln\left(\frac{n}{2} + 1\right) - \ln(2)}{\ln(2)}\right\}$$

Red-Black Trees

A little manipulation shows that the worst-case height simplifies to

$$h_{\text{worst}} = 2\lg(n + 2) - 3$$

This grows quicker than the worst-case height for an AVL tree

$$h_{\text{worst}} = \log_{\phi}(n) - 1.3277$$

Red-Black Trees

Plotting the growth of the height of the worst-case red-black tree (red) versus the worst-case AVL tree (blue) demonstrates this:

Red-Black Trees

This table shows the number of nodes in a worst-case trees for the given heights

Thus, an AVL tree with 131070 nodes has a height of 23 while a red-black tree could have a height as large as 31

Height	AVL Tree	Red-Black Tree
1	2	2
3	7	6
5	20	14
7	54	30
9	143	62
11	376	126
13	986	254
15	2583	510
17	6764	1022
19	17710	2046
21	46367	4094
23	121492	8190
25	317810	16382
27	832039	32766
29	2178308	65534
31	5702886	131070
33	14930351	262142

Red-Black Trees

Comparing red-black trees with AVL trees, we note that:

- Red-black trees require one extra bit per node
- AVL trees require one byte per node (assuming the height will never exceed 255)
  - aside: we can reduce this to two bits, storing one of -1, 0, or 1 indicating that the node is left heavy, balanced, or right heavy, respectively

Red-Black Trees

AVL trees are not as deep in the worst case as are red-black trees

- therefore AVL trees will perform better when numerous searches are being performed,
- however, insertions and deletions will require:
  - more rotations with AVL trees, and
  - require recursions from and back to the root
- thus AVL trees will perform worse in situations where there are numerous insertions and deletions

Insertions

We will consider two types of insertions:

- bottom-up (insertion at the leaves), and
- top-down (insertion at the root)

The first will be instructional and we will use it to derive the second case

Bottom-Up Insertions

After an insertion is performed, we must satisfy all the rules of a red-black tree:

1. The root must be black,
2. If a node is red, its children must be black, and
3. Each path from a node to any of its descendants which are is not a full node (*i.e.*, two children) must have the same number of black nodes

The first and second rules are local: they affect a node and its neighbours

The third rule is global: adding a new black node anywhere will cause all of its ancestors to become unbalanced

Bottom-Up Insertions

Thus, when we add a new node, we will add a node so as to break the global rule:

- the new node must be red

We will then travel up the tree to the root fixing the requirement that the children of a red node must be black

Bottom-Up Insertions

If the parent of the inserted node is already black, we are done

- Otherwise, we must correct the problem

We will look at two cases:

- the initial insertion, and
- the recursive steps back to the root

Bottom-Up Insertions

For the initial insertion, there are two possible cases:

- the grandparent has one child (the parent), or
- the grandparent has two children (both red)

Inserting A, the first case can be fixed with a rotation:

Consequently, we are finished...

Bottom-Up Insertions

The second case can be fixed more easily, just swap the colours:

Unfortunately, we now may cause a problem between the parent and the grandparent....

Bottom-Up Insertions

Fortunately, dealing with problems caused within the tree are identical to the problems at the leaf nodes

Like before, there are two cases:

- the grandparent has one child (the parent), or
- the grandparent has two children (both red)

Bottom-Up Insertions

Suppose that A and D, respectively were swapped

In both these cases, we perform similar rotations as before, and we are finished

Bottom-Up Insertions

In the other case, where both children of the grandparent are red, we simply swap colours, and recurs back to the root

Bottom-Up Insertions

If, at the end, the root is red, it can be coloured black

Examples of Insertions

Given the following red-black tree, we will make a number of insertions

Examples of Insertions

Adding 46 creates a red-red pair which can be corrected with a single rotation

Examples of Insertions

Because the pivot is still black, we are finished

Examples of Insertions

Similarly, adding 5 requires a single rotation

Examples of Insertions

Which again, does not require any additional work

Examples of Insertions

Adding 10 allows us to simply swap the colour of the grand parent and the parent and the parent's sibling

Examples of Insertions

Because the parent of 5 is black, we are finished

Examples of Insertions

Adding 90 again requires us to swap the colours of the grandparent and its two children

Examples of Insertions

This causes a red-red parent-child pair, which now requires a rotation

Examples of Insertions

A rotation does not require any subsequent modifications, so we are finished

Examples of Insertions

Inserting 95 requires a single rotation

Examples of Insertions

And consequently, we are finished

Examples of Insertions

Adding 99 requires us to swap the colours of its grandparent and the grandparent's children

Examples of Insertions

This causes another red-red child-parent conflict between 85 and 90 which must be fixed, again by swapping colours

Examples of Insertions

This results in another red-red parent-child conflict, this time, requiring a rotation

Examples of Insertions

Thus, the rotation solves the problem

Top-Down Insertions and Deletions

With a bottom-up insertion, it is first necessary to search the tree for the appropriate location, and only then recurs back to the root correcting any problems

- This is similar to AVL trees

With red-black trees, it is possible to perform both insertions and deletions strictly by starting at the root, but not requiring the recurs back to the root

Top-Down Insertions

The important observation is:

- swapping may require recursive corrections going back all the way to the root
- rotations do not require recursive steps back to the root

Therefore, while moving down from the root, automatically swap the colours of any black node with two red children

- this may require at most one rotation at the parent of the now-red node

Examples of Top-Down Insertions

We will start with the same red-black tree as before, but make top-down insertions (no recursion):

Examples of Top-Down Insertions

Adding 46 does not find any (necessarily black) parent with two red children

Examples of Top-Down Insertions

However, it does require one rotation at the end

Examples of Top-Down Insertions

Similarly, adding 5 does not meet any parent with two red children:

Examples of Top-Down Insertions

A rotation solves the last problem

Examples of Top-Down Insertions

Adding 10 causes us to search down two edges before we meet node 5 with two red children

Examples of Top-Down Insertions

We swap the colours, and this does not cause a problem between 5 and 11

Examples of Top-Down Insertions

We continue and place 10 in the appropriate location

– No further rotations are required

Examples of Top-Down Insertions

To add the node 90, we traverse down the right tree until we reach 85 which has two red children

Examples of Top-Down Insertions

We swap the colours, however this creates a red-red pair between 85 and its parent

Examples of Top-Down Insertions

We require only one rotation to solve this problem, and we are guaranteed that this will not cause any problem for its parents

Examples of Top-Down Insertions

We continue to search down the right path and add 90 in the appropriate location—no further corrections are required

Examples of Top-Down Insertions

Next, adding 95, we traverse down the right-hand until we reach node 77 which has two red children

Examples of Top-Down Insertions

We swap the colours, which causes a red-red parent-child combination which must be fixed by a rotation

Examples of Top-Down Insertions

The rotation is around the root

– Note this rotation was not necessary with the bottom-up insertion of 95

Examples of Top-Down Insertions

We can now proceed to add 95 by following the right-hand branch, and the insertion causes a red-red parent-child combination

Examples of Top-Down Insertions

This is fixed with a single rotation

– We are guaranteed that this will not cause any further problems

Examples of Top-Down Insertions

If we compare the result of doing bottom-up insertions (left, seen previously) and top-down insertions (right), we note the resulting trees are different, but both are still valid red-black trees

Examples of Top-Down Insertions

If we add 99, the first thing we note is that the root has two red children, and therefore we swap the colours

Examples of Top-Down Insertions

At this point, each path to a non-full node still has the same number of black nodes, however, we violate the requirement that the root is black

Examples of Top-Down Insertions

We change the colour of the root to black

– This adds one more black node to each path

Examples of Top-Down Insertions

Moving to the right, we now reach node 90 which has two red children and therefore we swap the colours

Examples of Top-Down Insertions

We continue down the right to add 99

Examples of Top-Down Insertions

This does not violate any of the rules of the red-black tree and therefore we are finished

Examples of Top-Down Insertions

- Again, comparing the result of doing bottom-up insertions (left) and top-down insertions (right), we note the resulting trees are different, but both are still valid red-black trees

Top-Down Deletions

If we are deleting a red leaf node X, then we are finished

If we are deleting a node X with one child, we only need to replace the value of the deleted node with the
value of the leaf node

Top-Down Deletions

If we are deleting a full node, we use the same strategy used in standard binary search trees:

- replace the node with the minimum element in the right sub-tree
- then delete that element from the right sub-tree

Top-Down Deletions

That minimum element must be either:

- a red leaf node,
- a black node with a single red leaf node, or
- a black leaf node

The first two cases are solved, consequently, we need only deal with the possibility that the leaf node we are deleting is black

Top-Down Deletions

Similar to top-down insertions, we will adopt a strategy which ensures that the leaf node being deleted is not black

...to be completed...

Red-Black Trees

In this topic, we have covered red-black trees

- simple rules govern how nodes must be distributed based on giving each node a colour of either red or black
- insertions and deletions may be performed without recursing back to the root
- only one bit is required for the “colour”
- this makes them, under some circumstances, more suited than AVL trees

References

[1] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, Ch.14, p.261-80.

[2] Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley, §12.2, p.525-34.

AVL Trees

AVL Tree Definition

- **AVL trees are balanced.**
- An AVL Tree is a ***binary search tree*** such that for every internal node v of T, the *heights of the children of v can differ by at most 1*.

An example of an AVL tree where the heights are shown next to the nodes:

Height of an AVL Tree

- **Fact**: The ***height*** of an AVL tree storing n keys is O(log n).
- **Proof**: Let us bound **n(h):** the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and another of height n-2.
- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So

  n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),

  n(h) > 2^in(h-2i)
- Solving the base case we get: n(h) > $2^{h/2-1}$
- Taking logarithms: h < 2log n(h) +2
- Thus the height of an AVL tree is O(log n)

Insertion in an AVL Tree

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example:

before insertion

after insertion

Trinode Restructuring

- let (*a*,*b*,*c*) be an inorder listing of *x*, *y*, *z*
- perform the rotations needed to make *b* the topmost node of the three

Insertion Example, continued

Restructuring (as Single Rotations)

- Single Rotations:

Restructuring (as Double Rotations)

- double rotations:

Removal in an AVL Tree

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
- Example:

before deletion of 32 after deletion

Rebalancing after a Removal

- Let *z* be the first unbalanced node encountered while travelling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height.
- We perform restructure(x) to restore balance at z.
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached

Running Times for AVL Trees

- a single restructure is O(1)
  - using a linked-structure binary tree
- find is O(log n)
  - height of tree is O(log n), no restructures needed
- insert is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- remove is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

B-Trees

Motivation for B-Trees

- So far we have assumed that we can store an entire data structure in main memory
- What if we have so much data that it won't fit?
- We will have to use disk storage but when this happens our time complexity fails
- The problem is that Big-Oh analysis assumes that all operations take roughly equal time
- This is not the case when disk access is involved

Motivation (cont.)

- Assume that a disk spins at 3600 RPM
- In 1 minute it makes 3600 revolutions, hence one revolution occurs in 1/60 of a second, or 16.7ms
- On average what we want is half way round this disk – it will take 8ms
- This sounds good until you realize that we get 120 disk accesses a second – the same time as 25 million instructions
- In other words, one disk access takes about the same time as 200,000 instructions
- It is worth executing lots of instructions to avoid a disk access

Motivation (cont.)

- Assume that we use an Binary tree to store all the details of people in Canada (about 32 million records)
- We still end up with a **very** deep tree with lots of different disk accesses; $\log_2 20{,}000{,}000$ is about 25, so this takes about 0.21 seconds (if there is only one user of the program)
- We know we can't improve on the log *n* for a binary tree
- But, the solution is to use more branches and thus less height!
- As branching increases, depth decreases

Definition of a B-tree

- A B-tree of order m is an m-way tree (i.e., a tree where each node may have up to m children) in which:
  1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
  2. all leaves are on the same level
  3. all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
  4. the root is either a leaf node, or it has from two to m children
  5. a leaf node contains no more than $m - 1$ keys
- The number m should always be odd

An example B-Tree

A B-tree of order 5 containing 26 items

Note that all the leaves are at the same level

Constructing a B-tree

- Suppose we start with an empty B-tree and keys arrive in the following order:1 12 8 2 25 6 14 28 17 7 52 16 48 68 3 26 29 53 55 45
- We want to construct a B-tree of order 5
- The first four items go into the root:

- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree

Add 25 to the tree

Constructing a B-tree (contd.)

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

6, 14, 28 get added to the leaf nodes:

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

7, 52, 16, 48 get added to the leaf nodes

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

Adding 68 causes us to split the right most leaf, promoting 48 to the root

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

Adding 3 causes us to split the left most leaf

Constructing a B-tree (contd.)

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Add 26, 29, 53, 55 then go into the leaves

Constructing a B-tree (contd.)

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Add 45 increases the trees level

Exceeds Order. Promote middle and split.

Exceeds Order. Promote middle and split.

Inserting into a B-Tree

- Attempt to insert the new key into a leaf
- If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf's parent
- If this would result in the parent becoming too big, split the parent into two, promoting the middle key
- This strategy might have to be repeated all the way to the top
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

Exercise in Inserting a B-Tree

- Insert the following keys to a 5-way B-tree:
- 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

Answer to Exercise

Java Applet Source

Removal from a B-tree

- During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:
- 1 - If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
- 2 - If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case can we delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

Removal from a B-tree (2)

- If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf in question:
  - 3: if one of them has more than the min' number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf
  - 4: if neither of them has more than the min' number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required

Type #1: Simple leaf deletion

Assuming a 5-way B-Tree, as before...

Delete 2: Since there are enough keys in the node, just delete it

Note when printed: this slide is animated

Type #2: Simple non-leaf deletion

Note when printed: this slide is animated

Type #4: Too few keys in node and its siblings

Note when printed: this slide is animated

Type #4: Too few keys in node and its siblings

Note when printed: this slide is animated

Type #3: Enough siblings

Note when printed: this slide is animated

Type #3: Enough siblings

Note when printed: this slide is animated

Exercise in Removal from a B-Tree

- Given 5-way B-tree created by these data (last exercise):
- 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

- Add these further keys: 2, 6,12

- Delete these keys: 4, 5, 7, 3, 14

Answer to Exercise

Analysis of B-Trees

- The maximum number of items in a B-tree of order m and height h:

root	$m - 1$
level 1	$m(m - 1)$
level 2	$m^2(m - 1)$
. . .	
level h	$m^h(m - 1)$

- So, the total number of items is

$$(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) =$$

$$[(m^{h+1} - 1)/ (m - 1)] (m - 1) = \boldsymbol{m^{h+1} - 1}$$

- When $m = 5$ and $h = 2$ this gives $5^3 - 1 = 124$

Reasons for using B-Trees

- When searching tables held on disc, the cost of each disc transfer is high but doesn't depend much on the amount of data transferred, especially if consecutive items are transferred
  - If we use a B-tree of order 101, say, we can transfer each node in one disc read operation
  - A B-tree of order 101 and height 3 can hold $101^4 - 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory)
- If we take $m = 3$, we get a **2-3 tree**, in which non-leaf nodes have two or three children (i.e., one or two keys)
  - B-Trees are always balanced (since the leaves are all at the same level), so 2-3 trees make a good type of balanced tree

Faster Multiplication

Karatsuba's

method for Faster Multiplication

Examples

E.g. Polynomial in 2 variables with 3 terms

E.g. Polynomial multiplication

$$(2x+3)(5x+4)=?$$
$$=(2x)(5x)+(2x)(4)+(3)(5x)+(3)(4)$$
$$=10x^2+8x+15x+12$$
$$=10x^2+23x+12$$

$$(x+2)(x^2-4x+5)$$
$$=x^3-4x^2+5x+2x^2-8x+10$$
$$=x^3-2x^2-3x+10$$

Polynomials

Polynomial: $A(x) = \sum_{j=0}^{n-1} a_j x^j$

coefficient: complex, e.g.,

$3.25 + 6.999i$

real part

imaginary part

$\sqrt{-1}$

Degree-bound of A(x) is n.

Degree is k if a_k is the highest non-zero coefficient.

Polynomial Addition

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

$$B(x) = \sum_{j=0}^{n-1} b_j x^j$$

$$C(x) = A(x) + B(x) = \sum_{j=0}^{n-1} c_j x^j, \; c_j = a_j + b_j$$

$\Theta(n)$ time to compute.

Example of Polynomial Multiplication

$$
\begin{array}{rrrrrrr}
 & & & 6x^3 & +\ 7x^2 & -\ 10x & +\ 9 \\
 & & & -2x^3 & & +\ 4x & -\ 5 \\
\hline
 & & & -30x^3 & -\ 35x^2 & +\ 50x & -\ 45 \\
 & & 24x^4 & +\ 28x^3 & -\ 40x^2 & +\ 36x & \\
-12x^6 & -\ 14x^5 & +\ 20x^4 & -\ 18x^3 & & & \\
\hline
-12x^6 & -\ 14x^5 & +\ 44x^4 & -\ 20x^3 & -\ 75x^2 & +\ 86x & -\ 45
\end{array}
$$

Q. What is the complexity of this?

Usual multiplication

- Normal multiplication would take $(2n)^2$ cross multiplies and some data manipulation, $O(n^2)$.
- Karatsuba multiplication is $O(n^{1.58})$.

Polynomial Multiplication

$$C(x) = A(x) \cdot B(x)$$

$$= \sum_{j=0}^{2n-2} c_j x^j$$

$$\text{where } c_j = \sum_{k=0}^{j} a_k b_{j-k}$$

called **convolution**

Note: Degree bound of $C(x)$ is $2n - 1$.

Straightforward computation takes $\Theta(n^2)$ time.

Maple

```
C:\MAPLEV4\BIN.WIN\WMAPLE32.EXE

>  P:= x^3 + 5*x^2 + 11*x + 15;
>  Q:= 2*x + 3;
> degree(P,x);                 =  3
> coeff(P,x,1);                = 11
> coeffs(P,x);                 = {15, 11, 1, 5}
> subs(x=3,P);                 = 120
```

Polynomials Maple

```
> P:= x^3+5*x^2+11*x+15;
> Q := 2*x + 3;
> m := expand(P * Q);
    m = 2 x^4  + 13 x^3  + 37 x^2  + 63 x + 45.

> factor(m);
(2*x+3)*(x+3)*(x^2+2*x+5)

> roots(m);   … {-3,-3/2}      … Two Real roots.
> roots(m, I) … {-1±2i,-3,-3/2}… Complex roots
```

Pari

- Free Computer Algebra Software for Linux and windows.
- C:\tools\pari> gp.exe
- gp > m = 2*x^4 + 13*x^3 + 37*x^2 + 63*x + 45;
- gp > factor(m)
  - [x + 3, 1]
  - [2*x + 3, 1]
  - [x^2 + 2*x + 5, 1]
- gp > log(x+1)
  - x - 1/2*x^2 + 1/3*x^3 - 1/4*x^4 + 1/5*x^5 - 1/6*x^6...

Multiplication of polynomials

- We assume P and Q are two polynomials in x of equal size (terms) and of odd degree.
- If they are not, pad them with 0 coefficients.

Divide and Conquer multiplication

Divide P and Q of size 2n each into smaller polynomials of size n each:

$P = a^{2n-1}x^{2n-1} + \ldots + a_0$ is a poly of size 2n, (2n terms).

factor common x^n from first n terms:

$P = x^n(a_{2n-1}x^{n-1} + \ldots + a_{n+1}x + a_n) + (a_{n-1}x^{n-1} + \ldots + a_0)$

$P = A\,x^n + B$

Similarly for Q:

$Q = C\,x^n + D$

Usual multiplication of P*Q

Divide P and Q of size 2n each into smaller polynomials of size n each:

- $P = A\ x^n + B$
- $Q = C\ x^n + D$
- $P*Q = (A*C)(x^n)^2 + (A*D+B*C)(x^n) + (B*D)$
- There are 4 multiplies of size n polynomials, plus a size 2n polynomial addition:
- $O(4*(n^2))$. What we expected, no better.

Karatsuba's multiplication

- $P*Q = (A*C)(x^n)^2 + (A*D+B*C)(x^n) +(B*D)$
- The new idea: Instead, compute {R,S,T} in 3*
- $R = (A+B)*(C+D) = \underline{A*C}+A*D+B*C+\underline{B*D}$ .. mult1
- $S = A*C$ .. mult2
- $T = B*D$ .. mult3
- $U = R-S-T = A*D+B*C$
- $P*Q = S*(x^n)^2 + U*x^n + T.$
- $= A*C*(x^n)^2 + (R-S-T)*x^n + B*D.$
- $= A*C*(x^n)^2 + (A*D+B*C)*x^n + B*D.$

- Cost: 3 multiplies (instead of 4) of size n polynomials and a few additions of polynomials of size 2n.

Cost of karatsuba's recursive multiplications

- $M(2n) = 3*M(n)$ + cheaper adds $O(n)$....
  (problem halves, and 3 multiples per stage).
- $M(k) = 3^{lg(n)}*M(1)$
- $3^{lg(n)} = 2^{lg\ 3\ lg\ n} = n^{lg\ 3} = n^{1.58}$ (lg is log base 2).
- Generally if P and Q are about the same size and dense, and "large but not too large" this is good.
- Larger sizes: better methods.
- How large are the coefficients?

O(n^2) grows much faster than O(n^lg3)

Strassen's

Faster Matrix Multiplication

Basic Matrix 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.**($2^{\log_2 8} = 2^3$)**

Matrix multiplication

```
C[i][j] = sum(A[i][k] * B[k][j]) for k = 0 ... n
```

```
In our case:
C[1][1] =>
A[1][0]*B[0][1] + A[1][1]*B[1][1] + A[1][2]*B[2][1] + A[1][3]*B[3][1] + A[1][4]*B[4][1]
```

Retangular matrices

- A[m x n] x B [n x p] = C [m x p]
  - Rows of A == Columns of B
  - Rows of C == Rows A
  - Columns of C == Columns of B

Rectangular matrix multiplication is only possible when the second dimension of A equals the first dimensiono of B!

Matrix multiplication is not-commutative

Basic Matrix Multiplication

```
matrix_mult
  for (i = 1...N)
      for (j = 1..N)
          compute C_{i,j};
```

algorithm

Time analysis

$$C_{i,j} = \sum_{k=1}^{N} a_{i,k} b_{k,j}$$

$$\text{Thus } T(N) = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{N} c = cN^3 = O(N^3)$$

Strassens's Matrix Multiplication

- Strassen showed that 2x2 matrix multiplication can be accomplished in 7 multiplication and 18 additions or subtractions, **($7 = 2^{\lg 7} = 2^{2.807} < 2^3 = 2^{\lg 8} = 8$)**
- This savings can be applied recursively by Divide and Conquer Approach.

Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data $\boldsymbol{S}$ in two or more disjoint subsets $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, …
  - Recur: solve the sub-problems recursively
  - Conquer: combine the solutions for $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, …, into a solution for $\boldsymbol{S}$
- The base case for the recursion are sub-problems of constant size
- Analysis can be done using **recurrence equations**

Divide and Conquer Matrix Multiply

$$A \times B = R$$

$$\begin{bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 \\ B_2 & B_3 \end{bmatrix} = \begin{bmatrix} \mathbf{A_0 \times B_0 + A_1 \times B_2} & \mathbf{A_0 \times B_1 + A_1 \times B_3} \\ \mathbf{A_2 \times B_0 + A_3 \times B_2} & \mathbf{A_2 \times B_1 + A_3 \times B_3} \end{bmatrix}$$

- Divide matrices into sub-matrices.
- Use blocked matrix multiply equations
- Recursively multiply sub-matrices

Divide and Conquer Matrix Multiply

$$A \times B = R$$

$$\boxed{a_0} \times \boxed{b_0} = \boxed{a_0 \times b_0}$$

- Terminate recursion on small base case.

Strassens's Matrix Multiplication

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

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

$C_{11} = P_1 + P_4 - P_5 + P_7$
$C_{12} = P_3 + P_5$
$C_{21} = P_2 + P_4$
$C_{22} = P_1 + P_3 - P_2 + P_6$

Strassen's method

$$\overbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}^{B} = \overbrace{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}^{C}$$

```
P1 = A(F - H)
P2 = (A + B)H
P3 = (C + D)E
P4 = D(G - E)
P5 = (A + D)(E + H)
P6 = (B - D)(G + H)
P7 = (A - C)(E + F)
```

$$AB = \begin{bmatrix} P5 + P4 - P2 + P6 & P1 + P2 \\ P3 + P4 & P1 + P5 - P3 - P7 \end{bmatrix}$$

After defining the sum P1 ... P7, the product AB can be done in O(n^lg(7))!

Comparison

$$
\begin{aligned}
C_{11} &= P_1 + P_4 - P_5 + P_7 \\
&= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22} * (B_{21} - B_{11}) - (A_{11} + A_{12}) * B_{22} + \\
&\quad (A_{12} - A_{22}) * (B_{21} + B_{22}) \\
&= A_{11} B_{11} + A_{11} B_{22} + A_{22} B_{11} + A_{22} B_{22} + A_{22} B_{21} - A_{22} B_{11} - \\
&\quad A_{11} B_{22} - A_{12} B_{22} + A_{12} B_{21} + A_{12} B_{22} - A_{22} B_{21} - A_{22} B_{22} \\
&= A_{11} B_{11} + A_{12} B_{21}
\end{aligned}
$$

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);
 }
```

Divide matrices in sub-matrices and recursively multiply sub-matrices using matmul(..).

Strassen vs O(n^3)

- for small n (usually n < 45) the general algorithm is practically a better choice.

Faster Multiplication,

using Polynomials

Polynomial interpolation

The red dots denote the data points (x_i,y_i),
while the blue curve shows the interpolation polynomial.

Polynomial interpolation

Polynomial interpolation

- Suppose we have 4 data points:

  $(x1, y1)$, $(x2, y2)$, $(x3, y3)$, and $(x4, y4)$.
- There is exactly one polynomial of deg 3,

  $p(x) = ax^3 + bx^2 + cx + d$
- which passes through the four points.

Interpolate in Maple

```
# Evaluate P(x) and Q(x) at 3 points x=[-1,0,1,-1]
# And Interpolate at 3 points ([x1,x2,x3], [y1,y2,y3])
> p := x -> 1 + 2 * x;
                        p := x -> 1 + 2 x
> q := x -> 2 + x;
                         q := x -> 2 + x
> seq(p(x)*q(x), x=-1..1);
                            -1, 2, 9
> interp([-1,0,1],[-1,2,9], x);
                         2 x^2  + 5 x + 2
```

Plot the graphs

> plot({p(x), q(x), p(x)*q(x)},x=-2..2}

All the maple commands together

```
1.  p := x -> 1 + 2 * x;
2.  q := x -> 2 + x;
3.  seq(p(x)*q(x), x=-1..1);
4.  interp([-1,0,1],[-1,2,9], x);
5.  py:= {seq([x,p(x)], x=-1..1)};
6.  qy:= {seq([x,q(x)], x=-1..1)};
7.  pqy:= {seq([x,p(x)*q(x)], x=-1..1)};
8.  pl1:=plot({p(x), q(x), p(x)*q(x)},x=-2..2):
9.  pl2:=pointplot(py):
10. pl3:=pointplot(qy):
11. pl4:=pointplot(pqy);
12. with(plots):
13. display({pl1,pl2,pl3,pl4});
```

Vandermonde matrix

- Given the points $\{p(x_i) : i=0..n\}$
- we can compute the polynomial

  $p(x)= a_n x^n + .. + a_0$
- using the matrix inverse:

$$\begin{bmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & \dots & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_1^{n-2} & \dots & x_1 & 1 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ x_n^n & x_n^{n-1} & x_n^{n-2} & \dots & x_n & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Multiply polynomials h(x)= f(x)*g(x) by Interpolation at n points

A. for i=0…n;

evaluate f(i) and g(i);

save the value of h(i) = f(i) * g(i).

B. Interpolate { (h(i),i) : i=0..n } to the unique polynomial h(x) that passes through these points: (0,h(0)),(1,h(1))...(n,h(n)).

Horner's rule to evaluate P(X)

- Evaluating $P(x)$ at a point x_0.
- Brute force takes $O(n^2)$

  $P(x) = a_0 + x\, a_1 + x^2\, a_2 + x^3\, a_3 \ldots$
- ***Horner's rule*** does it in O(n) by factoring on x:

  $P(x) = a_0 + x\, (a_1 + x\, (a_2 + x\, (\ldots)))$
- Doing it at n points takes $O(n^2)$.

Horner's Polynomial Evaluation

- Given the coefficients $(a_0,a_1,a_2,\ldots,a_{n-1})$ of x^i in p(x).
- Evaluate p(x) at z in O(n) time using:

- **Function Horner**($A=(a_0,a_1,a_2,\ldots,a_{n-1})$,z):

  If n==1 then return a_0

  else return (a_0+ z * **Horner**($A'=(a_1,a_2,\ldots,a_{n-1})$,z))

Using Maple

```
> convert( 5 * x^5 + 3 * x^3 + 22 * x^2 + 55,
  horner);
```

```
55+(22+(3+5*x^2)*x)*x^2
```

Homework

Write C program to compute p(x) using horner rule, given polynomial coeffs as p[ ] and points x[ ].

e.g. p(x) = 5 * x^5 + 3 * x^3 + 22 * x^2 + 55

Horner of p(x) is 55+(22+(3+5*x^2)*x)*x^2

```
double p [ ] = { 55,0,22,3,0,5};
double x [ ] = { 1, 10, 20};
for k in 0..2
   print horner(p, x[k])
```

Homework solution: horner in C

```
double horner( double *coeffs, int n, double x ) {
    double y = 0.0;
    while ( n-- > 0)
        y = y * x + coeffs[n];
    return y;
}
#define LEN(A) (sizeof(A)/sizeof(A[0]))
int main(  ) {
    double k, p[] = { 1, 0, 3 };    // p(x):=1+0x+3x^2 = 1+x(0+3*x));
    for(k=-2;k<=2;k++)
      printf( "p(%g)= %g\n", k, horner( p, LEN(p), k) );
    return 0;
}
```

A peculiar way to multiply polynomials f(x) by g(x)

How much does this cost?

- Evaluations: 2*n evaluations of size(n) using Horner's rule is $O(n^2)$.
- Interpolation: using "Lagrange" or "Newton Divided Difference" is also $O(n^2)$.
- So $O(n^2)$, it is slower than Karatsuba.
- BUT....

Evaluate P(x) faster?

- Our choice of points 0,1,2,... n was arbitrary.
- What if we choose “better” points?
- Evaluating P(0) is almost free.
- Evaluating at symmetric points:

  P(c) and P(-c) can be done “faster” by sharing the calculations
- e.g. take coefficients of odd and even powers separately.

Evaluate p(-x) faster from p(x)

- $p1 = P_{ood} := a_{2n+1}x^{2n+1} + a_{2n-1}x^{n-1} + \ldots + a_1$
- $p2 = P_{even} := a_{2n}x^{2n} \qquad + \ldots + a_0$

- $P(x) = P_{odd}(x^2)\, x + P_{even}(x^2)$
- $P(x) = +p1*x+p2$
- $P(-x) = -p1*x+p2$.
- So P(c) can be computed by evaluating
  - $p1 = P_{even}(c^2)$
  - $p2 = P_{odd}(c^2)$
  - and returning c*p1+p2
- Finally, P(-c) is just –c*p1+p2, nearly free.

Are complex numbers better?

- Take a number r (real, complex, finite field,..)
- Compute the 2^m roots of 1 = {1, w, w^2, w^3, … w^{2^m}}, these roots have lot of symmetry.
- Evaluate P at these roots, exploiting symmetry to save on calculations.
- Evaluating $P(-w^k)$ and $P(w^k)$ can be done "faster"..
  - take even and odd coefficients separately,
  - by the same trick as before, with $c^2 = w^{2k}$

Enter: Numerical FFT

- Evaluate P at “complex roots of unity.”
- This reduces cost from n^2 to $O(n \log n)$;
- and same complexity for interpolation.

Fast multiplication of polynomials

Complex roots of Unity

(n 1 for evaluating the FFT)

Complex roots of unity

- *The n-th root of unity* is the complex number ω such that $\omega^n=1$.
- The *value* $w_n=exp(i\ 2\pi/n)$ is called ***the principal n-th root of unity*,**
- where: $i=\sqrt{-1}$
- $exp(i*u)=cos(u)+i*sin(u)$.

All the n roots of unity

The n distinct roots of 1 are

$= \{1, \omega, \omega^2, \ldots, \omega^{n-1}\}$

$= \{ \exp(k * i\, 2\, \square / n),\ k=1..n\}$

with $\omega = e^{2\pi \mathbf{i}/n}$

The n complex roots of unity are equally spaced around the circle of unit radius centered at the origin of the complex plane.

Properties of the roots of unity

Lemma-1:

For any integers $n \geq 0$ and $d > 0$,

$W_dn\hat{}dk = w_n\hat{}k$

Proof: $w_dn\hat{}dk$

$= exp(2pi\ i/dn)\hat{}(dk)$

$= exp(2pi\ i/n)\hat{}k = w_n\hat{}k$

Complex roots of unity

Corollary: For any even integer $n > 0$,

w_n^(n/2) = w_2=-1

Proof:

exp(n/2 * i 2 □ / n) = exp(i □) = -1

Complex roots of unity

Lemma-2: If $n > 0$ is even, then the squares of the n-th roots of unity are the ($n/2$) roots of unity:

$(w_n^k)^2 = w_\{n/2\}^k$

Proof:

$(w_n^\{k+n/2\})^2$

$= w_n^\{2k+n\}$

$= w_n^\{2k\} * w_n^n$

$= w_n^\{2k\} = (w_n^k)^2$

Complex roots of unity

Lemma-3:

For any integer $n \geq 1$ and $k >= 0$ not divisible by n,

S = sum_{j=0..n-1}{ (w_k^n)^j } = 0

Proof:

S = ((w_n^k)^n – 1) / (w_n^k – 1) Geometric series sum.

= ((w_n^n)^k – 1) / …

= (1 - 1)/ …

= 0

Properties of Primitive Roots of Unity

- **Inverse Property:** If ω is a primitive root of unity, then $\omega^{-1}=\omega^{n-1}$
  - Proof: $\omega\omega^{n-1}=\omega^{n}=1$
- **Cancellation Property:** if -n<k<n, $\sum_{j=0}^{n-1}\omega^{kj}=0$
  - Proof:

$$\sum_{j=0}^{n-1}\omega^{kj}=\frac{(\omega^{k})^{n}-1}{\omega^{k}-1}=\frac{(\omega^{n})^{k}-1}{\omega^{k}-1}=\frac{(1)^{k}-1}{\omega^{k}-1}=\frac{1-1}{\omega^{k}-1}=0$$

- **Reduction Property:** If w is a primitve (2n)-th root of unity, then ω^2 is a primitive n-th root of unity:
- Proof: If $1,\omega,\omega^2,\ldots,\omega^{2n-1}$ are all distinct, so is the subset $1,\omega^2,(\omega^2)^2,\ldots,(\omega^2)^{n-1}$

- **Reflective Property:** If n is even, then $\omega^{n/2}=-1$.
  - Proof: By the cancellation property, for k=n/2:

$$0=\sum_{j=0}^{n-1}\omega^{(n/2)j}=\omega^{0}+\omega^{n/2}+\omega^{0}+\omega^{n/2}+\cdots+\omega^{0}+\omega^{n/2}=(n/2)(1+\omega^{n/2})$$

  - Corollary: $\omega^{k+n/2}=-\omega^{k}$.

FFT
Fast Fourier Transform

The Fast Fourier Transform in O(n log n).

The DFT

We wish to evaluate a polynomial A(x) at each n-th roots of unity.

Assume *n* is a power of 2 (pad with 0 if required).

The DFT *Discrete Fourier Transform*

The vector y=(y_0..y_{n-1}) is called the **DFT** of the coefficient vector a=(a_0..a_{n-1}) of a polynomial A(x), and written as vector: $y = \mathrm{DFT}_n(a)$.

Each y_k is evaluated at the k-th root of 1:

y_k = A(w_n^k) = sum_{j=0..n-1}(a_j * w_n^{kj})

for k=0,1, ..., n-1

Divide the Polynomial into Odd/Even

Write: $A(x) = A0(x^2) + x\, A1(x^2)$.

Where

$A0(x) = a_0 + a_2\, x + a_4\, x^2 + ..$

$A1(x) = a_1 + a_3\, x + a_5\, x^2 + ..$

Break the polynomial in odd/even

– If n is even, we can divide a polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

into two polynomials

$$\begin{aligned} p^{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} \\ p^{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} \end{aligned}$$

$$p(x) = p^{\text{even}}(x^2) + x p^{\text{odd}}(x^2).$$

FFT divide and conquer

The problem of evaluating $A(x)$ at {w_n^0..w_n^{n-1}} reduces to

1. Evaluating the two $n/2$ degree polynomials A1(x), A0(x) at square of {w_n^0..w_n^{n-1}}
2. Combining the results, we get y_k=A(w_n^k).

Recursive FFT

```
RECURSIVE - FFT(a)
n ← length[a]
if n = 1
    then return a
ω_n ← e^{2πi/n}
ω ← 1
a^[0] ← (a_0, a_2, …, a_{n-2})
a^[1] ← (a_1, a_3, …, a_{n-1})
y^[0] ← RECURSIVE - FFT(a^[0])
y^[1] ← RECURSIVE - FFT(a^[1])
for k ← 0 to n/2 - 1
    do y_k ← y_k^[0] + ω y_k^[1]
       y_{k+n/2} ← y_k^[0] - ω y_k^[1]
       ω ← ω ω_n
return y                                  ▷ y is assumed to be a vector.
```

Running time of RECURSIVE-FFT

- Each invocation takes time $O(n)$, beside the recursive calls.
- $T(n)=2\,T(n/2) + O(n) = O(n \log n)$.

Interpolation

We can write the DFT as the matrix product $\boldsymbol{y} = \boldsymbol{Va}$ that is

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Interpolation, inverse matrix

Theorem: for j and k *in* {0,1, ..., *n*-1},

the (j,k) entry of the inverse of matrix is w_n^{-kj}/n.

And $V^* V^{-1} = I$

Proof: Next slide, from [Cormen 3e book, pg 913].

iDFT_n(y) = $(a_0..a_{n-1})$ can be computed using V^{-1}

a_j = 1/n * sum_{k=0..n-1}{y_k * w_n^{-kj}}

Inverse FFT

Theorem 30.7
For $j, k = 0, 1, \ldots, n-1$, the (j, k) entry of V_n^{-1} is ω_n^{-kj}/n.

Proof We show that $V_n^{-1} V_n = I_n$, the $n \times n$ identity matrix. Consider the (j, j') entry of $V_n^{-1} V_n$:

$$\begin{aligned}
[V_n^{-1} V_n]_{jj'} &= \sum_{k=0}^{n-1} (\omega_n^{-kj}/n)(\omega_n^{kj'}) \\
&= \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n \ .
\end{aligned}$$

This summation equals 1 if $j' = j$, and it is 0 otherwise by the summation lemma (Lemma 30.6). Note that we rely on $-(n-1) \le j' - j \le n-1$, so that $j' - j$ is not divisible by n, in order for the summation lemma to apply. ■

Given the inverse matrix V_n^{-1}, we have that $\mathrm{DFT}_n^{-1}(y)$ is given by

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj} \tag{30.11}$$

for $j = 0, 1, \ldots, n-1$. By comparing equations (30.8) and (30.11), we see that by modifying the FFT algorithm to switch the roles of a and y, replace ω_n by ω_n^{-1}, and divide each element of the result by n, we compute the inverse DFT (see Ex-

How to find inverse: F_n^{-1}?

Proposition. Let ω be a primitive *l*-th root of unity over a field L.Then

$$\sum_{k=0}^{l-1} \omega^k = \begin{Bmatrix} 0 & if\ l > 1 \\ 1 & otherwise \end{Bmatrix}$$

Proof. The *l* =1 case is immediate since ω=1.

Since ω is a primitive *l*-th root, each ω^k ,$k \neq 0$ is a distinct *l*-th root of unity.

$$Z^l - 1 = (Z - \omega_l^0)(Z - \omega_l)(Z - \omega_l^2)...(Z - \omega_l^{l-1}) =$$

$$= Z^l - (\sum_{k=0}^{l-1} \omega_l^k) Z^{l-1} + ... + (-1)^l \prod_{k=0}^{l-1} \omega_l^k$$

Comparing the coefficients of Z^{l-1} on the left and right hand sides of this equation proves the proposition.

Inverse matrix to F_n

Proposition. Let ω be an n-th root of unity.Then,

$$F_n(\omega)\cdot F_n(\omega^{-1}) = nE_n$$

Proof. The ij^{th} element of $F_n(\omega)F_n(\omega^{-1})$ is

$$\sum_{k=0}^{n-1}\omega^{ik}\omega^{-ik} = \sum_{k=0}^{n-1}\omega^{k(i-j)} = \begin{cases} 0, & \text{if } i \neq j \\ n, & \text{otherwise} \end{cases}$$

The *i=j* case is obvious.If *i≠j* then ω^{i-j} will be a primitive root of unity of order *l*, where *l*|*n*.Applying the previous proposition completes the proof.

So,

$$F_n^{-1}(\omega) = \frac{1}{n}F_n(\omega^{-1})$$

Evaluating	**y**= $F_n(\omega)$ **a**
Interpolation	**a**= $\frac{1}{n}F_n(\omega^{-1})$ **y**

FFT and iFFT are O(n log n)

By using the FFT and the iFFT,
we can transform a polynomial of degree n
back and forth between its coefficient
representation and a point-value representation
in time $O(n \log n)$.

The convolution

For any two vectors ***a*** and ***b*** of length *n* is a power of 2, we can do:

$$\mathbf{a} \otimes \mathbf{b} = DFT_{2n}^{-1}(DFT_{2n}(\mathbf{a}) \cdot DFT_{2n}(\mathbf{b}))$$

The Butterfly operation

The **for** loop involves computing the value $\omega_n^k y_k^{[1]}$ twice.We can change the loop(the butterfly operation):

$$
\begin{aligned}
&for\ \ k \leftarrow 0\ to\ n/2\text{-}1\\
&\quad do\ t \leftarrow \omega y_k^{[1]}\\
&\qquad y_k \leftarrow y_k^{[0]} + t\\
&\qquad y_{k+(n/2)} \leftarrow y_k^{[0]} \text{-} t\\
&\qquad \omega \leftarrow \omega\omega_n
\end{aligned}
$$

A butterfly operation. **(a)** The two input values enter from the left, the twiddle factor w_n^k is multiplied by y_k^[1] , and the sum and difference are output on the right. **(b)** A simplified drawing of a butterfly operation.

Convolution

- The DFT and the iDFT can be used to multiply two polynomials
- So we can get the coefficients of the product polynomial quickly if we can compute the DFT (and iDFT) quickly

Iterative FFT

We take the elements in pairs, compute the DFT of each pair, using one butterfly operation, and replace the pair with its DFT

We take these n/2 DFT's in pairs and compute the DFT of the four vector elements.

We take 2 (n/2)-element DFT's and combine them using n/2 butterfly operations into the final n-element DFT

Iterative-FFT Code with bit reversal.

0,4,2,6,1,5,3,7→000,100,010,110,001,101,011,111→000,001,010,011,100,101,110,111

```
BIT-REVERSE-COPY(a,A)
n←length [a]
for k←0 to n-1
      do A[rev(k)]←a_k
ITERATIVE-FFT
1.      BIT-REVERSE-COPY(a,A)
2.      n←length [a]
3.      for s←1 to log n
4.            do m←2^s
5.                ω_m ←e^(2πi/m)
6.       for j←0 to n-1 by m ←1
7.                for j←0 to m/2-1
8.                      do for  k←j to n-1 by m
9.                              do t← ωA[k+m/2]
10.                                u←A[k]
11.                                A[k]←u+t
12.                                A[k+m/2]←u-t
13.     ω← ω ω_m
14.     return A
```

A parallel FFT circuit

FFT Computation example in Maple

Example: Q=A*B

Usual multiplication the following polynomials in O(n^2):

A(x):=1+x+2x^2;

B(x):=1+2x+3x^2;

Q(x):=A(x)*B(x);

=1+3x+7x^2+7x^3+6x^4

Fast Polynomial Multiplication

Multiply the polynomials in O(n log(n))
using *DFT* of the coefficient vectors:

A=(1,1,2,0,0)

B=(2,1,3,0,0)

DFT(A)=

[4.000, (-0.309 - 2.126*i*), (0.809 + 1.314*i*),
(0.809 - 1.314*i*), (-0.309+ 2.126*i*)]

DFT(B)=

[6.000, (-0.809 - 3.665*i*), (0.309 + 1.677*i*),
(0.309 -1.677*i*), (-0.809 + 3.665*i*)]

A x B = iDFT(DFT(A)*DFT(B))

DFT(A) · DFT(B) =

[24.00, (-7.545 + 2.853*i*),

(-1.954 + 1.763*i*), (-1.954 - 1.763*i*),

(-7.545 - 2.853*i*)]

and

A x B = iDFT(DFT(A) . DFT(B)) = (1,3,7,7,6).

FFT speedup

If one complex multiplication takes 500ns:

N	T_{DFT}	T_{FFT}
2^{12}	8 sec.	0.013 sec.
2^{16}	0.6 hours	0.26 sec.
2^{20}	6 days	5 sec.

FFT in Maple

```
# Multiply polynomials A and B using FFT
A(x):=1+x+2*x^2;
B(x):=1+2*x+3*x^2;
Q(x):=expand(A(x)*B(x));
readlib(FFT);
ar := array([1,1,2,0,0,0,0,0]); ai := array([0,0,0,0,0,0,0,0]);
br := array([1,2,3,0,0,0,0,0]); bi := array([0,0,0,0,0,0,0,0]);
FFT(3,ar,ai); af := evalm(ar+I*ai);
FFT(3,br,bi); bf := evalm(br+I*bi);
abf := zip( (x,y)->x * y, af, bf );
abfr := evalm(map(x->Re(x), abf));
abfi := evalm(map(x->Im(x), abf));
iFFT(3,abfr, abfi);
evalm(abfr+I*abfi);
# coeffs of Q := [1,3,7,7,6]
```

FFT Programs in C and Python

FFT with numpy in python

```
#!python2.5
# What: fft in python using numpy.

from numpy.fft import fft
from numpy.fft import ifft
from numpy import array

from numpy import set_printoptions, get_printoptions
# print get_printoptions()
set_printoptions(precision = 4)
set_printoptions(suppress=True)

a = array((0, 1, 7, 2, -1, 3, 7, 8, 0, -23, -7, 31, 1, 31, -7, -31))

print "data =", a
y = fft(a)
print "fft(data) = ", y
z = ifft(y)
print "ifft(fft(data)) = ", z
```

Running fft-numpy.py

```
> fft-numpy.py
data = [  0   1   7   2  -1   3   7   8
          0 -23  -7  31   1  31  -7 -31]

fft(data) =  [
  22  +0j     -14.2 +10.7j -79.1  +0j      -4.8-101.8j
   0  -2.j      4.8 -58.2j  79.1  -0j      14.2 +46.3j
 -22  +0j      14.2 -46.3j  79.1  -0j       4.8 +58.2j
   0  +2.j     -4.8+101.8j -79.1  +0j     -14.2 -10.7j]

ifft(fft(data)) =  [
   0+0j   1-0j   7-0j   2-0j  -1+0j   3+0j   7-0j   8+0j
   0+0j -23-0j  -7+0j  31+0j   1+0j  31+0j  -7+0j -31+0j]
```

fft.c 1 of 2.

```
double PI = 3.14159265358979;
const double eps = 1.e-7;
typedef double complex cplx;

void fft_rec(cplx buf[], cplx out[], int n, int step, int inv) {
  if (step < n) {
    int i;
    fft_rec(out       ,     buf          , n, step * 2, inv);
    fft_rec(out + step, buf + step, n, step * 2, inv);
    for (i = 0; i < n; i += 2 * step) {
      int sign = inv? -1 : 1;
      cplx t = cexp(-I * PI * i * sign / n) * out[i + step];
      buf[ i / 2]       = out[i] + t;
      buf[(i + n)/2] = out[i] - t;
    }
  }
}
```

fft.c continued 2 of 2

```
void fft(cplx buf[], int n, int inv) {
  cplx out[n];
  int i;
  assert(is_power_of_two(n));
  for (i = 0; i < n; i++)
    out[i] = buf[i];
  fft_rec(buf, out, n, 1, inv);
  if (inv) for (i = 0; i < n; i++) buf[i] /= n;
}

int main() {
  cplx buf[] = {1, 2, 5, 1, 0, 0, 0, 99};
  fft(buf, 8, 0);
  fft(buf, 8, 1); // invfft
}
```

Running fft.c

```
> gcc -std=gnu99 -g -o fft.exe fft.c
> ./fft.exe
```

- Data: x[0]=(1, 0), x[1]=(2, 0), x[2]=(5, 0), x[3]=(1, 0),
- x[4]=(0, 0), x[5]=(0, 0), x[6]=(0, 0), x[7]=(99, 0),

- FFT: x[0]=(108, 0), x[1]=(71.71, 62.88),
- x[2]=(-4, 98), x[3]=(-69.71, 72.88),
- x[4]=(-96, 0), x[5]=(-69.71, -72.88),
- x[6]=(-4, -98), x[7]=(71.71, -62.88),

- iFFT: x[0]=(1, 0), x[1]=(2, 0), x[2]=(5, 0), x[3]=(1, 0),
- x[4]=(0, 0), x[5]=(0, 0), x[6]=(0, 0), x[7]=(99, 0),

Integers are also polynomials

- Cryptography involves multiplying large (100s of digits) integers. So it must be done efficiently.

- E.g. 2345 = 2x^3+3x^2+4x+5 (x=10)
- E.g. 0x45f = 4x^2 + 5x + f (x=16, hex).

Now we can use fast polynomial multiplication algorithm for integers also.

Example: Compute E to 100,000 decimals using fft-mul.c in 1 second

- > e2718.exe 100000 | head
- Total Allocated memory = 4608 K
- Starting series computation
- Starting final division
- Total time : 1.01 seconds
- Worst error in FFT (should be less than 0.25): 0.0003662109
- E = 2.7182818284 5904523536 0287471352 …

from http://xavier.gourdon.free.fr/Constants/constants.html

FFT Applications
Mathematica
Demonstrations

Mathematica demonstration of DCT/DST

2D FT

- 2 dimensional Fourier transforms simply involve a number of 1 dimensional fourier transforms.
- More precisely, a 2 dimensional transform is achieved by first transforming each row, replacing each row with its transform and then transforming each column, replacing each column with its transform.
- From http://paulbourke.net/miscellaneous/dft/

Images are also integers

- The JPEG compression algorithm (which is also used in MPEG compression) is based on the two-dimensional discrete cosine transform (DCT) applied to image blocks that are 8×8 pixels in size.
- DCT concentrates information about the pixels in the top-left corner of the 8×8 matrix so that the importance of information in the direction of the bottom-right corner decreases.
- It is then possible to degrade the low information value coefficients by dividing and multiplying them by the so-called quantization matrix.
- These operations are rounded to whole numbers, so the smallest values become zero and can be deleted to reduce the size of the JPEG.
- From http://demonstrations.wolfram.com/JPEGCompressionAlgorithm/

Mathematica Demonstration of JPG compression

Fourier descriptors for shape approximation

Fourier descriptors are a way of encoding the shape of a two-dimensional object by taking the Fourier transform of the boundary, where every (x,y) point on the boundary is mapped to a complex number x+i y.

The original shape can be recovered from the inverse Fourier transform.

However, if only a few terms of the inverse are used, the boundary becomes simplified, providing a way to smooth or filter the boundary.

From http://demonstrations.wolfram.com/FourierDescriptors/

Fourier descriptors details

The Fourier descriptors of a shape are calculated as follows.

- 1. Find the coordinates of the edge pixels of a shape and put them in a list in order, going clockwise around the shape.
  2. Define a complex-valued vector using the coordinates obtained. For example: .
  3. Take the discrete Fourier transform of the complex-valued vector.

Fourier descriptors inherit several properties from the Fourier transform.

- 4. Translation invariance: no matter where the shape is located in the image, the Fourier descriptors remain the same.
  5. Scaling: if the shape is scaled by a factor, the Fourier descriptors are scaled by that same factor.
  6. Rotation and starting point: rotating the shape or selecting a different starting point only affects the phase of the descriptors.

Because the discrete Fourier transform is invertible, all the information about the shape is contained in the Fourier descriptors. A common thing to do with Fourier descriptors is to set the descriptors corresponding to values above a certain frequency to zero and then reconstruct the shape. The effect of this is a low-pass filtering of the shape, smoothing the boundary. Since many shapes can be approximated with a small number of parameters, Fourier descriptors are commonly used to classify shapes.

- The slider lets you choose how many terms to use in the reconstruction. With more terms, the shape looks more like the original. With fewer terms, the shape becomes smoother and rounder.
- The basic method of Fourier descriptors is discussed in R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, Englewood Cliffs, NJ: Prentice Hall, 2007.

Dynamic Programming
Shortest Path

All pair shortest paths

- Given a weighted graph G(V,E,W)
- Calculate the shortest path between all pairs of vertices in the graph.
- D(i, j) for all i, j in V.

All Pairs Shortest Paths: Compute the matrix D(u, v) the shortest path distance from u to v for all pairs of vertices u and v in G.

Floyd-Warshall - all pair shortest paths

$D^{(0)}(i,j)$ = direct distance i to j

$D^{(k)}(i,j)$ = shortest path from i to j via k

Floyd-Warshall recurrence:

$$D^{(k)}(i,j) = \text{minimum} \{ D^{(k-1)}(i,j) , D^{(k-1)}(i,k) + D^{(k-1)}(k,j) \}$$

$$\underline{D^{(k-1)}(i,k)} + \underline{D^{(k-1)}(k,j)}$$

$$D^{(k-1)}(i,j)$$

Algorithm:

```
Initialize D^(0)(i,j)
For k = 1 .. n
    For i = 1 .. n
        For j = 1 .. n
            D(i,j) = min{ D(i,j) , D(i,k)+ D(k,j)}
```

Performance: $O(n^3)$ time and $O(n^2)$ space.

Example

$$D^0 = \begin{pmatrix} 0 & 3 & 0 & 0 \\ 0 & 0 & 12 & 5 \\ 4 & 0 & 0 & -1 \\ 2 & -4 & 0 & 0 \end{pmatrix} \quad D^1 = \begin{pmatrix} 0 & 3 & 0 & 0 \\ 0 & 0 & 12 & 5 \\ 4 & 7 & 0 & -1 \\ 2 & -4 & 0 & 0 \end{pmatrix} \quad D^2 = \begin{pmatrix} 0 & 3 & 15 & 8 \\ 0 & 0 & 12 & 5 \\ 4 & 7 & 0 & -1 \\ 2 & -4 & 8 & 0 \end{pmatrix}$$

$$D^3 = \begin{pmatrix} 0 & 3 & 15 & 8 \\ 16 & 0 & 12 & 5 \\ 4 & 7 & 0 & -1 \\ 2 & -4 & 8 & 0 \end{pmatrix} \quad D^4 = \begin{pmatrix} 0 & 3 & 15 & 8 \\ 7 & 0 & 12 & 5 \\ 1 & -5 & 0 & -1 \\ 2 & -4 & 8 & 0 \end{pmatrix}$$

Dynamic Programming

- ***Dynamic programming*** is typically applied to optimization problems. In such problem there can be ***many solutions***. Each solution has a value, and we wish to find *a* ***solution*** with the optimal value.

The development of a dynamic programming algorithm can be broken into a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom up fashion.
4. Construct an optimal solution from computed information.

Rod cutting

Given a rod of length n inches and a table of prices for each size (inch) of rod.

Determine the maximum revenue obtainable by cutting up the rod and selling the pieces.

Note that if the price for a rod of length n is large enough, an optimal soltion may require no cutting at all.

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

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

$$r_n = p_{i1} + p_{i2} + \ldots + p_{ik}$$

$r_1 = 1$ from solution 1 = 1 (no cuts) ,

$r_2 = 5$ from solution 2 = 2 (no cuts) ,

$r_3 = 8$ from solution 3 = 3 (no cuts) ,

$r_4 = 10$ from solution 4 = 2 + 2 ,

$r_5 = 13$ from solution 5 = 2 + 3 ,

$r_6 = 17$ from solution 6 = 6 (no cuts) ,

$r_7 = 18$ from solution 7=1+6 or 7=2+2+3 ,

$r_8 = 22$ from solution 8 = 2 + 6 ,

$r_9 = 25$ from solution 9 = 3 + 6 ,

$r_{10} = 30$ from solution 10 = 10 (no cuts) ,

- More generally , we can frame the values r_n for n ≥ 1 in terms of optimal revenues from shorter rods:

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

- ***Optimal substructure*** : optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

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

Recursive top-down implementation

```
CUT-ROD (p,n)
if n==0
  return 0
q = - ∞
for i = 1 to n
q= max(q, p[i] + CUT-ROD(p,n-i)
return q
```

- If you were to code up CUT-ROD in your favorite programming language and run it on your computer, you would find that once the input size becomes moderately large, your program would take a long time to run.

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

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

Using dynamic programming optimal rod cutting

- ***top-down with memoization***
- ***bottom-up method***

- MEMOIZ-CUT-ROD(p,n)

```
let r[0…n] be a new array
for i = 0 to n
    r[i] = - ∞
return MEMOIZED-CUT-ROD-AUX(p,n,r)
```

- MEMOIZED-CUT-ROD-AUX(p,n,r)

```
if  r[n] ≥ 0
      return r[n]
if  n==0
   q= 0
else  q=-∞
   for i= 1 to n
       q= max(q,p[i]+MEMOIZED-CUT-AUX(p,n-i,r)
 r[n]=q
return q
```

- BOTTOM-UP-CUT-ROD(p,n)

```
let r[0…n] be a new array
r[0] =0
for j =1 to n
 q=-∞
for i =1 to j
 q=max(q,p[i] + r[j-i])
r[j] = q
 return r[n]
```

- The running time of procedure BOTTOM-UP-CUT-ROD is $\Theta(n^2)$

Subproblem graphs

- The **subproblem graph** for the problem embodies exactly this information.

Reconstruction a solution

```
let r[0…n] and s[0…n] be new arrays
r[0] =0
for j =1 to n
q= -∞
for i =1 to j
if q < p[i] + r[j-i]
q = p[i] + r[j-i]
   s[j] = i
 r[j] = q
   return r and s
```

i	0	1	2	3	4	5	6	7	8	9	10
r[i]	0	1	5	8	10	13	17	18	22	25	30
s[i]	0	1	2	3	2	2	6	1	2	3	10

- PRINT-CUT-ROD-SOLUTION(p,n)

```
(r,s)= EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
while n>0
 print  s[n]
n = n – s[n]
```

15.1 Assembly-line scheduling

An automobile chassis enters each assembly line, has parts added to it at a number of stations, and a finished auto exits at the end of the line.

Each assembly line has n stations, numbered j = 1, 2,...,n. We denote the jth station on line j (where i is 1 or 2) by $S_{i,j}$. The jth station on line 1 ($S_{1,j}$) performs the same function as the jth station on line 2 ($S_{2,j}$).

The stations were built at different times and with different technologies, however, so that the time required at each station varies, even between stations at the same position on the two different lines. We denote the assembly time required at station $S_{i,j}$ by $a_{i,j}$.

As the coming figure shows, a chassis enters station 1 of one of the assembly lines, and it progresses from each station to the next. There is also an entry time e_i for the chassis to enter assembly line i and an exit time x_i for the completed auto to exit assembly line i.

a manufacturing problem
to find the fast way through a factory

Normally, once a chassis enters an assembly line, it passes through that line only. The time to go from one station to the next within the same assembly line is negligible.

Occasionally a special rush order comes in, and the customer wants the automobile to be manufactured as quickly as possible.

For the rush orders, the chassis still passes through the n stations in order, but the factory manager may switch the partially-completed auto from one assembly line to the other after any station.

The time to transfer a chassis away from assembly line i after having gone through station S_{ij} is $t_{i,j}$, where i = 1, 2 and j = 1, 2, ..., n-1 (since after the nth station, assembly is complete). The problem is to determine which stations to choose from line 1 and which to choose from line 2 in order to minimize the total time through the factory for one auto.

An instance of the assembly-line problem with costs

(a)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

(b)

Step1 The structure of the fastest way through the factory

- the fast way through station $S_{1,j}$ is either
  - the fastest way through Station $S_{1,j-1}$ and then directly through station $S_{1,j}$, or
  - the fastest way through station $S_{2,j-1}$, a transfer from line 2 to line 1, and then through station $S_{1,j}$.
- Using symmetric reasoning, the fastest way through station $S_{2,j}$ is either
  - the fastest way through station $S_{2,j-1}$ and then directly through Station $S_{2,j}$, or
  - the fastest way through station $S_{1,j-1}$, a transfer from line 1 to line 2, and then through Station $S_{2,j}$.

Step 2: A recursive solution

$$f^* = \min(f_1[n] + x_1,\ f_2[n] + x_2)$$

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if} \quad j = 1, \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if} \quad j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if} \quad j = 1, \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if} \quad j \geq 2 \end{cases}$$

step 3: computing the fastest times

- Let $r_i(j)$be the number of references made to $f_i[j]$ in a recursive algorithm.

  $r_1(n)=r_2(n)=1$

  $r_1(j) = r_2(j)=r_1(j+1)+r_2(j+1)$

- The total number of references to all $f_i[j]$ values is $\Theta(2^n)$.

- We can do much better if we compute the $f_i[j]$ values in different order from the recursive way. Observe that for $j \geq 2$, each value of $f_i[j]$ depends only on the values of $f_1[j\text{-}1]$ and $f_2[j\text{-}1]$.

FASTEST-WAY procedure

```
FASTEST-WAY(a, t, e, x, n)
f_1[1] ← e_1 + a_{1,1}
f_2[1] ← e_2 + a_{2,1}
for j ← 2 to n
      do if  f_1[j-1] + a_{1,j} ≤ f_2[j-1] + t_{2,j-1} + a_{1,j}
             then f_1[j] ← f_1[j-1] + a_{1,j}
                  l_1[j] ← 1
             else f_1[j] ← f_2[j-1] + t_{2,j-1} + a_{1,j}
                  l_1[j] ← 2
           if f_2[j-1] + a_{2,j} ≤ f_1[j-1] + t_{1,j-1} + a_{2,j}
```

```
            then  f_2[j] ← f_2[j – 1] + a_{2,j}
                  l2[j] ← 2
            else  f_2[j] ← f_1[j – 1] + t_{1,j-1} + a_2,j
                  l_2[j] ← 1
if  f_1[n] + x_1 ≤ f_2[n] + x_2
    then  f* = f_1[n] + x_1
          l* = 1
    else  f* = f_2[n] + x_2
          l* = 2
```

step 4: constructing the fastest way through the factory

```
PRINT-STATIONS(l, n)
i ← l*
print "line" i ",station" n
for j ← n downto 2
        do i ← l_i[j]
                print "line" i ",station" j – 1
```

output

line 1, station 6

line 2, station 5

line 2, station 4

line 1, station 3

line 2, station 2

line 1, station 1

Optimization

This, generally, refers to classes of problems that possess multiple solutions at one level, and where we have a real-valued function defined on the solutions.

Problem: find a solution that minimizes or maximizes the value of this function.

Note: there is no guarantee that such a solution will be unique and, moreover, there is no guarantee that you will find it (local maxima, anyone?) unless the search is over a small enough search space or the function is restricted enough.

Optimization

Question: are there classes of problems for which you can guarantee an optimizing solution can be found?

Answer: yes. BUT you also need to find such a solution in a "reasonable" amount of time.

We are going to look at two classes of problems, and the techniques that will succeed in constructing their solutions in a "reasonable" (i.e., low degree polynomial in the size of the initial data) amount of time.

Optimization

We begin with a rough comparison that contrasts a method you are familiar with (divide and conquer) and the method (still unspecified) of Dynamic Programming (developed by Richard Bellman in the late 1940's and early 1950's).

For some history and other ideas, see:
http://en.wikipedia.org/wiki/Dynamic_programming

Two Algorithmic Models:

	Divide & Conquer	Dynamic Programming
View problem as collection of subproblems	✓	✓
"Recursive" nature	✓	✓
Independent subproblems	✓	
Overlapping subproblems		✓
Number of subproblems	depends on partitioning factors	typically small
Preprocessing		
Characteristic running time	typically log function of n	depends on number and difficulty of subproblems
Primarily for optimization problems		✓
Optimal substructure: ***optimal solution to problem contains within it optimal solutions to subproblems***		✓

Dynamic Programming

Example: Rod Cutting

- You are given a rod of length $n \geq 0$ (n in inches)
- A rod of length i inches will be sold for p_i dollars
- Cutting is free (simplifying assumption)
- **Problem**: given a table of prices p_i determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

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

Example: Rod Cutting

We can see immediately (from the values in the table) that

$$n \leq p_n \leq 3n.$$

This is not very useful because:

- The range of potential revenue is very large
- Our finding quick upper and lower bounds depends on finding quickly the minimum and maximum p_i/i ratios (one pass through the table), but then we are back to the point above….

Example: Rod Cutting

Step 1: Characterizing an Optimal Solution

Question: in how many different ways can we cut a rod of length n?
For a rod of length 4:

$2^{4-1} = 2^3 = 8$

For a rod of length n: 2^{n-1}. **Exponential**: we cannot try all possibilities for n "large". The obvious exhaustive approach won't work.

Example: Rod Cutting

Step 1: Characterizing an Optimal Solution

Question: in how many different ways can we cut a rod of length n?

Proof Details: a rod of length n can have exactly n-1 possible cut positions – choose $0 \le k \le n$-1 actual cuts. We can choose the k cuts (without repetition) anywhere we want, so that for each such k the number of different choices is $\binom{n-1}{k}$

When we sum up over all possibilities (k = 0 to k = n-1):

$$\sum_{k=0}^{n-1}\binom{n-1}{k} = \sum_{k=0}^{n-1}\frac{(n-1)!}{k!(n-1-k)!} = (1+1)^{n-1} = 2^{n-1}.$$

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

Example: Rod Cutting

Characterizing an Optimal Solution

Let us find a way to solve the problem recursively (we might be able to modify the solution so that the maximum can be actually computed): assume we have cut a rod of length n into $0 \le k \le n$ pieces of length $i_1, \ldots, i_k$,

$n = i_1 + \ldots + i_k$,

with revenue

$r_n = p_{i1} + \ldots + p_{ik}$

Assume further that this solution is optimal.

How can we construct it?

Advice: when you don't know what to do next, start with a simple example and hope something will occur to you…

Example: Rod Cutting

Characterizing an Optimal Solution

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

We begin by constructing (by hand) the optimal solutions for i = 1, …, 10:

$r_1 = 1$ from sln. 1 = 1 (no cuts)
$r_2 = 5$ from sln. 2 = 2 (no cuts)
$r_3 = 8$ from sln. 3 = 3 (no cuts)
$r_4 = 10$ from sln. 4 = 2 + 2
$r_5 = 13$ from sln. 5 = 2 + 3
$r_6 = 17$ from sln. 6 = 6 (no cuts)
$r_7 = 18$ from sln. 7 = 1 + 6 or 7 = 2 + 2 + 3
$r_8 = 22$ from sln. 8 = 2 + 6
$r_9 = 25$ from sln. 9 = 3 + 6
$r_{10} = 30$ from sln. 10 = 10 (no cuts)

Example: Rod Cutting

Characterizing an Optimal Solution

Notice that in some cases $r_n = p_n$, while in other cases the optimal revenue r_n is obtained by cutting the rod into smaller pieces.
In ALL cases we have the recursion

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

exhibiting **optimal substructure** (meaning?)
A slightly different way of stating the same recursion, which avoids repeating some computations, is

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

And this latter relation can be implemented as a simple top-down recursive procedure:

```
CUT-ROD(p, n)
if n == 0
    return 0
q = -∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n - i))
return q
```

Example: Rod Cutting
Characterizing an Optimal Solution

Time Out: How to justify the step from:

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

to

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

Note: every optimal partitioning of a rod of length n has a first cut – a segment of, say, length i. The optimal revenue, r_n, must satisfy $r_n = p_i + r_{n-i}$, where r_{n-i} is the optimal revenue for a rod of length $n - i$. If the latter were not the case, there would be a better partitioning for a rod of length $n - i$, giving a revenue $r'_{n-i} > r_{n-i}$ and a total revenue $r'_n = p_n + r'_{n-i} > p_i + r_{n-i} = r_n$. Since we do not know which one of the leftmost cut positions provides the largest revenue, we just maximize over all the possible first cut positions.

Example: Rod Cutting
Characterizing an Optimal Solution

We can also notice that all the items we choose the maximum of are optimal in their own right: each substructure (max revenue for rods of lengths 1, …, n-1) is also optimal (again, **optimal substructure property**).

Nevertheless, we are still in trouble: computing the recursion leads to recomputing a number (= overlapping subproblems) of values – how many?

Example: Rod Cutting
Characterizing an Optimal Solution

Let's call Cut-Rod(p, 4), to see the effects on a simple case:

```
CUT-ROD(p, n)
if n == 0
    return 0
q = -∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n - i))
return q
```

The number of nodes for a tree corresponding to a rod of size n is:

$$T(0) = 1, \quad T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 2^n,\ n \ge 1.$$

Example: Rod Cutting
Beyond Naïve Time Complexity

We have a problem: "reasonable size" problems are not solvable in "reasonable time" (but, in this case, they are solvable in "reasonable space").

Specifically:
- Note that navigating the whole tree requires 2^n stack-frame activations.
- Note also that no more than n + 1 stack-frames are active at any one time and that no more than n + 1 different values need to be computed or used.

Can we exploit these observations?
A standard solution method involves saving the values associated with each $T(j)$, so that we compute each value only once (called "**memoizing**" = writing yourself a memo).

Rod Cutting with *Naïve Caching*

We introduce two procedures:

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = -∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = -∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
r[n] = q
return q
```

Rod Cutting *with better Caching*

We now remove some unnecessary complications:

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

Rod Cutting *Time Complexity*

Whether we solve the problem in a top-down or bottom-up manner the asymptotic time is $\Theta(n^2)$, the major difference being recursive calls as compared to loop iterations.

Dynamic programming

- Fibonacci Numbers
- Longest Common Subsequence (LCS)
- Shortest Path

Dynamic programming

- One disadvantage of using Divide-and-Conquer is that the process of recursively solving separate sub-instances can result in the **same computations being performed repeatedly** since *identical* sub-instances may arise.

- The idea behind ***dynamic programming*** is to avoid this pathology by obviating the requirement to calculate the same quantity twice.
- The method usually accomplishes this by maintaining a *table of sub-instance results*.

Bottom up

- Dynamic Programming is a **Bottom-Up Technique** in which the smallest sub-instances are *explicitly* solved first and the results of these used to construct solutions to progressively larger sub-instances.
- In contrast, Divide-and-Conquer is a **Top-Down Technique** which *logically* progresses from the initial instance down to the smallest sub-instance via intermediate sub-instances.
- **dynamic programming**. There are a couple of standard ways to progress:
  - memoization
  - converting from top-down to bottom-up

DYNAMIC PROGRAMMING

- ***Dynamic programming*** solves a problem by partitioning the problem into sub-problems.
  - If the subproblems are ***independent***: use divide-and-conquer method.
  - If the subproblems are ***dependent***: use dynamic programming.
- A dynamic programming algorithm solves every subproblem once and saves its answer to sub-problems in a table for reuse it.

divide-and-conquer

Dynamic Programming

Computing the Fibonacci sequence the slow way

```
int  fibo_very_slow(n)
   if n < 2 then return 1
   else return f(n-1) + f(n-2);
```

$\Theta(2^n)$ time and $\Theta(n)$ space.

fibo(6) call tree with duplicate calls

Memoization

Remember earlier solutions (memoization)

```
int fibo_dynamic(int n)
  static saved[N]
  if n < 2  then return 1
  if saved[n] then return saved[n];
  saved[N] = f(n-1) + f(n-2)
  return saved[N]
```

Θ(n) time and Θ(n) space

Fibonacci numbers

```
int fibo_fast(int n)
If n < 2 then return 1
f1 = f2 = 1;
 for k =1 to n
     f1 = f1 + f2
     f2 = f1
return f1
```

Θ(n) time and
Θ(1) space

LCS: Longest Common Subsequence

- Example of Dynamic programming
- Matching two strings with missing characters

DNA are strings over {C,G,A,T}
RNA are strings over {C,G,A,U}

DNA Matching

- The main objective of DNA analysis is to get a visual representation of DNA left at the scene of a crime.
- A DNA "picture" features columns of dark-colored parallel bands and is equivalent to a fingerprint lifted from a smooth surface.
- To identify the owner of a DNA sample, the DNA "fingerprint," or profile, must be matched, either to DNA from a suspect or to a DNA profile stored in a database.

LCS example in DNA matching

The longest common subsequence (LCS) problem

string : A = b a c a d

subsequence of A is formed by deleting 0 or more symbols from A (not necessarily consecutive).

e.g. ad, ac, bac, acad, bacad, bcd.

Common subsequences of

A = b a c a d

B = a c c b a d c b

Are: ad, ac, bac, acad.

The **longest common subsequence (LCS)** of A and B:

a c a d.

LONGEST COMMON SUBSEQUENCE

- ***Substring***
  - *CBD* is a substring of *ABCBDAB*

Subsequence

- *BCDB* is a subsequence of *ABCBDAB*

- ***Common subsequence***
  - *BCA* is a common subsequence of

    X=*ABCBDAB* and *Y*=*BDCABA*
  - *BCBA* is the longest common subsequence of *X* and *Y*

    X = *A B C B D A B*

    Y = *B D C A B A*

LCS problem

Given two sequences $X=<x_1, x_2, \ldots, x_m>$ and $Y=<y_1, y_2, .., y_n>$ to find an LCS of X and Y.

- **Brute force approach**
  - Enumerate all subsequences of X and Y
  - Find the common to both.
  - Find the longest one.
- Is Infeasible: space $\Theta(2^m+2^n)$, time $\Theta(2^{m+n})$,

  (There are 2^m subsequences of X is)

LCS data structure is a table c

- $X = X_1 X_2 X_3 \ldots\ldots\ldots\ldots\ldots X_n$
- $Y = Y_1 Y_2 Y_3 \ldots. Y_m$
- $c[i, j]$ is table of length(LCS(*X[1..i],Y[1..j])*)

  Where:

  c[0,i] = c[i,0] = 0 // first row and column are zero.

  c[i,j] = 1+c[i-1,j-1] if $X_i == Y_j$

  = max(c[i,j-1],c[i-1,j]) otherwise

- **Time complexity: O(mn)**
- **Space complexity: O(mn)**

LCS sub-problem table computation

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

// first row and column are zero.
c[0,i] = c[i,0] = 0
c[i,j] = 1+c[i-1,j-1] if Xi == Yj
= max(c[i,j-1],c[i-1,j]) otherwise

```
PRINT-LCS(i, j)
if i = 0 or j = 0
    then return
if c[i, j] == 1+c[i-1,j-1] then
    PRINT-LCS( i - 1, j - 1)
    print x[i]
else if c[i, j] == c[i-1,j] then
        PRINT-LCS(b, X, i - 1, j)
else PRINT-LCS(b, X, i, j - 1)
```

LCS example

		-1	0	1	2	3	4	5	6	7	8
			C	A	G	A	T	A	G	A	G
-1		0	0	0	0	0	0	0	0	0	0
0	A	0	0	1	1	1	1	1	1	1	1
1	G	0	0	1	2	2	2	2	2	2	2
2	C	0	1	1	2	2	2	2	2	2	2
3	G	0	1	1	2	2	2	2	3	3	3
4	A	0	1	2	2	3	3	3	3	4	4

This corresponds to the four following alignments:

$$\begin{pmatrix} - & A & G & C & - & - & - & G & A & - \\ C & A & G & - & A & T & A & G & A & G \end{pmatrix}$$

$$\begin{pmatrix} - & A & G & - & C & - & - & G & A & - \\ C & A & G & A & - & T & A & G & A & G \end{pmatrix}$$

$$\begin{pmatrix} - & A & G & - & - & C & - & G & A & - \\ C & A & G & A & T & - & A & G & A & G \end{pmatrix}$$

$$\begin{pmatrix} - & A & G & - & - & - & C & G & A & - \\ C & A & G & A & T & A & - & G & A & G \end{pmatrix}$$

Exercises

Build the LCS table for

1. X=ABC, Y=ABC
2. X=AAA, Y=BBB
3. X=ABA, Y=BAB

Solutions

Build the LCS table for

1. X=ABC, Y=ABC
2. X=AAA, Y=BBB
3. X=ABA, Y=BAB

Homework

Find LCS of

1. X=TAGAGA,
   Y=GAGATA

2. X=YOUR_FIRST_NAME
   Y=YOUR_LAST_NAME

Homework Solution

Find LCS of X=TAGAGA, Y=GAGATA

	j	0	1	2	3	4	5	6
i		y	G	A	G	A	T	A
0	x	0	0	0	0	0	0	0
1	T	0	↑ 0	↑ 0	↑ 0	↑ 0	↖ 1	← 1
2	A	0	↑ 0	↖ 1	← 1	↖ 1	↑ 1	↖ 2
3	G	0	↖ 1	↑ 1	↖ 2	← 2	← 2	↑ 2
4	A	0	↑ 1	↖ 2	↑ 2	↖ 3	← 3	↖ 3
5	G	0	↖ 1	↑ 2	↖ 3	↑ 3	↑ 3	↑ 3
6	A	0	↑ 1	↖ 2	↑ 3	↖ 4	← 4	↖ 4

Max subarray sum

(example of dynamic programming)

Max subarray sum

Example:

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Figure 4.1 Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

Max subarray sum

Another Example: buying low and selling high, even with perfect knowledge, is not trivial:

Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

Figure 4.2 An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of \$3 per share would be earned by buying after day 2 and selling after day 3. The price of \$7 after day 2 is not the lowest price overall, and the price of \$10 after day 3 is not the highest price overall.

Max subarray sum

First Solution: compute the value change of each subarray corresponding to each pair of dates, and find the maximum.

1. How many pairs of dates: $\binom{n}{2}$
2. This belongs to the class $\Theta(n^2)$
3. The rest of the costs, although possibly constant, don't improve the situation: $\Omega(n^2)$.

Not a pleasant prospect if we are rummaging through long time-series (Who told you it was easy to get rich???), even if you are allowed to post-date your stock options…

Max subarray sum

We are going to find an algorithm with an $o(n^2)$ running time (i.e. **strictly** asymptotically faster than n^2), which should allow us to look at longer time-series.

Transformation: Instead of the daily price, let us consider the daily change: A[i] is the difference between the closing value on day i and that on day i-1.

The problem becomes that of finding a contiguous subarray the sum of whose values is maximum.

On a first look this seems even worse: roughly the same number of intervals (one fewer, to be precise), and the requirement to add the values in the subarray rather than just computing a difference: $\Omega(n^3)$?

Max subarray sum

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8..11]$, with sum 43, has the greatest sum of any contiguous subarray of array A.

It is actually possible to perform the computation in $\Theta(n^2)$ time by

1. Computing all the daily changes;
2. Computing the changes over 2 days (one addition each)
3. Computing the changes over 3 days (one further addition to extend the length-2 arrays… etc… check it out. You'll need a two-dimensional array to store the intermediate computations.

Still bad though. Can we do better??

Max subarray sum

How do we divide?

We observe that a maximum contiguous subarray A[i…j] must be located as follows:

1. It lies entirely in the left half of the original array: [low…mid];
2. It lies entirely in the right half of the original array: [mid+1…high];
3. It straddles the midpoint of the original array: i ≤ mid < j.

Figure 4.4 **(a)** Possible locations of subarrays of $A[low\,..\,high]$: entirely in $A[low\,..\,mid]$, entirely in $A[mid+1\,..\,high]$, or crossing the midpoint mid. **(b)** Any subarray of $A[low\,..\,high]$ crossing the midpoint comprises two subarrays $A[i\,..\,mid]$ and $A[mid+1\,..\,j]$, where $low \le i \le mid$ and $mid < j \le high$.

Max subarray sum

The “left” and “right” subproblems are smaller versions of the original problem, so they are part of the standard Divide & Conquer recursion. The “middle” subproblem is not, so we will need to count its cost as part of the “combine” (or “divide”) cost.

The crucial observation (and it may not be entirely obvious) is that **we can find the maximum crossing subarray in time linear in the length of the A[low…high] subarray**.

How? A[i,…,j] must be made up of A[i…mid] and A[m+1…j] – so we find the largest A[i…mid] and the largest A[m+1…j] and combine them.

Max subarray sum

The Algorithm:

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

Max subarray sum

The total numbers of iterations for both loops is exactly high-low+1.

The left-recursion will return the indices and value for the largest contiguous subarray in the left half of A[low…high], the right recursion will return the indices and value for the largest contiguous subarray in the left half of A[low…high], and FIND-MAX-CROSSING-SUBARRAY will return the indices and value for the largest contiguous subarray that straddles the midpoint of A[low…high].

It is now easy to choose the contiguous subarray with largest value and return its endpoints and value to the caller.

Max subarray sum

The recursive algorithm:

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Max subarray sum

The analysis:
1. The base case $n = 1$. This is easy: $T(1) = \Theta(1)$, as one might expect.
2. The recursion case $n > 1$. We make the simplifying assumption that n is a power of 2, so we can always "split in half". $T(n)$ = cost of splitting + $2T(n/2)$ + cost of finding largest midpoint-straddlingsubarray + cost of comparing the three subarray values and returning the correct triple.
The cost of splitting is constant = $\Theta(1)$; the cost of finding the largest straddling subarray is $\Theta(n)$; the cost of the final comparison and return is constant $\Theta(1)$.

$T(n) = \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1)$.

Max subarray sum

We finally have:

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

The recurrence has the same form as that for MERGESORT, and thus we should expect it to have the same solution T(n) = Θ(n lg n).

This algorithm is clearly substantially faster than any of the brute-force methods. It required some cleverness, and the programming is a little more complicated – but the payoff is large.

MCM: Optimal Matrix Chain Multiplication

example of Dynamic Programming

Dynamic Programming vs. Recursion and Divide & Conquer

- In a recursive program, a problem of size n is solved by first solving a sub-problem of size $n-1$.
- In a divide & conquer program, you solve a problem of size n by first solving a sub-problem of size k and another of size $k-1$, where $1 < k < n$.
- In dynamic programming, you solve a problem of size n by first solving *all* sub-problems of *all* sizes k, where $k < n$.

Matrix Multiplication

Matrix A - *p* rows & *q* columns (dimension: p * q)

Matrix B - *q* rows & *r* columns (diamension: q * r)

Note: We can only multiply two matrices, A & B,

if number of columns in A = number of rows in B

Cost of AB = # of scalar multiplications

$= p\,q\,r$

Matrix-Chain Multiplication

Given a sequence of *n* matrices: $A_1 A_2 \ldots A_n$

Compute $= A_1 * A_2 * \ldots * A_n$

The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product.

Matrix Chain Multiplication

- Given some matrices to multiply, determine the ***best*** order to multiply them so you minimize the number of single element multiplications.
  - i.e. Determine the way the matrices are parenthesized.

- First off, it should be noted that matrix multiplication is associative, but not commutative. But since it is associative, we always have:

- ((AB)(CD)) = (A(B(CD))), or any other grouping as long as the matrices are in the same consecutive order.

- BUT NOT: ((AB)(CD)) = ((BA)(DC))

Matrix-chain multiplication

- A product of matrices is **fully parenthesized** if it is either a single matrix, or a product of two fully parenthesized matrix product, surrounded by parentheses.

Example

Consider the chain: $A_1A_2A_3A_4$ of 4 matrices

There are 5 possible ways to compute the product $A_1A_2A_3A_4$

1. $(A_1(A_2(A_3A_4)))$
2. $(A_1((A_2A_3)A_4))$
3. $((A_1A_2)(A_3A_4))$
4. $((A_1(A_2A_3))A_4)$
5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions

- Can you write the algorithm to multiply two matrices?

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p\times q}$ and $B_{q\times r}$ (with dimensions $p\times q$ and $q\times r$)
Result: Matrix $C_{p\times r}$ resulting from the product $A\cdot B$

MATRIX-MULTIPLY($A_{p\times q}$, $B_{q\times r}$)

```
1.  for i ← 1 to p
2.          for j ← 1 to r
3.              C[i, j] ← 0
4.              for k ← 1 to q
5.                  C[i, j] ← C[i, j] + A[i, k] · B[k, j]
6.  return C
```

Scalar multiplication in line 5 dominates time to compute CNumber of scalar multiplications = *pqr*

MATRIX MULTIPLY

```
MATRIX-MULTIPLY(A, B)
if A.columns ≠ B.rows
    error "incompatible dimensions"
else let C be a new A.rows × B.columns matrix
    for i = 1 to A.rows
        for j = 1 to B.columns
            c_ij = 0
            for k = 1 to A.columns
                c_ij = c_ij + a_ik · b_kj
    return C
```

Complexity:

- Let ***A*** be a $p \times q$ matrix, and ***B*** be a $q \times r$ matrix.
- Then the complexity is $p \times q \times r$

Matrix-Chain Multiplication Problem

Computing the minimum cost order of multiplying a list of n matrices:

$$A_1 A_2 \ldots A_n$$

For i = 1,2, ..,n; matrix A_i has dimension $p_{i-1} * p_i$

The number of alternative parenthesizations is exponential in *n (catalan number).*

Matrix-chain Multiplication

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 =2{,}500$ scalar multiplications

    Total: 7,500
  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 =50{,}000$ scalar multiplications

    Total: 75,000

Matrix Chain Multiplication
Optimal Parenthesization

- Example: A[30][35], B[35][15], C[15][5]
  minimum of A*B*C

  A*(B*C) = 30*35*5 + 35*15*5 = 7,585

  (A*B)*C = 30*35*15 + 30*15*5 = 18,000
- How to optimize:
  - Brute force – look at every possible way to parenthesize : $\Omega(4^n/n^{3/2})$
  - Dynamic programming – time complexity of $\Omega(n^3)$ and space complexity of $\Theta(n^2)$.

Matrix Chain Multiplication

- Let's get back to our example: We will show that the way we group matrices when multiplying A, B, C ***matters***:
  - Let A be a 2x10 matrix
  - Let B be a 10x50 matrix
  - Let C be a 50x20 matrix

- Consider computing **A(BC):**
  - # multiplications for (BC) = 10x50x20 = 10000, creating a 10x20 answer matrix
  - # multiplications for A(BC) = 2x10x20 = 400
  - Total multiplications = 10000 + 400 = 10400.

- Consider computing **(AB)C**:
  - # multiplications for (AB) = 2x10x50 = 1000, creating a 2x50 answer matrix
  - # multiplications for (AB)C = 2x50x20 = 2000,
  - Total multiplications = 1000 + 2000 = 3000

Matrix Chain Multiplication

- Our final multiplication will ALWAYS be of the form
  - $(A_0 \bullet A_1 \bullet \ldots A_k) \bullet (A_{k+1} \bullet A_{k+2} \bullet \ldots A_{n-1})$
- In essence, there is exactly one value of k for which we should "split" our work into two separate cases so that we get an optimal result.
  - Here is a list of the cases to choose from:
  - $(A_0) \bullet (A_1 \bullet A_{k+2} \bullet \ldots A_{n-1})$
  - $(A_0 \bullet A_1) \bullet (A_2 \bullet A_{k+2} \bullet \ldots A_{n-1})$
  - $(A_0 \bullet A_1 \bullet A_2) \bullet (A_3 \bullet A_{k+2} \bullet \ldots A_{n-1})$
  - ...
  - $(A_0 \bullet A_1 \bullet \ldots A_{n-3}) \bullet (A_{n-2} \bullet A_{n-1})$
  - $(A_0 \bullet A_1 \bullet \ldots A_{n-2}) \bullet (A_{n-1})$
- Basically, count the number of multiplications in each of these choices and **pick the minimum**.
  - One other point to notice is that you have to account for the minimum number of multiplications in each of the two products.

Catalan number: Number of ways to parenthesize

$$P(n) = \begin{cases} 1 & if \;\; n = 1 \\ \sum_{k=1}^{n-1} P(k)p(n-k) & if \;\; n \geq 2 \end{cases}$$

$$P(n) = C(n-1)$$

$$= \frac{1}{n+1}\binom{2n}{n} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

Step 1: The structure of an optimal parenthesization

Optimal

$$((A_1 A_2 \ldots A_k)(A_{k+1} A_{k+2} \ldots A_n))$$

Combine

Step 2: A recursive solution

- Define *m*[*i, j*]= minimum number of scalar multiplications needed to compute the matrix $A_{i..j} = A_i A_{i+1} .. A_j$
- Goal find *m*[*1, n*]
- $$m[i, j] = \begin{cases} 0 & i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1, j] + p_{i-1} p_k p_j\} & i \ne j \end{cases}$$

How to split the chain?

- Basically, we're checking different places to "split" our matrices by checking different values of k and seeing if they improve our current minimum value.

Step 3. Compute Optimal Cost

Input: Array $p[0\ldots n]$ containing matrix dimensions and n

Result: Minimum-cost table *m[1..n,1..n]* and split table *s[1..n,1..n]*

```
MATRIX-CHAIN-ORDER(p[ ], n)
    for i ← 1 to n
        m[i, i] ← 0
    for l ← 2 to n
        for i ← 1 to n-l+1
                j ← i+l-1
                m[i, j] ← ∞
                for k ← i to j-1
                        q ← m[i, k] + m[k+1, j] + p[i-1] p[k] p[j]
                        if q < m[i, j]
                                m[i, j] ← q
                                s[i, j] ← k
    return m and s
```

Takes $O(n^3)$ time

Requires $O(n^2)$ space

Constructing Optimal Solution

- Our algorithm computes the minimum-cost table *m* and the split table *s*
- The optimal solution can be constructed from the split table *s*
- Each entry $s[i, j]=k$ shows where to split (cut) the product chain $A_i A_{i+1} \ldots A_j$ for the minimum cost.

Step 4. Find the optimal cuts

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

- example:

$$((A_1(A_2A_3))((A_4A_5)A_6))$$

Example: Given 6 matrices to multiply

$$
\begin{array}{lll}
A_1 & 30 \times 35 & = p_0 \times p_1 \\
A_2 & 35 \times 15 & = p_1 \times p_2 \\
A_3 & 15 \times 5 & = p_2 \times p_3 \\
A_4 & 5 \times 10 & = p_3 \times p_4 \\
A_5 & 10 \times 20 & = p_4 \times p_5 \\
A_6 & 20 \times 25 & = p_5 \times p_6
\end{array}
$$

Find the cheapest way to multiply these 6 matrices

- Show how to multiply this matrix chain optimally

Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

- Brute force Solution
  - Minimum cost 15 125
  - Optimal parenthesization $((A_1(A_2A_3))((A_4 A_5)A_6))$

The m and s table computed by MATRIX-CHAIN-ORDER for n=6

((A1(A2A3))((A4 A5)A6))

Example, computation of m[2,5]

$m[2,5] = 7125 =$

Minimum of the 3 cuts {

$m[2,2]+m[3,5]+p_1p_2p_5 = 0+2500+35\times15\times20 = 13000,$

$m[2,3]+m[4,5]+p_1p_3p_5 = 2625+1000+35\times5\times20 = 7125,$

$m[2,4]+m[5,5]+p_1p_4p_5 = 4375+0+35\times10\times20 = 11374$

}

Example using the C program mcm.exe:

Input: A1[30x35] A2[35x15] A3[15x5] A4[5x10] A5[10x20] A6[20x25]

```
./mcm.exe 30 35 15 5 10 20 25
```

order of filling m is:

```
 |      1|      2|      3|      4|      5|      6
-+-------+-------+-------+-------+-------+-------+
1:      0       1       7      15      22      29
2:              0       2       9      17      24
3:                      0       3      10      18
4:                              0       4      12
5:                                      0       5
6:                                              0
```

Example: continued **A1[30x35] A2[35x15] A3[15x5] A4[5x10] A5[10x20] A6[20x25]**

m is:

	1	2	3	4	5	6
1:	0	15750	7875	9375	11875	15125
2:		0	2625	4375	7125	10500
3:			0	750	2500	5375
4:				0	1000	3500
5:					0	5000
6:						0

Example: continued **A1[30x35] A2[35x15] A3[15x5] A4[5x10] A5[10x20] A6[20x25]**

s is:

```
 |      1|      2|      3|      4|      5|      6
-+-------+-------+-------+-------+-------+-------+
1:      0       1       1       3       3       3
2:              0       2       3       3       3
3:                      0       3       3       3
4:                              0       4       5
5:                                      0       5
6:                                              0
```

Elements of dynamic programming

- Optimal substructure
- Overlapping subproblems
- How memoization might help

Exercise

mcm 1 2 3 4

A1[1x2]
A2[2x3]
A3[3x4]

Solution

```
mcm 1 2 3 4; A1[1x2] A2[2x3] A3[3x4]

m is:
 |      1|      2|      3
-+-------+-------+-------+
1:      0       6      18
2:              0      24
3:                      0
s is:
 |      1|      2|      3
-+-------+-------+-------+
1:      0       1       2
2:              0       2
3:                      0
Minimum mult is 18 by:(( A1 A2) A3)
```

Exercise

mcm 3 4 1 2

A1[3x4] A2[4x1] A3[1x2]

Exercise Solution

```
A1[3x4] A2[4x1] A3[1x2]

m|       1|       2|       3
-+--------+--------+--------+
1:       0       12       18
2:                0        8
3:                         0

s|       1|       2|       3
-+--------+--------+--------+
1:       0        1        2
2:                0        2
3:                         0
Minimum mult is 18 by:(( A1 A2) A3)
```

Exercise

mcm.exe 1 5 3 2 1

A1[1x5]* A2[5x3]* A3[3x2]* A4[2x1]

```
m is:
 |      1|      2|      3|      4
-+-------+-------+-------+-------+
1:      0      15      21      23
2:              0      30      21
3:                      0       6
4:                              0

s is:
 |      1|      2|      3|      4
-+-------+-------+-------+-------+
1:      0       1       2       3
2:              0       2       2
3:                      0       3
```

Minimum mult is 23 by:(((A1 A2) A3) A4)

Exercise Solution

```
A1[1x5]* A2[5x3]* A3[3x2]* A4[2x1]
-----------------------------------
m|      1|      2|      3|      4
-+-------+-------+-------+-------+
1:      0      15      21      23
2:              0      30      21
3:                      0       6
4:                              0
-----------------------------------
s|      1|      2|      3|      4
-+-------+-------+-------+-------+
1:      0       1       2       3
2:              0       2       2
3:                      0       3
```

Minimum mult is 23 by:(((A1 A2) A3) A4)

Graph algorithms

- Breadth-first search (BFS) and depth-first search (DFS) are two distinct orders in which to visit the vertices and edges of a graph.

BFS

- The BFS begins at a root node and inspects all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on.

procedure BFS

```
procedure BFS(G,v):
    create a queue Q
    enqueue v onto Q
    mark v
    while Q is not empty:
        t ← Q.dequeue()
        if t is what we are looking for:
            return t
        for all edges e in G.adjacentEdges(t) do
            u ← G.adjacentVertex(t,e)
            if u is not marked:
                mark u
                enqueue u onto Q
```

BFS animation

BFS radiates out from a root to visit vertices in order of their distance from the root. Thus closer nodes get visited first.

DFS

- **Depth-first search** (**DFS**) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

Animation of DFS on a tree

procedure DFS

```
procedure DFS(G,v):
    label v as explored
    for all edges e in G.adjacentEdges(v) do
        if edge e is unexplored then
            w ← G.adjacentVertex(v,e)
            if vertex w is unexplored then
                label e as a discovery edge
                recursively call DFS(G,w)
            else
                label e as a back edge
```

DFS on a graph - animation

DFS prefers to visit undiscovered vertices immediately, so the search trees tend to be deeper rather than balanced as with BFS. Notice that the DFS consists of three ``Hamiltonian'' paths, one in each component -- while the BFS tree has far more degree-3 nodes, reflecting balance.

Breadth First Search

FIFO Queue

Breadth First Search

enqueue source node

front **A**

FIFO Queue

Breadth First Search

dequeue next vertex

front **A**

FIFO Queue

Breadth First Search

visit neighbors of A

front

FIFO Queue

Breadth First Search

visit neighbors of A

front

FIFO Queue

Breadth First Search

B discovered

front **B**

FIFO Queue

Breadth First Search

visit neighbors of A

front **B**

FIFO Queue

Breadth First Search

I discovered

front **B I**

FIFO Queue

Breadth First Search

finished with A

front **B I**

FIFO Queue

Breadth First Search

dequeue next vertex

front **B I**

FIFO Queue

Breadth First Search

visit neighbors of B

front **I**

FIFO Queue

Breadth First Search

visit neighbors of B

front **I**

FIFO Queue

Breadth First Search

F discovered

front **I F**

FIFO Queue

Breadth First Search

visit neighbors of B

front **I F**

FIFO Queue

Breadth First Search

A already discovered

front **I F**

FIFO Queue

Breadth First Search

finished with B

front **I F**

FIFO Queue

Breadth First Search

dequeue next vertex

front **I F**

FIFO Queue

Breadth First Search

visit neighbors of I

front **F**

FIFO Queue

Breadth First Search

visit neighbors of I

front **F**

FIFO Queue

Breadth First Search

A already discovered

front **F**

FIFO Queue

Breadth First Search

visit neighbors of I

front **F**

FIFO Queue

Breadth First Search

E discovered

front **F E**

FIFO Queue

Breadth First Search

visit neighbors of I

front **F E**

FIFO Queue

Breadth First Search

F already discovered

front **F E**

FIFO Queue

Breadth First Search

I finished

front **F E**

FIFO Queue

Breadth First Search

dequeue next vertex

front **F E**

FIFO Queue

Breadth First Search

visit neighbors of F

front **E**

FIFO Queue

Breadth First Search

G discovered

front **E G**

FIFO Queue

Breadth First Search

F finished

front **E G**

FIFO Queue

Breadth First Search

dequeue next vertex

front **E G**

FIFO Queue

Breadth First Search

visit neighbors of E

front **G**

FIFO Queue

Breadth First Search

E finished

front **G**

FIFO Queue

Breadth First Search

dequeue next vertex

front **G**

FIFO Queue

Breadth First Search

visit neighbors of G

front

FIFO Queue

Breadth First Search

C discovered

front **C**

FIFO Queue

Breadth First Search

visit neighbors of G

front **C**

FIFO Queue

Breadth First Search

H discovered

front **C H**

FIFO Queue

Breadth First Search

G finished

front **C H**

FIFO Queue

Breadth First Search

dequeue next vertex

front **C H**

FIFO Queue

Breadth First Search

visit neighbors of C

front **H**

FIFO Queue

Breadth First Search

D discovered

front **H D**

FIFO Queue

Breadth First Search

C finished

front **H D**

FIFO Queue

Breadth First Search

get next vertex

front **H D**

FIFO Queue

Breadth First Search

visit neighbors of H

front **D**

FIFO Queue

Breadth First Search

finished H

front **D**

FIFO Queue

Breadth First Search

dequeue next vertex

front **D**

FIFO Queue

Breadth First Search

visit neighbors of D

front

FIFO Queue

Breadth First Search

D finished

front

FIFO Queue

Breadth First Search

dequeue next vertex

front

FIFO Queue

Breadth First Search

STOP

front

FIFO Queue

Directed Depth First Search

Adjacency Lists

A:	F G
B:	A H
C:	A D
D:	C F
E:	C D G
F:	E
G:	
H:	B
I:	H

Directed Depth First Search

Function call stack:

Directed Depth First Search

Function call stack:

Directed Depth First Search

Function call stack:

Directed Depth First Search

Function call stack:

Directed Depth First Search

Function call stack:

Directed Depth First Search

Directed Depth First Search

Directed Depth First Search

Directed Depth First Search

Function call stack:

Directed Depth First Search

Function call stack:

Directed Depth First Search

Function call stack:

Directed Depth First Search

Function call stack:

Directed Depth First Search

Function call stack:

Directed Depth First Search

Function call stack:

Directed Depth First Search

Function call stack:

Directed Depth First Search

Function call stack:

Directed Depth First Search

Nodes reachable from A: A, C, D, E, F, G

Undirected Depth First Search

Adjacency Lists

```
A:  F C B G
B:  A
C:  A
D:  F E
E:  G F D
F:  A E D
G:  E A
H:  I
I:  H
```

Undirected Depth First Search

F newly discovered

Stack

Undirected Depth First Search

A already marked

Undiscovered
Marked
Active
Finished

Stack

Undirected Depth First Search

Undirected Depth First Search

G newly discovered

Undiscovered
Marked
Active
Finished

Stack

Undirected Depth First Search

E already marked

Stack

Undiscovered
Marked
Active
Finished

Undirected Depth First Search

A already marked

Stack

Undiscovered
Marked
Active
Finished

Undirected Depth First Search

Undirected Depth First Search

F already marked

Undiscovered
Marked
Active
Finished

Stack

Undirected Depth First Search

Undirected Depth First Search

F already marked

Undiscovered
Marked
Active
Finished

Stack

Undirected Depth First Search

Undirected Depth First Search

Undirected Depth First Search

Undirected Depth First Search

D already marked

Undiscovered
Marked
Active
Finished

Stack

Undirected Depth First Search

Undiscovered
Marked
Active
Finished

Undirected Depth First Search

Undirected Depth First Search

A already marked

Undiscovered
Marked
Active
Finished

Stack

Undirected Depth First Search

Undirected Depth First Search

Stack

Undirected Depth First Search

A already marked

Undiscovered
Marked
Active
Finished

Stack

Undirected Depth First Search

Undirected Depth First Search

G already finished

Undiscovered
Marked
Active
Finished

Stack

Undirected Depth First Search

Undirected Depth First Search

Undiscovered
Marked
Active
Finished

Stack

Undirected Depth First Search

Undirected Depth First Search

H already marked

Undiscovered
Marked
Active
Finished

Undirected Depth First Search

Undirected Depth First Search

Undirected Depth First Search

Undiscovered
Marked
Active
Finished

Stack

Greedy Algorithms

- MST is Minimum Spanning tree
- Used to find cheapest network of road/wiring/distribution to build
- Algorithms
  - Kruskal MST
  - Prims MST

What is the Greedy Approach?

- **Greedy approach used to solve optimization problems to find a feasible solution that either maximizes or minimizes a given objective functions.**
  - it builds up a solution in *steps,* each step increases the size of the partial solution
  - At each step, it chooses a decision that *optimize (maximize/ minimize)* some *underlying criterion*
- **consists of a simple iterative procedure that tries to find a local optimal solution.**
- **In some instances, these local optimal solutions translate to global optimal solutions. In others, they fail to give optimal solutions.**

Spanning Tree

- **A tree : a connected graph without a cycle.**
- **Tree properties:**
  - A tree of n vertices has n-1 edges
  - There exists a unique path between any two vertices of a tree.
  - Adding any edge to a tree creates a cycle.
- Spanning tree: a tree that spans the graph (has all vertices of the graph).
- The Graph can have multiple spanning trees.
- We are interested in the minimum weight tree.

MST Input Output

Input:

- a Weighted Undirected Graph G = (V,E,w): each edge (v, u) $\in$E has a weight w(u, v).

Output required:

- Spanning tree $E_T \subseteq E_G$ that connects all of the vertices of G
- Minimum weight T

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

MST Example

MST Construction

- maintain a set of tree-edges E_T (initially empty.)
- Add Edges one at a time to E_T.
- Make sure that adding any edge will not form a cycle.
- After adding a safe edge, add its weight to the total tree weight.
- Algorithms
  - Kruskals Algorithm
  - Prims Algorithm

Kruskal Algorithm Animation

Minimum Spanning Tree

www.combinatorica.com

Kruskals Algorithm

1. $E_T = \phi$
2. Sort E_G in increasing order by weight w
3. For each edge (u,v) in the sorted list

   If union(E_T, (u,v)) makes no cycle
   - Add (u,v) to E_T
   - $W_T = W_T + w(u,v)$

   Else discard that edge
4. Return E_T;

The algorithm runs in O(|E| log |E|)

Kruskal Example

Prims Algorithm

- Maintains a set of vertices V_T already in the spanning tree
- The tree starts from an arbitrary root vertex ***r*** and grows until the tree spans all the vertices in V_G
- At each step, the min weight edge that is adjacent to the already chosen nodes in V_T is added to the tree.

Prims Algorithm

1. $E_T = \phi$
2. V_T contains an arbitrary node.
3. Repeat
4. For each node adjacent to V_T
   1. V = Select-min(V_G - V_T)
5. If union(E_T, (u,v)) makes no cycle
   - Add-edge (u,v) to E_T
   - Remove-node(V) from V_G
   - Add-node(v) to V_T
   - $W_T = W_T + w$ (u,v)

   Else discard that edge
6. Until V_G is empty
7. Return E_T

Using heaps we can solve in $O((|V| + |E|)\log|V|) = m \log n$

Prim algorithm animation

SET: { }

Prim’s algorithm for Hooking up a 10-home village with phone lines

Greedy Algorithms

- Dijkstra’s shortest path

Single-Source Shortest Paths

- **Given:** A single source vertex in a weighted, directed graph.
- **Want:** to compute a shortest path for each possible destination.
- Approach : **greedy** → Dijkstra's algorithm

Dijkstra's Algorithm

```
INITIALIZE (SOL)          E.G. S = { S };
Q := V[G]; SET OF VERTICIES
for (i = 1; i < n; i++)
      D[i] = G[ S,i];
while Q ≠ ∅ do
      u := Extract-Min(D);
      SOL := SOL ∪ {u}
      Q = Q – {U}
for each vertex v  IN Q
      D[v] = min (D[v], D[u] + G[u, v])
   END FOR
END WHILE
```

What is the performance of the algorithm? $O(n^2)$

EXAMPLE

SOURCE = 1

	1	2	3	4	5
S = {1}	0	10	∞	30	100
S = {1, 2}	0	10	60	30	100
S = {1, 2, 4}	0	10	50	30	90
S = {1, 2, 4, 3}	0	10	50	30	60

S = {1, 2, 4, 3 , 5}

Dijkstra's Shortest Path Algorithm

Find shortest path from s to t.

Dijkstra's Shortest Path Algorithm

S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }

Dijkstra's Shortest Path Algorithm

S = { }
PQ = { s, 2, 3, 4, 5, 6, 7, t }

Dijkstra's Shortest Path Algorithm

S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }

Dijkstra's Shortest Path Algorithm

S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }

Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }

Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }

Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }

Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }

Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }

Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }

Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }

Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }

Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }

Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }

Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }

Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }

Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }

Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }

Bipartite Matching

in Graphs

Bipartite Matching

The Bipartite Marriage Problem:

- There are n boys and n girls.
- For each pair, the pair is either compatible or not.

Goal: to find the maximum number of compatible pairs.

Bipartite Matching

The Bipartite Marriage Problem:

- There are n boys and n girls.
- For each pair, it is either compatible or not.

Goal: to find the maximum number of compatible pairs.

Graph Problem

A graph is **bipartite** if its vertex set can be partitioned into two subsets A and B so that each edge has one endpoint in A and the other endpoint in B.

A **matching** is a subset of edges so that every vertex has degree at most one.

Maximum Matching

The bipartite matching problem:
Find a matching with the maximum number of edges.

A perfect matching is a matching in which every vertex is matched.

The perfect matching problem: Is there a perfect matching?

Application of Bipartite Matching

Job Assignment Problem:

Each person is willing to do a subset of jobs.

Can you find an assignment so that all jobs are taken care of?

In fact, there is an efficient procedure to find such as assignment!

Perfect Matching

Does a perfect matching always exist?

Suppose you work for shadi.com, and your job is to find a perfect matching between 200 men and 200 women. If there is a perfect matching, then you can show it to the shadi.com. But suppose there is no perfect matching, how can you convince your boss of this fact?

Perfect Matching

Does a perfect matching always exist?

Of course not, eg.
If there are more vertices on one side,
then of course it is impossible.

Let N(S) be the neighbours of vertices in S, for every subset S.

If |N(S)| < |S|, then it is impossible to have a perfect matching.

Hall's Theorem:

A Necessary and Sufficient Condition.

It tells you exactly when a bipartite graph does and does-not have a perfect matching.

Hall's Theorem: A bipartite graph $G=(V,W;E)$ has a perfect matching

if and only if $|N(S)| >= |S|$

for every subset S of V and W.

An Application of Max Flow: Maximum Bipartite Matching

Maximum Bipartite Matching

- A bipartite graph is a graph $G=(V,E)$ in which V can be divided into two parts L and R such that every edge in E is between a vertex in L and a vertex in R.
- e.g. vertices in L represent skilled workers and vertices in R represent jobs. An edge connects workers to jobs they can perform.

A matching in a graph is a subset *M* of *E*, such that for all vertices *v* in *V*, at most one edge of *M* is incident on *v*.

A **maximum matching** is a matching of maximum size (maximum number of edges).

not maximum

maximum

A Maximum Matching

- No matching of cardinality 4, because only one of v and u can be matched.
- In the workers-jobs example a maximal-matching provides work for as many people as possible.

Solving the Maximum Bipartite Matching Problem

- Reduce the maximum bipartite matching problem on graph **G** to the max-flow problem on a corresponding flow network **G'**.

- Solve using Ford-Fulkerson method.

Corresponding Flow Network

- To form the corresponding flow network **G'** of the bipartite graph **G**:
  - Add a source vertex s and edges from s to L.
  - Direct the edges in E from L to R.
  - Add a sink vertex t and edges from R to t.
  - Assign a capacity of 1 to all edges.
- Claim: max-flow in **G'** corresponds to a max-bipartite-matching on **G**.

Solving Bipartite Matching as Max Flow

Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$.

Let $G' = (V', E')$ be its corresponding flow network.

If M is a matching in G,

then there is an integer-valued flow f in G' with value $|f| = |M|$.

Conversely if f is an integer-valued flow in G',

then there is a matching M in G with cardinality $|M| = |f|$.

Thus $\max |M| = \max(\text{integer } |f|)$

Does this mean that max |f| = max |M|?

- Problem: we haven't shown that the max flow f(u,v) is necessarily integer-valued.

Integrality Theorem

- If the capacity function c takes on only integral values, then:
  1. The maximum flow f produced by the Ford-Fulkerson method has the property that |f| is integer-valued.
  2. For all vertices u and v the value f(u,v) of the flow is an integer.

- So max|M| = max |f|

Example

$|M| = 3$ ←→ max flow $= |f| = 3$

Conclusion

- Network flow algorithms allow us to find the maximum bipartite matching fairly easily.
- Similar techniques are applicable in other combinatorial design problems.

Example

- Department has n courses and m instructors.
- Every instructor has a list of courses he or she can teach.
- Every instructor can teach at most 3 courses during a year.

- The goal: find an allocation of courses to the instructors subject to these constraints.

Bipartite Matching Applications

Domino Tiling

Add a vertex for each square in the board.

Add an edge for two squares if they are adjacent.

This is a bipartite graph with the black and white squares form the two sides.

A perfect matching in this graph corresponds to a perfect placement of dominos.

Domino tiling

dominos

With Hall's theorem, now you can determine exactly when a partial chessboard can be filled with dominos.

Latin Square

Latin Square: n x n square, the goal is to fill the square with numbers from 1 to n so that:

- Each row contains every number from 1 to n.
- Each column contains every number from 1 to n.

1	2	3	4
3	4	2	1
2	1	4	3
4	3	1	2

Extending a Latin Square

Suppose you are given a partial Latin Square when some rows are already filled in.

2	4	5	3	1
4	1	3	2	5
3	2	1	5	4

Can you always extend it to a Latin Square?

With Hall's theorem, you can prove that the answer is yes.

Sudoku

Given a partial Latin square, we construct a bipartite graph with choices to fill in the next row.

2	4	5	3	1
4	1	3	2	5
3	2	1	5	4

We want to "match" the numbers to the columns.

Add one vertex for each column, and one vertex for each number.

Add an edge between column i and color j if color j can be put in column i.

Extending a Latin Square, using matching

Given a partial Latin square, we construct a bipartite graph to fill in the next row.

2	4	5	3	1
4	1	3	2	5
3	2	1	5	4
1	5	2	4	3

A perfect matching corresponds to a valid assignment of the next row.

If we can always complete the next row, then by induction we are done.

The key is to prove that the bipartite graph always has a perfect matching.

Using Hall's Theorem

Hall's Theorem: A bipartite graph G=(V,W;E) has a perfect matching if and only if |N(S)| >= |S| for every subset S of V and W.

A graph is **k-regular** if every vertex is of degree k.

A 3-regular bipartite graph

Using Hall's Theorem

Hall's Theorem: A bipartite graph G=(V,W;E) has a perfect matching if and only if |N(S)| >= |S| for every subset S of V and W.

Claim: Every k-regular bipartite graph has a perfect matching.

A 3-regular bipartite graph

Using Hall's Theorem

Hall's Theorem: A bipartite graph $G=(V,W;E)$ has a perfect matching if and only if $|N(S)| >= |S|$ for every subset S of V and W.

Claim: Every k-regular bipartite graph has a perfect matching.

To prove this claim using Hall's theorem,
we need to verify $|N(S)| >= |S|$ for every subset S.

Proof by contradiction:

1. Suppose there is a subset S with $|S| > |N(S)|$.
2. All the edges from S go to $N(S)$.
3. There are total $k|S|$ edges from S to $N(S)$.
4. There are at most $k|N(S)|$ edges from $N(S)$ to S.
5. A contradiction.

Completing Latin Square

Claim: Every k-regular bipartite graph has a perfect matching.

2	4	5	3	1
4	1	3	2	5
3	2	1	5	4

The bipartite graphs coming from Latin square are always regular because:

Suppose there are k unfilled rows.

Then each column already has n-k numbers, and so connected to k numbers.

Each number appeared in n-k columns above, and so connected to k columns.

So, the bipartite graph is k-regular, and thus always has a perfect matching.

Applications of Graph Matching

Stable Marriage Problem

What is a Matching?

- Each boy is matched to at most one girl.
- Each girl is matched to at most one boy.

What is a Good Matching?

- ***A* stable matching:** They have no incentive to break up.
  (unstable pairs - Willing to break up and switch).
- ***A* maximum matching:** To maximize the number of pairs matched.

Stable Matching Problem

- There are n boys and n girls.
- For each boy, there is a preference list of the girls.
- For each girl, there is a preference list of the boys.

Boys

1: CBEAD

2 : ABECD

3 : DCBAE

4 : ACDBE

5 : ABDEC

Girls

A : 35214

B : 52143

C : 43512

D : 12345

E : 23415

What is a ***stable*** matching?

Consider the following matching.

It is **unstable**, why?

Unstable pair

- Boy 4 prefers girl C more than girl B (his current partner).
- Girl C prefers boy 4 more than boy 1 (her current partner).

So they have the incentive to leave their current partners, and switch to each other, we call such a pair an **unstable pair**.

What is a ***stable*** matching?

A stable matching is a matching with no unstable pair, and every one is married.

Can you find a stable matching in this case?

Does a stable matching always exists? - Not clear…

Can you find a stable matching in this case?
(continued after Stable Roommate problem)

Stable Marriage, Gale Shapley Algorithm

Stable Matching

Can you now construct an example where there is no stable matching?

not clear…

Gale,Shapley [1962]:

There is always a stable matching in the stable matching problem.

This is more than a solution to a puzzle:

- College Admissions (original Gale & Shapley paper, 1962)
- Matching Hospitals & Residents.
- Matching Dancing Partners.

Shapley received Nobel Prize 2012 in Economics for it!

The proof is based on a marriage procedure…

Stable Matching

Why stable matching is easier than stable roommate?

Intuition: It is enough if we only satisfy one side!

This intuition leads us to a very natural approach.

The Marrying Procedure

Morning: boy propose to their favourite girl

Billy Bob

Brad

Angelina

The Marrying Procedure

Morning: boy propose to their favourite girl
Afternoon: girl rejects all but her favourite

The Marrying Procedure

Morning: boy propose to their favourite girl
Afternoon: girl rejects all but her favourite
Evening: rejected boy writes off girl

This procedure is then repeated until all boys propose to a different girl

Day 1

Morning: boy propose to their favourite girl
Afternoon: girl rejects all but her favourite
Evening: rejected boy writes off girl

Day 2

Morning: boy propose to their favourite girl
Afternoon: girl rejects all but her favourite
Evening: rejected boy writes off girl

Day 3

Morning: boy propose to their favourite girl
Afternoon: girl rejects all but her favourite
Evening: rejected boy writes off girl

Day 4

Morning: boy propose to their favourite girl
Afternoon: girl rejects all but her favourite
Evening: rejected boy writes off girl

OKAY, marriage day!

Proof of Gale-Shapley Theorem

Gale,Shapley [1962]:
This procedure always find a stable matching in the stable marriage problem.

What do we need to check?

1. The procedure will terminate.
2. Everyone is married.
3. No unstable pairs.

Step 1 of the Proof

Claim 1. The procedure will terminate in at most n^2 days with (n boys and n girls) .

1. If every girl is matched to exactly one boy,
   then the procedure will terminate.

2. Otherwise, since there are n boys and n girls,
   there must be a girl receiving more than one proposal.

3. She will reject at least one boy in this case,
   and those boys will write off that girl from their lists,
   and propose to their next favourite girl.

4. Since there are n boys and each list has at most n girls,
   the procedure will last for at most n^2 days.

Step 2 of the Proof

Claim 2. Every one is married when the procedure stops.

Proof: by contradiction.

1. If *B* is not married, his list is empty.
2. That is, *B* was rejected by all girls.
3. A girl only rejects a boy if she already has a more preferable partner.
4. Once a girl has a partner, she will be married at the end.
5. That is, all girls are married (to one boy) at the end, but *B* is not married.
6. This implies there are more boys than girls, a contradiction.

Step 3 of the Proof

Claim 3. There is no unstable pair.

Fact. If a girl G rejects a boy B,
then G will be married to a boy (she likes) better than B.

Consider any pair (B,G).

Case 1. If G is on B's list, then B is married to be the best one on his list.
So B has no incentive to leave.

Case 2. If G is not on B's list, then G is married to a boy she likes better.
So G has no incentive to leave.

Proof of Gale-Shapley Theorem

Gale,Shapley [1962]:
There is always a stable matching in the stable marriage problem.

Claim 1. The procedure will terminate in at most n^2 days.

Claim 2. Every one is married when the procedure stops.

Claim 3. There is no unstable pair.

So the theorem follows.

Optional Questions

Intuition: It is enough if we only satisfy one side!

Is this marrying procedure better for boys or for girls??

- All boys get the best partner simultaneously!
- All girls get the worst partner simultaneously!

Why?

That is, among all possible stable matching,
boys get the best possible partners simultaneously.

Can a boy do better by lying? NO!

Can a girl do better by lying? YES!

Stable Roommate:

matching pairs in graphs

Stable Roommate Problem

- There are 2n people.
- There are n rooms, each can accommodate 2 people.
- Each person has a preference list of 2n-1 people.
- Find a stable matching (match everyone and no unstable pair).

Does a stable matching always exist?
Not clear?

When is it difficult to find a stable matching?

Stable Roommate: No matching if there is a circular dependency

	1	2	3
a	b	c	d
b	c	a	d

- a prefers b more than c
- b prefers c more than a
- c prefers a more than b

Let's match (a & b) and (c & d), Then (b&c) is unstable, because b prefers c more than a, and c prefers b more than d.

Maxflow:

Maximum Network Flow

Network Flows

Important problem in the *optimal management* of *resources*.

Types of Networks

- Internet
- Telephone
- Cell
- Roads, Highways
- Railways
- Electrical Power
- Water
- Sewer
- Gas
- Any more?

Maximum Flow Problem

- How can we maximize the flow in a network from a source or set of sources to a destination or set of destinations?
- The problem reportedly rose to prominence in relation to the rail networks of the Soviet Union, during the 1950's.
- The US wanted to know how quickly the Soviet Union could get supplies through its rail network to its satellite states in Eastern Europe.

Source: lbackstrom, The Importance of Algorithms, at www.topcoder.com

Dual of Max Flow is Min cut

- In addition, the US wanted to know which rails it could destroy most easily to cut off the satellite states from the rest of the Soviet Union.
- It turned out that these two problems were closely related, and that solving the **max flow problem** also solves the **min cut problem** of figuring out the cheapest way to cut off the Soviet Union from its neighbours.

Source: Ibackstrom, The Importance of Algorithms, at www.topcoder.com

Network Flow definitions

- Instance:
  - A Network is a *directed* graph ***G***
  - Edges represent pipes that carry flow
  - Each edge (u,v) has a maximum capacity c(u,v)
  - A source node **s** from which flow arrives
  - A sink node **t** out of which flow leaves
- Example:

Network Flow Problems

What is the max flow in these Graphs?

Network Flow Solutions

What is the max flow in these Graphs?

G1. maxflow =max(1,5,2)=1

G2. maxflow = 3 + 7 = 10

G3. maxflow = 2

G4. maxflow = max(1,3)+max(2,4) = 1+2 = 3

The General Maxflow Problem

- Use a graph to model material that flows through pipes.
- Each edge represents one pipe, and has a capacity, which is an upper bound on the flow rate, in some units / time.
- Think of edges as pipes of different sizes.
- Want to compute max rate that we can ship material from a given source to a given sink.
- Example:

What is a Flow Network?

- Each edge (u,v) has a nonnegative capacity c(u,v).
- If (u,v) is not in E, let c(u,v)=0.
- We have a source s, and a sink t.
- Assume that every vertex v in V is on some path from s to t.
- Here $c(s,v_1)=16$; $c(v_1,s)=0$; $c(v_2,v_3)=0$
- We can write it as a matrix

Constraints on Flow

- For each edge (u,v), the flow f(u,v) is a real-valued function that must satisfy 3 conditions:

  Capacity constraint: $\forall u,v \in V,\ f(u,v) \le c(u,v)$

  Skew symmetry: $\forall u,v \in V,\ f(u,v) = -f(v,u)$

  Flow conservation: $\forall u \in V - \{s,t\},\ \sum_{v \in V} f(u,v) = 0$

- Notes:
  - The skew symmetry condition implies that f(u,u)=0.
  - We show only the *positive* flows in the flow network.

Example of a Flow numbering:

- $f(v_2, v_1) = 1$ of available $c(v_2, v_1) = 4$.
- $f(v_1, v_2) = -1$ of available $c(v_1, v_2) = 10$.
- $f(v_3, s) + f(v_3, v_1) + f(v_3, v_2) + f(v_3, v_4) + f(v_3, t) = 0 + (-12) + 4 + (-7) + 15$
  $= 0$.

The Value of a flow

- The value of a flow is given by

$$\boxed{|f| = \sum_{v \in V} f(s,v) = \sum_{v \in V} f(v,t)}$$

This is the total flow leaving s
= the total flow arriving in t.

Example of flow at s and t

$|f| = f(s, v_1) + f(s, v_2) + f(s, v_3) + f(s, v_4) + f(s, t)$

$= 11 + 8 + 0 + 0 + 0 = 19$

$|f| = f(s, t) + f(v_1, t) + f(v_2, t) + f(v_3, t) + f(v_4, t)$

$= 0 + 0 + 0 + 15 + 4 = 19$

A flow in a network

- We assume that there is only flow in one direction at a time.
- Sending 7 trucks from Edmonton to Calgary and 3 trucks from Calgary to Edmonton has the same net effect as sending 4 trucks from Edmonton to Calgary.

Multiple Sources Network

- We have several sources and several targets.
- Reduce to max-flow by adding a supersource and a supersink:

Residual Networks (available capacity)

- The residual capacity of an edge *(u,v)* in a network with a flow *f* is given by:

$$\boxed{c_f(u,v) = c(u,v) - f(u,v)}$$

- The residual network of a graph *G* induced by a flow *f* is the graph with only the edges having positive residual capacity, i.e.,

$G_f = (V, E_f)$, where $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$

Example of Residual Network

Flow Network:

Residual Network:

Augmenting Paths

- An augmenting path *p* is a simple path from s to t on the residual network.
- We can put more flow from *s* to *t* through *p*.
- We call the maximum capacity by which we can increase the flow on *p* the residual capacity of *p*.

$$\boxed{c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is on } p\}}$$

Augmenting Paths

Network:

Residual Network:

Augmenting path

The residual capacity of this augmenting path is 4.

Computing Max Flow

- Classic Method:
  - Identify an augmenting path
  - Increase flow along that path
  - Repeat until no more paths

Ford-Fulkerson Method

```
Ford FulkersonMethod(G,s,t) {
  f = 0 // initial flow in G, from s to t.
  while ( there exists an augmenting path p
            with spare capacity h ) {
        f = f + h // augment flow along p
    }
  // No more paths, f is maxflow in G.
  return f
}
```

Example

Flow(1)

Residual(1)

No more augmenting paths → max flow attained.

Flow(2)

Residual(2)

Cut

Cuts of Flow Networks

A cut (S,T) of a flow network is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$.

The Net Flow through a Cut (S,T)

$$f(S,T) = \sum_{u \in S, v \in T} f(u,v)$$

- f(S,T) = 12 – 4 + 11 = 19

The Capacity of a Cut (S,T)

$$c(S,T) = \sum_{u \in S, v \in T} c(u,v)$$

- c(S,T)= 12+ 0 + 14 = 26

Augmenting Paths – example

Capacity of the cut

= maximum possible flow through the cut

= 12 + 7 + 4 = 23

Flow(2)

- The network has a capacity of **at most** 23.
- In this case, the network **does** have a capacity of 23, because this is a **minimum cut**.

Net Flow of a Network

- The net flow across any cut is the same and equal to the flow of the network |f|.

Bounding the Network Flow

- The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.

Exercise: list all the cuts and find the min-cut

Exercise: Find maxflow from source to target, and show the mincut

Max-Flow Min-Cut Theorem

- If f is a flow in a flow network $G=(V,E)$, with source s and sink t, then the following conditions are equivalent:
  1. f is a maximum flow in G.
  2. The residual network G_f contains no augmented paths.
  3. $|f| = c(S,T)$ for some cut (S,T) (a min-cut).

Ford-Fulkerson for max flow

Ford-Fulkerson(G)

  $f = \mathbf{0}$

  while($\exists$ simple path p from s to t in G_f)

    $f := f + f_p$

  output f

The Basic Ford-Fulkerson Algorithm

```
FORD-FULKERSON(G, s, t)
for each edge (u, v) ∈ E[G]
      do f[u, v] ← 0
         f[v, u] ← 0
while there exists a path p from s to t in the residual network G_f
      do c_f(p) ← min {c_f(u, v) : (u, v) is in p}
         for each edge (u, v) in p
             do f[u, v] ← f[u, v] + c_f(p)
                f[v, u] ← −f[u, v]
```

Example

augmenting path

Original Network

Flow Network

Resulting Flow = 4

Example

Flow Network

Resulting Flow = 4

Residual Network

augmenting path

Flow Network

Resulting Flow = 11

Example

Flow Network

Resulting Flow = 11

Residual Network

augmenting path

Flow Network

Resulting Flow = 19

Example

Flow Network

Resulting Flow = 19

Residual Network

augmenting path

Flow Network

Resulting Flow = 23

Example

Resulting
Flow = 23

No augmenting path:

Maxflow=23

(e)

Residual Network

Analysis

```
FORD-FULKERSON(G, s, t)
for each edge (u, v) ∈ E[G]
        do f[u, v] ← 0
           f[v, u] ← 0
while there exists a path p from s to t in the residual network G_f
        do c_f(p) ← min{c_f(u, v) : (u, v) is in p}
           for each edge (u, v) in p
                do f[u, v] ← f[u, v] + c_f(p)
                   f[v, u] ← −f[u, v]
```

O(E)

?

O(E)

Analysis

- If capacities are all integer, then each augmenting path raises $|f|$ by ≥ 1.
- If max flow is f^*, then need $\leq |f^*|$ iterations → time is $O(E|f^*|)$.
- Note that this running time is **not polynomial** in input size. It depends on $|f^*|$, which is not a function of $|V|$ or $|E|$.
- If capacities are rational, can scale them to integers.
- If irrational, FORD-FULKERSON might never terminate! But we can't use irrational capacities as inputs to digital computers anyway.

Polynomial time algorithms

- **Defintion:** A ***polynomial time algorithm*** is an algorithm than runs in time polynomial in *n*, where *n* is the ***number of bits*** of the input.

- How we intend to encode the input influences if we have a polynomial algorithm or not. Usually, some “standard encoding” is implied.

- In general, we want Polynomial ¼ Fast
  because Exponential ¼ Slow

Complexity of Ford-Fulkerson

- With standard (decimal or binary) representation of integers, Ford-Fulkerson is an ***exponential*** time algorithm (in size of capacities).
- Example: a capacity 999 can be written in as 3 digits, but it may take 999 iterations.

Complexity of Ford-Fulkerson

- With ***unary*** (4 ~ 1111) representation of integers, Ford-Fulkerson is a ***polynomial*** time algorithm.
- Intuition: When the input is longer it is easier to be polynomial time as a function of the input length.
- An algorithm which is polynomial if integer inputs are represented in unary is called a ***pseudo-polynomial*** algorithm.
- Intuitively, a pseudo-polynomial algorithm is an algorithm which is fast if all numbers in the input are small.

The Basic Ford-Fulkerson Algorithm

- With time O (E |f*|), the algorithm is **not** polynomial.
- This problem is real: Ford-Fulkerson may perform badly if we are unlucky:

|f*|=2,000,000

Run Ford-Fulkerson on this example

Augmenting Path

Residual Network

Run Ford-Fulkerson on this example

Augmenting Path

Residual Network

Run Ford-Fulkerson on this example

- Repeat 999,999 more times…
- Can we do better than this?

Edmonds-Karp max flow algoritm

Implement Ford-Fulkerson by always choosing the ***shortest possible*** augmenting path, i.e., the one with fewest possible edges.

The Edmonds-Karp Algorithm

- A small fix to the Ford-Fulkerson algorithm makes it work in polynomial time.
- Select the augmenting path using **breadth-first search** on residual network.
- The augmenting path *p* is the shortest path from *s* to *t* in the residual network (treating all edge weights as 1).
- Runs in time $O(V E^2)$.

Complexity of Edmonds-Karp

- Each iteration of the while loop can still be done in time $O(|E|)$.

- The number of iterations are now at most $O(|V||E|)$ regardless of capacities – to be seen next.

- Thus, the total running time is $O(|V|\ |E|^2)$ and Edmonds-Karp is a polynomial time algorithm for Max Flow.

Why at most O(|V| |E|) iterations?

When executing Edmonds-Karp, the residual network G_f gradually changes (as f changes). This sequence of different residual networks G_f satisfies:

Theorem

1) The distance between s and t in G_f ***never decreases*:** After each iteration of the while-loop, it either increases or stays the same.
2) The distance between s and t in G_f can stay the same for at most $|E|$ iterations of the while-loop before increasing.

As the distance between s and t can never be more than $|V|$-1 and it starts out as at least 1, it follows from the theorem that we have at most $(|V|-2)|E|$ iterations.

The Edmonds-Karp Algorithm - example

- The Edmonds-Karp algorithm halts in only 2 iterations on this graph.

Max flow algorithms and their complexity

- Fulkerson Ford $O(E|f^*|)$.. psuedo-poly
- Edmonds Karp $O(V E^2)$.
- Push-relabel $O(V^2 E)$ (CLRS, 26.4)
- The relabel-to-front $O(V^3)$ (CLRS, 26.5)

Maxflow Animation

of

Ford-Fulkerson Algorithm

Ford-Fulkerson Algorithm

Flow value = 0

Ford-Fulkerson Algorithm

Flow value = 0

Ford-Fulkerson Algorithm

Flow value = 8

Ford-Fulkerson Algorithm

G:

Flow value = 10

G_f:

Ford-Fulkerson Algorithm

G:

Flow value = 16

G_f:

Ford-Fulkerson Algorithm

Flow value = 18

Ford-Fulkerson Algorithm

Flow value = 19

Ford-Fulkerson Algorithm

Cut capacity = 19

Flow value = 19

String Searh and Pattern Matching

(naïve algorithm)

Uses

- Text search using text-editor/word-processor
- Web search
- Packet filtering
- DNA sequence matching
- C program: strstr(P, T)

String Searching

- String searching problem is to find the location of a specific text pattern within a larger body of text (e.g., a sentence, a paragraph, a book, etc.).
- Problem: given a string T and a pattern P, find if and where P occurs in T.
- Example: Find P="n th" in T="the rain in spain stays mainly on the plain"
- We want fast, save space, efficiency.

Substring, prefix, suffix

- Given a string S of size m = S[0..m-1].
- A *substring* S[i .. j] of S is the string fragment between indexes i and j inclusive.
- A *prefix* of S is a substring S[0 .. i]

  w pre x, w ● x, w is a prefix of x,

  e.g. aba pre abaz.
- A *suffix* of S is a substring S[i .. m-1]

  w suf x, w ✿ x, w is a suffix of x,

  e.g. abc ✿ zzabc.
- i and j lie between 0 and m-1, inclusive.

Examples of prefixes and suffixes

S =

a	n	d	r	e	w
0					5

- Substring S[1..3] is "ndr"
- Prefixes of S are:
  - "andrew", "andre", "andr", "and", "an", "a"
- Suffixes of S are:
  - "andrew", "ndrew", "drew", "rew", "ew", "w"

String matching algorithms

- Naive Algorithm
- RK: Rabin-Karp Algorithm
- String Matching using Finite Automata
- KMP: Knuth-Morris-Pratt Algorithm
- BM: Boyer Moore

Algorithm	Preprocessing Time	Matching Time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite Automaton	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$
Boyer-Moore	$\Theta(m)$	$\Theta(n)$

KMP: Knuth-Morris-Pratt Algorithm

- KMP Algorithm has two parts: (1) PREFIX function (2) Matcher.
- Information stored in prefix function to speed up the matcher.
- Running time:

  Creating PREFIX function takes $O(m)$

  MATCHER takes $O(m+n)$ (instead of O(m n)).

Boyer-Moore Algorithm

- Published in 1977
- The longer the pattern is, the faster it works
- Starts from the end of pattern, while KMP starts from the beginning
- Works best for character string, while KMP works best for binary string
- KMP and Boyer-Moore

  - Preprocessing existing patterns

  - Searching patterns in input strings

Automata and Patterns: Regular Expressions

- notation for describing a set of strings, possibly of infinite size
- ε denotes the empty string.
- a + c denotes the set {a, c} (a or c).
- a* denotes the set {ε, a, aa, aaa, ...}, 0 or more a.
- Examples

  (a+b)* all the strings from the alphabet {a,b}

  b*(ab*a)*b* strings with an even number of a's

  (a+b)*sun(a+b)* strings containing the pattern "sun"

  (a+b)(a+b)(a+b)a 4-letter strings ending in a

Alphabet

- Elements of *T* and *P* are characters from a *finite alphabet* Σ
- The *pattern* is in an array *P* [1..*m*]
- The *text* is in an array *T* [1..*n*]
- E.g., $\Sigma = \{0,1\}$ or $\Sigma = \{a, b, \ldots, z\}$
- Usually *T* and *P* are called *strings* of characters
- ε (epsilon or lambda is the empty string)

String Matching

Given: Two strings P[1..m] and T[1..n] over alphabet Σ.

Find all occurrences of P[1..m] in T[1..n]

Example: $\Sigma = \{a, b, c\}$

Terminology:

P occurs with shift s in T

P occurs beginning at position s+1.

s is a valid shift.

Goal: Find <u>all</u> valid shifts

Example 1, valid shift is a match

shift s = 3, is a valid shift
(n=13, m=4 and $0 \le s \le n\text{-}m$ holds)

Example 2, another match

Naive Brute-Force String matching

```
Naïve(T, P)
    n := length[T];
    m := length[P];
    for s := 0 to n – m do
        if P[1..m] = T[s+1..s+m] then
            print "pattern occurs with shift s"
    done
```

Running time is $\Theta((n - m + 1)m)$.
Bound is tight. Consider: $T = a^n$, $P = a^m$.

Exercise

Q1. Given p="abab", t="abababa".

How many places does p match with t?

Q2. Given p="xx", t="xxxx".

How many places does this match?

Exercise

- Write the C function to find pattern in text:

```
char *sfind(char *pattern, char
  *text)
```

  1. to return NULL if p does not occur in t, or
  2. to return pointer into t, where p occurs.

- Also read **`strstr`** C-function documentation, and compare strstr wit sfind.

Solution: using sfind

```
1. #include <stdio.h>
2. char *sfind(char *pat, char *text);
3. int main( ) {
4.   char *const pattern = "ab", *text =  "abababa";
5.   char *t = text, *found;   int i=0;
6.   printf("Searching p='%s' in t='%s'\n", pattern,
   t);
7.   while( found = sfind(pattern, t)) {
8.      printf("%2d Found in text[%2d..]=%s\n",
9.         ++i, found-text, found);
10.     t=found+1;
11.   }
12.   return 0;
13.}
```

Solution: sfind

```
char *sfind(char *pat, char *text) {
   char *t = text;
   while(*t) {
     char *q=pat, *s=t;
     while(*q && *s && *q == *s)
       q++, s++;
     if(!*q)  // reached end of q?
        return t;
     t++;
   }
   return 0;
}
```

Example

a	c	a	a	b	c

s = 0

a	a	b

Example

Example

Example

a	c	a	a	b	c

s = 1 → | a | a | b |
|---|---|---|

Example

Example

a	c	a	a	b	c

s = 2 → | a | a | b |
|---|---|---|

Example

Example

Example

match!

Example

Example

Example

Worst-case Analysis

- Brute force pattern matching can take $O(m\ n)$ in the worst case.
- But most searches of ordinary text is fast: $O(m+n)$.
- There are m comparisons for each shift into $n-m+1$ shifts: worst-case time is $\Theta(m(n-m+1))$

Why is naive matching slow?

- Naive method is inefficient because information from a mismatch is not used to jump upto m-steps forward.
- The brute force algorithm is fast when the alphabet of the text is large
  - e.g. {A..Z, a..z, 0..9, ..}
- It is slower when the alphabet is small
  - e.g. binary, image, DNA {G,C,A,T}.

continued

Worst and Average case examples

- Example of a worst case:
  - P: "aaah"
  - T: "aaaaaaaaaaaaaaaaaaaaaaaaaaaah"

- Example of a more average case:
  - P: "store"
  - T: "a string searching example is standard"

DNA are strings over {C,G,A,T}
RNA are strings over {C,G,A,U}

DNA Matching

- The main objective of DNA analysis is to get a visual representation of DNA left at the scene of a crime.
- A DNA "picture" features columns of dark-colored parallel bands and is equivalent to a fingerprint lifted from a smooth surface.
- To identify the owner of a DNA sample, the DNA "fingerprint," or profile, must be matched, either to DNA from a suspect or to a DNA profile stored in a database.

Homework: **`lastfind(s,t)`**

Write a C function, lastfind(char *s, char*t) to return the ptr to last occurrence of s in t.

In the main program call lastfind( ) with various inputs, and assert the expected output. E.g.

```
assert(lastfind(s="ab",t="abcabc"), t+3);
assert(lastfind("ab","ba")==NULL);
```

Homework: Write brute force string search function

- Write find2(char * source, char *target) to find the 2nd occurrence of source in target.
- E.g. find2("xy", t="axybxyc") returns t+4
- E.g. find2("x", t="x") return NULL
- E.g. find2("x", t="xx") returns t+1.

Rabin Karp String Matching Algorithm

Rabin-Karp Algorithm

We can treat the alphabet as numbers $\Sigma = \{0, 1, 2, \ldots, 9\}$ = digits instead of characters.

So we can view P as a decimal number.

We can also treat substrings of text as decimal numbers.

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

Rabin-Karp

- The Rabin-Karp string searching algorithm calculates a **hash value** for the pattern, and a hash for each M-character subsequence of text to be compared.
- If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence.
- Else the hash values are equal, the algorithm will do a Brute Force comparison between the pattern and the M-character sequence.
- In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.

Rabin-Karp Example

- Hash value of “AAAAA” is 37
- Hash value of “AAAAH” is 100

```
1) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH
   37≠100      1 comparison made
2) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
     AAAAH
   37≠100      1 comparison made
3) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
       AAAAH
   37≠100      1 comparison made
4) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
         AAAAH
   37≠100      1 comparison made

    ...

N) AAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
                                AAAAH
    5 comparisons made           100=100
```

Modular Equivalence

if *(a mod q) = (b mod q),* we say:

"a is equivalent to b, modulo q" , we write:

$a \equiv b \pmod{q}$, i.e. *a* and *b* have the same remainder when divided by q.

E.g. $23 \equiv 37 \equiv -19 \pmod{7}$.

The **mod** function (%) has nice properties:

1 $[(x \% q) + (y \% q)] \% q \equiv (x + y) \% q$

2 $[(x \% q) * (y \% q)] \% q \equiv (x * y) \% q$

3 $(x \% q) \% q \equiv x \% q$

Hashing a string

- The alphabet consists of 10 letters = { a, b, c, d, e, f, g, h, i, j }
- Let "a" correspond to 1, "b" to 2 ..
- hash("cah") would be:

$$c*100 + a*10 + h*1$$

$$3*100 + 1*10 + 8*1 = 318$$

Rabin-Karp, incremental mod

We don't compute a new value.

We simply adjust the existing value as move right.

Given x(i), we can compute x(i+1) for the next subsequence t[i+1 .. i+M] in constant time, as follows:

```
h(i)=( (t[i]  * b^M-1    % q)
       + (t[i+1] * b^M-2 % q) + ...
         +(t[i+M-1]        % q))  % q

h(i+1) =( h(i) * b   % q          # Shift left one digit
            - t[i] * b^M % q      # Subtract leftmost digit
             +t[i+M]    % q )     # Add new rightmost digit
                        % q
```

How incremental hash mod 13 is computed

$$14152 \equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13}$$

$$\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13}$$

$$\equiv 8 \pmod{13}$$

Spurious match

Example

3 1 4 1 5
pattern
mod 13
7
text
1 2 3 4 5 6 7 8 9 10 11 12 13 14
2 3 1 4 1 5 2 6 7 3 9 9 2 1
mod 13
1 7 8 4 5 10 11 7 9 11
valid match
spurious hit

Rabin Karp Algorithm

•RK: Rabin Karp
T: Text
P: Pattern
d: power, e.g. 10
h: 10^m-1, e.g. 10000
q: modulus, e.g. 13

•Compute initial hashes

•Avoid spurious hits by explicit check whenever there is a potential match.

•Compute incremental hash as we go right

```
RK(T, P, d, q)
    n := length[T];
    m := length[P];
    h := d^(m-1) mod q;
    p := 0;
    t_0 := 0;
    for i := 1 to m do
        p := (dp + P[i]) mod q;
        t_0 := (dt_0 + P[i]) mod q
    od;
    for s := 0 to n – m do
        if p = t_s then
            if P[1..m] = T[s+1..s+m] then
                print "pattern occurs with shift s"
            fi
        fi;
        if s < n-m then
            t_(s+1) := (d(t_s – T[s+1]h) + T[s+m+1]) mod q
        fi
    od
```

Rabin-Karp Complexity

- $O(n*m)$ in Worst case, if every shift has to be verified.
- $O(n+m)$ in Average case, using a large prime number in *hash function*, less collisions.

where

- n = size(Text),
- m = size(Pattern).

Homework: Write **`Rabin Karp String Search`**

Write a C function **`rkfind(s,t)`** to search string s in t.

Let hash(str) be the sum of chars in str mod 256.

In the main function, check these:

```
assert(rkfind(s="ab", t="xabc") == t+1);
assert(rkfind("ab", "baa")==NULL);
```

Homework: Rabin Karp String search

- Implement Rabin karp string search in C.
- Use xor(chars of the string) as the hash function.

SUFFIX TREES

From exact to approximate string matching.

Luca Bortolussi

Exact String matching: a brief history

- Naive algorithm
- Knuth-Morris-Pratt (1977)
- Boyer-Moore (1977)
- Suffix Trees: Weiner (1973),

  McCreight (1978), Ukkonen (1995)

Naive Algorithm

Knuth-Morris-Pratt

Boyer-Moore

Maximum between:

- the bad character rule
- the good suffix rule

Suffix Trees

Definition: *A suffix tree for a string T of length m is a rooted tree such that:*

1. *It has exactly m leafs, numbered from 1 to m;*
2. *Every edge has a label, which is a substring of T;*
3. *Every internal node has at least two children;*
4. *Labels of two edges starting at an internal node do not start with the same character;*
5. *The label of the path from the root to a leaf numbered I is the suffix of T starting at position i, i.e. T[i..m]*

Suffix Trees - II

abbcbab#

Suffix Trees – searching a pattern

abbcbab#

Pattern: **bcb**

Suffix Trees – naive construction

Suffix Trees – Ukkonen Algorithm

Ukkonen algorithm was published in 1995, and it is the fastest and well performing algorithm for building a suffix tree in linear time.

The basic idea is constructing iteratively the **implicit suffix trees** for S[1..i] in the following way:

Construct tree$_1$
for *i = 1* to *m-1* // do **phase *i+1***
  for *j = 1* to *i+1* // do **extension *j***
    1. find the end of the path from the root with label *S[j…i]* in the current tree.
    2. Extend the path adding character *S(i+1),* so that *S[j…i+1]* is in the tree.

Ukkonen: 3 extension rules

The extension will follow one of the next three rules, being $\beta = S[j..i]$:

1. β ends at a leaf. Add $S(i+1)$ at the end of the label of the path to the leaf
2. There's one path continuing from the end of β, but none starting with $S(i+1)$. Add a node at the end of β and a path stating from the new node with label $S(i+1)$, terminating in a leaf with number j.
3. There's one path from the end of β starting with $S(i+1)$. In this case do nothing.

Ukkonen Algorithm - II

The main idea to speed up the construction of the tree is the concept of **suffix link**.

Suffix links are pointers from a node v with path label $x\alpha$ to a node $s(v)$ with path label α (α is a string and x a character).

The interesting feature of suffix trees is that every internal node, except the root, **has** a suffix link towards another node.

abbcbab#

Suffix link

Suffix Trees – Ukkonen Algorithm - III

With suffix links, we can speed up the construction of the ST

In addition, every node can be crossed in costant time, just keeping track of the label's length of every single edge. This can be done because no two edges exiting from a node can start with the same character, hence a single comparison is needed to decide which path must be taken.

Anyway, using suffix links, complexity is still quadratic.

Ukkonen – speed up from n^2 to n

Storing the path labels explicitly will cost a quadratic space. Anyway, each edge need only costant space, i.e. two pointers, one to the beginning and one to the end of the substring it has as label.

To complete the speed up of the algorithm, we need the following observations:
Once a leaf is created, it will remain forever a leaf.
Once a phase rule 3 is used, all succeccive extensions make use of it, hence we can ignore them.
If in phase i the rule 1 and 2 are applied in the first ji moves, in phase i+1 the first ji extensions can be made in costant time, simply adding the character S(i+2) at the end of the paths to the first ji leafs (we will use a global variable e do do this). Hence the extensions will be computed explicitly from ji+1, reducing their global number to 2m.

Generalized Suffix Trees

A generalized suffix tree is simply a ST for a set of strings, each one ending with a different marker. The leafs have two numbers, one identifiing the string and the other identifiing the position inside the string.

S_1 = **abbc$**

S_2 = **babc#**

Longest common substring

Let S_1 and S_2 be two string over the same alphabet. The **<u>Longest Common Substring</u>** problem is to find the longest substring of S_1 that is also a substring of S_2.

Knuth in 1970 conjectured that this problem was $\Theta(n^2)$

Building a generalized suffix tree for S_1 and S_2, to solve the problem one has to identify the nodes which belong to both suffix trees of S_1 and S_2 and choose the one with greatest **string depth** (length of the path label from the root to itself). All these operations cost $O(n)$.

Longest Common Extension

A problem that can be solved linearly using suffix trees is the LCS problem, that is, for every pair of indexes (i,j), finding the length of the longest substring of T starting at position i that matches a substring of P starting at position j.

It can be solved in O(n+m) time, building a generalized suffix tree for T and P, and finding, for every leaf i of T and j of P, their lowest common ancestor in the tree (it can be done in constant time after preprocessing the tree).

Hamming and Edit Distances

Hamming Distance: two strings of the same length are aligned and the distance is the number of mismatches between them.

abbcdbaabbc

abbdcbbbaac

$H = 6$

Edit Distance: it is the minimum number of insertions, deletions and substitutions needed to trasform a string into another.

abbcdbaabbc

cbcdbaabc

abbcdbaabbc

abbcdbaabbc

$E = 3$

The k-mismatches problem

We have a text T and a pattern P, and we want to find occurences of P in T, allowing a maximum of k mismatches, i.e. we want to find all the substring T' of T such that $H(P,T') \leq k$.

We can use suffix trees, but they do not perfom well anymore: the algorithm scans all the paths to leafs, keeping track of errors, and abandons the path if this number becomes greater that k.

The algorithm is made faster by using the LCS. For every suffix of T, the pieces of agreement between the suffix and P are matched together until P is exausted or the errors overcome k. Every piece is found in costant time. The complexity of the resulting algorithm is $O(k|T|)$.

aaacaabaaaaa….

aabaab

An occurence is found in position 2 of T, with one error.

Inexact Matching

In biology, inexact matching is very important:

• Similarity in DNA sequences implies often that they have the same biological function (viceversa is not true);

• Mutations and error transcription make exact comparison not very useful.

There are a lot of algorithms that deal with inexact matching (with respect to edit distance), and they are mainly based on dynamic programming or on automata. Suffix trees are used as a secondary tools in some of them, because their structure is inadapt to deal with insertions and deletions, and even with substitutions.

The main efforts are spend in speeding up the average behaviour of algorithms, and this is justified because of the fact that random sequences often fall in these cases (and DNA sequences have an high degree of randomness).

Dynamic Programming

How to compute edit distance (global alignments) between two string S and T?

The main idea is computing the edit distance between any of the prefixes of S and T.

Let $D(i,j)$ be this distance. Of course, the edit distance between S and T is $D(n,m)$, where $n=|P|$ and $m=|T|$.

The following properties hold:

1. $D(i,0) = i$, $D(0,j) = j$;
2. $D(i,j) = \min \{ D(i,j-1) + 1, D(i-1,j) + 1, D(i-1,j-1) + t(i,j) \}$.

Hence in **O(mn)** time we can compute a matrix which encodes not only the edit distance, but also the way to trasform a string into another (just keeping track, by means of pointers, of which elements realize the minimum)

Dynamic Programming II

			C	A	S	E
		0	1	2	3	4
	0	0	1	2	3	4
A	1	1	1	1	2	3
R	2	2	2	2	2	3
E	3	3	3	3	3	2

Non-Deterministic Automata

To recognize the approximate occurences of a pattern P in a text T, we can build a non-deterministic automaton for P, and run it with T as input. This leads to faster algorithms for the search, but the problem is building the automaton.

Longest Common Subsequence

The **LCS** between two strings S1 and S2 is the greater number of characters of S1 that can be aligned to S2.

It is a global alignement problem, which is obviously connected with edit distance. Anyway, often it is modelled with a scoring scheme, which gives a positive score to matches and a negative one to mismatches, insertions and substitutions. So the best global alignement is the one which maximizes the total score. Clearly, given the best global alignement, the number of matches is the longest common subsequence solution.

a b b c d a b b a

a b _ c b a b _ a

The k–differences problem

This problem is to find all the occurences of a pattern P in a text T, allowing a maximum number of k insertions, deletions or substitutions.

The **Landau-Vishkin** algorithm solves it in $O(k|T|)$ time, and implements an hybrid dynamic programming tecnique, which uses suffix trees to solve a subproblem.

The algorithm looks for paths in the dynamic programming matrix (which start in the upper row), in particular for **d-paths**, which are paths that specify exactly d mismatches and spaces.

Some of these paths are computed, for d ≤ k, and the ones that reach the bottom row correspond to approximate occurences of P in T, with exactly d mismatches or spaces.

Landau-Vishkin Algorithm

Each diagonal is numbered: the main diagonal is numbered with 0, the upper diagonals with increasing positive integers while the lower diagonals with decreasing negative integers.

A d-path is **farthest reaching diagonal i** if it ends in diagonal i and the index of its ending column is greater than or equal to the one of every other d-path ending in diagonal i.

Landau-Vishkin Algorithm - II

The farthest reaching d-path that ends in diagonal i is one of the following three:

1. (d-1)-path of diagonal $i + 1$, plus a vertical edge plus the maximum extension along diagonal i that corresponds to identical substrings in P and T
2. (d-1)-path of diagonal $i - 1$, plus an horizontal edge plus the maximum extension along diagonal i that corresponds to identical substrings in P and T
3. (d-1)-path of diagonal i, plus a diagonal edge that corresponds to a mismatch plus the maximum extension along diagonal i that corresponds to identical substrings in P and T

The maximum extension between substring of P and T can be done in costant time by means od suffix trees.

Inexact-Matching, a new approach

Suffix trees work very well for exact matching, but they fail when we admit errors in the matching process. This happens because, the only way to find approximate occurences of a pattern, when we search it in a suffix tree, is to walk down every path, keeping track of errors and discarding the paths which overcome the tolerance level previously chosen.

A different approach may be that of defining a different data structure, though similar to suffix trees, which encodes in some way a concept of distance, in particular the Hamming Distance.

A possible way is to shift from alphabeth ✌ to alphabet ✌^{k}, encoding the distance in a relation between letters: two letters are said to be “equivalent” if and only if their Hamming distance is less than a threshold δ.

Equivalence between letters

Let's show and example of this idea of equivalence, with ✌ = $\{0,1\}$ and $k = 3$. So, we can build the following table for A^3:

		a	b	c	d	e	f	g	h
		000	001	010	011	100	101	110	111
a	000	0	1	1	2	1	2	2	3
b	001	1	0	2	1	2	1	3	2
c	010	1	2	0	1	2	3	1	2
d	011	2	1	1	0	3	2	2	1
e	100	1	2	2	3	0	1	1	2
f	101	2	1	3	2	1	0	2	1
g	110	2	3	1	2	1	2	0	1
h	111	3	2	2	1	2	1	1	0

If the distance between two letters is less or equal than 1, we define them equivalent. For example a🕓b, b🕓d, but NOT(a🕓d).

Bundled Suffix Trees

Given this equivalence relation (which is not transitive), we want to incorporate it in a tree structure.

For simplicity, we assume that the tree for the sequence S is the smallest tree which contains, for every substring of S, all the exact paths and all the equivalent paths that can be found in S. For "historical reasons", we will call it a **Bundled Suffix Tree.**

Definition: *A bundled suffix tree* 👌 *for a string* S *of length* m *is a rooted tree such that:*

- *It has exactly* m *leafs, numbered from* 1 *to* m*;*
- *Every edge has a label, which is a substring of* S*;*
- *Every node has a set of labels, which is a subset of* $\{1,2,..,m,\sigma\}$*;*
- *The tree obtained deleting all nodes which do not has* σ *as label is the suffix tree for* S*;*
- *For every substring* P *of* S*, the subtree of* 👌 *rooted at the end of the path labeled with* P *has node labels which union (discarding* σ*) gives all the starting positions of substrings of* S *equivalent to* P*;*
- *In every path from the root to a leaf no two nodes can be labelled with the same number.*

Bundled Suffix Trees - II

abbcda#

SUFFIX TREES

Exact String matching: a brief history

- Naive algorithm
- Knuth-Morris-Pratt 1977.
- Boyer-Moore 1977.
- Suffix Trees: Linear time.
  - Weiner 1973 - Right to left
  - McCreight 1978 - Left to right.
  - Ukkonen 1995 - Online, Left to right.

Suffix Trees Uses

- Suitable for DNA searching because:
  - small alphabet (4 chars),
  - size of T = m is huge (billion)
  - many patterns to search on T.
  - Linear time algorithms (large constants).
- Music database

Finding repeats in DNA

- human chromosome 3
- the first 48 999 930 bases
- 31 min cpu time (8 processors, 4 GB)

- Human genome: $3x10^9$ bases
- Tree(HumanGenome) feasible

DNA string

tgagacggagtctcgctctgtcgcccaggctg
gagtgcagtggcgggatctcggctcactgca
agctccgcctcccgggttcacgccattctcctg
cctcagcctcccaagtagctgggactacagg
cgcccgccactacgcccggctaatttttgtatt
tttagtagagacggggtttcaccgttttagccg
ggatggtctcgatctcctgacctcgtgatccgc
ccgcctcggcctcccaaagtgctgggattaca
ggcgt….

Example: Longest repeat in a DNA?

<u>Occurrences</u> at: 28395980, 28401554r **<u>Length</u>: 2559**

ttagggtacatgtgcacaacgtgcaggtttgttacatatgtatacacgtgccatgatggtgtgctgcacccattaactcgtcatttagcgttaggtatatctccgaat
gctatccctcccccctcccccccaccccacaacagtccccggtgtgtgatgttccccttcctgtgtccatgtgttctcattgttcaattcccacctatgagtgagaac
atgcggtgtttggtttttgtccttgcgaaagtttgctgagaatgatggtttccagcttcatccatatccctacaaaggacatgaactcatcattttttatggctgcata
gtattccatggtgtatatgtgccacattttcttaacccagtctacccttgttggacatctgggttggttccaagtctttgctattgtgaatagtgccgcaataaacatac
gtgtgcatgtgtctttatagcagcatgatttataatccttttgggtatatacccagtaatgggatggctgggtcaaatggtatttctagttctagatccctgaggaatca
ccacactgacttccacaatggttgaactagtttacagtcccagcaacagttcctatttctccacatcctctccagcacctgttgtttcctgactttttaatgatcgcca
ttctaactggtgtgagatggtatctcattgtggttttgatttgcatttctctgatggccagtgatgatgagcatttttcatgtgtttttggctgcataaatgtcttcttttga
gaagtgtctgttcatatccttcgcccacttttgatggggttgtttgtttttttcttgtaaatttgttggagttcattgtagattctgggtattagccctttgtcagatgagtag
gttgcaaaaattttctcccattctgtaggttgcctgttcactctgatggtggtttcttctgctgtgcagaagctctttagtttaattagatcccatttgtcaattttggctttt
gttgccatagcttttggtgttttagacatgaagtccttgcccatgcctatgtcctgaatggtattgcctaggttttcttctagggtttttatggttttaggtctaacatgta
agtctttaatccatcttgaattaattataaggtgtatattataaggtgtaattataaggtgtataattatatattaattataaggtgtatattaattataaggtgtaaggaag
ggatccagtttcagctttctacatatggctagccagttttccctgcaccatttattaaatagggaatcctttccccattgcttgtttttgtcaggtttgtcaaagatcaga
tagttgtagatatgcggcattatttctgagggctctgttctgttccattggtctatatctctgttttggtaccagtaccatgctgttttggttactgtagccttgtagtatag
tttgaagtcaggtagcgtgatggttccagctttgttcttttggcttaggattgacttggcaatgtgggctctttttggttccatatgaactttaaagtagtttttttccaatt
ctgtgaagaaattcattggtagcttgatggggatggcattgaatctataaattaccctgggcagtatggccattttcacaatattgaatcttcctacccatgagcgt
gtactgttcttccatttgtttgtatcctcttttatttcattgagcagtggtttgtagttctccttgaagaggtccttcacatcccttgtaagttggattcctaggtattttattct
ctttgaagcaattgtgaatgggagttcactcatgatttgactctctgtttgtctgttattggtgtataagaatgcttgtgatttttgcacattgattttgtatcctgagacttt
gctgaagttgcttatcagcttaaggagattttgggctgagacgatggggttttctagatatacaatcatgtcatctgcaaacagggacaatttgacttcctcttttcc
taattgaatacccgttatttccctctcctgcctgattgccctggccagaacttccaacactatgttgaataggagtggtgagagagggcatccctgtcttgtgcca
gttttcaaagggaatgcttccagtttttgtccattcagtatgatattggctgtgggtttgtcatagatagctcttattattttgagatacatcccatcaatacctaatttatt
gagagtttttagcatgaagagttcttgaattttgtcaaaggccttttctgcatcttttgagataatcatgtggtttctgtctttggttctgtttatatgctggagtacgtttat
tgattttcgtatgttgaaccagccttgcatcccagggatgaagcccacttgatcatggtggataagctttttgatgtgctgctggattcggtttgccagtattttattg
aggatttctgcatcgatgttcatcaaggatattggtctaaaattctctttttttgttgtgtctctgtcaggctttggtatcaggatgatgctggcctataaaatgagtta
gg

Suffix Trees

- Specialized form of keyword trees
- New ideas
  - preprocess text T, not pattern P
    - O(m) preprocess time
    - O(n+k) search time
      - k is number of occurrences of P in T
  - edges can have string labels

Keyword Tree example

- P = {poet, pope, popo, too}

Suffix Tree

- Take any *m* character string S like "xabxac"
- Set of keywords is the set of suffixes of S
  - {xabxac, abxac, bxac, xac, ac, c}
- Suffix tree `T` is essentially the keyword tree for this set of suffixes of S,
- Changes:
  - Assumption: no suffix is a prefix of another suffix (can be a substring, but not a prefix)
    - Assure this by adding a character $ to end of S
  - Internal nodes except root must have at least 2 children
  - edges can be labeled with strings

Example: Suffix tree for "xabcxac"

- {xabxac, abxac, bxac, xac, ac, c}

Notation

- Label of a path from r (root) to a node v is the concatenation of labels on edges from r to v
- label of a node v is L(v), path label from r to v
- string-depth of v is number of characters in v's label L(v)
- Comment: In constructing suffix trees, we will need to be able to "split" edges "in the middle"

Suffix trees for exact matching

- Build suffix tree T for text.
- Match pattern P against tree starting at root until
  - Case 1, P is completely matched
    - Every leaf below this match point is the starting location of P in T
  - Case 2: No match is possible
    - P does not occur in T

Example: suffix tree match(P, T)

- T = xabxac

  suffixes of T = {xabxac, abxac, bxac, xac, ac, c}
- Pattern P_1 : xa .. matches at 1, 4.
- Pattern P_2 : xb .. no matches.

Running Time Analysis

- Build suffix tree:
  - Ukkonen's Linear time algorithm: O(m)
  - This is preprocessing of data.
- Search time:
  - O(n+k) where k is the number of occurrences of P in T
  - O(n) to find match point if it exists
  - O(k) to find all leaves below match point

Building trees: $O(m^2)$ algorithm

- Initialize
  - One edge for the entire string S[1..m]$
- For i = 2 to m
  - Add suffix S[i..m] to suffix tree
    - Find match point for string S[i..m] in current tree
    - If in “middle” of edge, create new node w
    - Add remainder of S[i..m] as edge label to suffix i leaf
- Running Time
  - O(m-i) time to add suffix S[i..m]

Exercises: build suffix trees

1. Build a suffix tree for: "abbcbab#"
2. Search for "bcb" in the tree.

Suffix Tree - Solution

abbcbab#

Suffix Trees – searching a pattern

abbcbab#

Pattern: **bcb**

Suffix Trees – naive construction

Drawbacks

- Suffix trees consume a lot of space
- It is O(n)
- The constant is quite big
- Notice that if we indeed want to traverse an edge in O(1) time then we need an array of pointers of size $|\Sigma|$ in each node (useful when alphabet is small, e.g. 4 chars in DNA and RNA).

Ukkonen Algorithm
Suffix trees

Ukkonen Algorithm

Ukkonen algorithm [1995] is the fastest and well performing algorithm for building a suffix tree in linear time.

The basic idea is constructing iteratively the **implicit suffix trees** for S[1..i] in the following way:

Construct tree$_1$
for *i = 1* to *m-1* // do **phase *i+1***
  for *j = 1* to *i+1* // do **extension *j***
    1. find the end of the path from the root with label *S[j…i]* in the current tree.
    2. Extend the path adding character *S(i+1),* so that *S[j…i+1]* is in the tree.

Ukkonen: 3 extension rules

The extension will follow one of the next three rules, being b = S[j..i]:

1. b ends at a leaf. Add S(i+1) at the end of the label of the path to the leaf
2. There's one path continuing from the end of b, but none starting with S(i+1). Add a node at the end of b and a path stating from the new node with label S(i+1), terminating in a leaf with number j.
3. There's one path from the end of b starting with S(i+1). In this case do nothing.

Ukkonen Algorithm - II

The main idea to speed up the construction of the tree is the concept of **suffix link**.

Suffix links are pointers from a node v with path label xa to a node s(v) with path label a (a is a string and x a character).

The interesting feature of suffix trees is that every internal node, except the root, **has** a suffix link towards another node.

abbcbab#

Suffix link

Suffix Trees – Ukkonen Algorithm - III

With suffix links, we can speed up the construction of the ST

In addition, every node can be crossed in costant time, just keeping track of the label's length of every single edge. This can be done because no two edges exiting from a node can start with the same character, hence a single comparison is needed to decide which path must be taken.

Anyway, using suffix links, complexity is still quadratic.

Ukkonen – speed up from n^2 to n

Storing the path labels explicitly will cost a quadratic space. Anyway, each edge need only costant space, i.e. two pointers, one to the beginning and one to the end of the substring it has as label.

To complete the speed up of the algorithm, we need the following observations:

Once a leaf is created, it will remain forever a leaf.

Once a phase rule 3 is used, all succeccive extensions make use of it, hence we can ignore them.

If in phase i the rule 1 and 2 are applied in the first ji moves, in phase i+1 the first ji extensions can be made in costant time, simply adding the character S(i+2) at the end of the paths to the first ji leafs (we will use a global variable e do do this). Hence the extensions will be computed explicitly from ji+1, reducing their global number to 2m.

Ukkonen: construct suffix tree in linear time

Ukkonen's *linear* time construction

ACTAATC

A

ACTAATC

AC

ACTAATC

AC

ACTAATC

AC

ACTAATC

ACT

ACTAATC

ACT

ACTAATC

ACT

ACTAATC

ACT

ACTAATC

ACTA

ACTAATC

ACTA

ACTAATC

ACTA

ACTAATC

ACTA

ACTAATC

ACTAA

ACTAATC

ACTAA

ACTAATC

ACTAA

ACTAATC

ACTAA

ACTAATC

ACTAA

Phases & extensions

- Phase i is when we add character i

- In phase i we have i extensions of suffixes

ACTAATC

ACTAAT

ACTAATC

ACTAAT

ACTAATC

ACTAAT

ACTAATC

ACTAAT

ACTAATC

ACTAAT

ACTAATC

ACTAAT

Extension rules

- Rule 1: The suffix ends at a leaf, you add a character on the edge entering the leaf
- Rule 2: The suffix ended internally and the extended suffix does not exist, you add a leaf and possibly an internal node
- Rule 3: The suffix exists and the extended suffix exists, you do nothing

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

Skip forward..

ACTAATCACT
A
C
T
ATCAC
CTAATCAC
TCAC
TAATCAC
AC
AATCAC
CAC
4
1
5
2
7
3
6

ACTAATCACT
A
C
T
ATCAC
CTAATCACT
TCAC
TAATCAC
AC
AATCAC
CAC
4
1
5
2
7
3
6

ACTAATCACT
A
C
T
ATCAC
CTAATCACT
TCAC
TAATCACT
AC
AATCAC
CAC
4
1
5
2
7
3
6

ACTAATCACT
A
C
T
ATCAC
CTAATCACT
TCAC
TAATCACT
AC
AATCACT
CAC
4
1
5
2
7
3
6

ACTAATCACT
A
C
T
ATCACT
CTAATCACT
TCAC
TAATCACT
AC
AATCACT
CAC
4
1
5
2
7
3
6

ACTAATCACT
A
C
T
ATCACT
CTAATCACT
TCACT
TAATCACT
AC
AATCACT
CAC
4
1
5
2
7
3
6

ACTAATCACT
A
C
T
ATCACT
CTAATCACT
TCACT
TAATCACT
AC
AATCACT
CACT
4
1
5
2
7
3
6

ACTAATCACT
A
C
T
ATCACT
CTAATCACT
TCACT
TAATCACT
ACT
AATCACT
CACT
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACT
CTAATCACT
TCACT
TAATCACT
ACT
AATCACT
CACT
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACT
CTAATCACTG
TCACT
TAATCACT
ACT
AATCACT
CACT
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACT
CTAATCACTG
TCACT
TAATCACTG
ACT
AATCACT
CACT
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACT
CTAATCACTG
TCACT
TAATCACTG
ACT
AATCACTG
CACT
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACTG
CTAATCACTG
TCACT
TAATCACTG
ACT
AATCACTG
CACT
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACTG
CTAATCACTG
TCACTG
TAATCACTG
ACT
AATCACTG
CACT
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACTG
CTAATCACTG
TCACTG
TAATCACTG
ACT
AATCACTG
CACTG
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACTG
CTAATCACTG
TCACTG
TAATCACTG
ACTG
AATCACTG
CACTG
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACTG
CT
TCACTG
TAATCACTG
ACTG
AATCACTG
CACTG
4
5
2
7
3
6
G
AATCACTG
1
8

ACTAATCACTG
A
C
T
ATCACTG
CT
TCACTG
T
ACTG
AATCACTG
CACTG
4
5
7
3
6
G
AATCACTG
AATCACTG
G
1
8
2
9

ACTAATCACTG
A
C
T
G
11
ATCACTG
CT
TCACTG
T
ACTG
AATCACTG
CACTG
G
4
5
7
3
6
10
G
AATCACTG
G
AATCACTG
1
8
2
9

Observations

At the first extension we must end at a leaf because no longer suffix exists (rule 1)

At the second extension we still most likely to end at a leaf.

We will not end at a leaf only if the second suffix is a prefix of the first

Say at some extension we do not end at a leaf

Then this suffix is a prefix of some other suffix (suffixes)

We will not end at a leaf in subsequent extensions

Is there a way to continue using i^{th} character ?

(Is it a prefix of a suffix where the next character is the ith character ?)

Rule 3

Rule 2

Rule 3

Rule 2

If we apply rule 3 then in all subsequent extensions we will apply rule 3

Otherwise we keep applying rule 2 until in some subsequent extensions we will apply rule 3

Rule 3

In terms of the rules that we apply a phase looks like:

1 1 1 1 1 1 1 2 2 2 2 3 3 3 3

We have nothing to do when applying rule 3, so once rule 3 happens we can stop

We don't really do anything significant when we apply rule 1 (the structure of the tree does not change)

Representation

- We do not really store a substring with each edge, but rather pointers into the starting position and ending position of the substring in the text

- With this representaion we do not really have to do anything when rule 1 applies

How do phases relate to each other

1 1 1 1 1 1 1 2 2 2 2 3 3 3 3

The next phase we must have:

1 1 1 1 1 1 1 1 1 1 1 2/3

So we start the phase with the extension that was the first where we applied rule 3 in the previous phase

Suffix Links

ACTAATCACTG
A
C
T
G
11
ATCACTG
CT
TCACTG
T
ACTG
AATCACTG
CACTG
G
4
5
7
3
6
10
G
AATCACTG
AATCACTG
G
1
8
2
9

ACTAATCACTG
A
C
T
G
11
ATCACTG
CT
TCACTG
T
ACTG
AATCACTG
CACTG
G
4
5
7
3
6
10
G
AATCACTG
AATCACTG
G
1
8
2
9

ACTAATCACTG
A
C
T
G
11
ATCACTG
CT
TCACTG
T
ACTG
AATCACTG
CACTG
G
4
5
7
3
6
10
G
AATCACTG
AATCACTG
G
1
8
2
9

ACTAATCACTG
A
C
T
G
11
ATCACTG
CT
TCACTG
T
ACTG
AATCACTG
CACTG
G
4
5
7
3
6
10
G
AATCACTG
AATCACTG
G
1
8
2
9

Suffix Links

- From an internal node that corresponds to the string $a\beta$ to the internal node that corresponds to β (if there is such node)

- Is there such a node ?

Suppose we create v applying rule 2. Then there was a suffix aβx… and now we add aβy

So there was a suffix βx…

- Is there such a node ?

Suppose we create v applying rule 2. Then there was a suffix aβx… and now we add aβy

So there was a suffix βx…

If there was also a suffix βz…

Then a node corresponding to β is there

- Is there such a node ?

Suppose we create v applying rule 2. Then there was a suffix aβx… and now we add aβy

So there was a suffix βx…

If there was also a suffix βz…

Then a node corresponding to β is there

Otherwise it will be created in the next extension when we add βy

Inv: All suffix links are there except (possibly) of the last internal node added

You are at the (internal) node corresponding to the last extension

Remember: we apply rule 2

You start a phase at the last internal node of the first extension in which you applied rule 3 in the previous iteration

1) Go up one node (if needed) to find a suffix link

2) Traverse the suffix link

3) If you went up in step 1 along an edge that was labeled δ then go down consuming a string δ

Create the new internal node if necessary

Create the new internal node if necessary

Create the new internal node if necessary, add the suffix

Create the new internal node if necessary, add the suffix and install a suffix link if necessary

Analysis

Handling all extensions of rule 1 and all extensions of rule 3 per phase take O(1) time ➔ O(n) total

How many times do we carry out rule 2 in all phases ?

O(n)

Does each application of rule 2 takes constant time ?

No ! (going up and traversing the suffix link takes constant time, but then we go down possibly on many edges..)

β
aβ
δ
δ
y
v
x..
y

So why is it a linear time algorithm ?

How much can the depth change when we traverse a suffix link ?

It can decrease by at most 1

Punch line

Each time we go up or traverse a suffix link the depth decreases by at most 1

When starting the depth is 0, final depth is at most n

So during all applications of rule 2 together we cannot go down more than 3n times

THM: The running time of Ukkonen's algorithm is O(n)

Suffix tree Applications

Lowest common ancestors (LCA)

A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it

Why LCA?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes

Generalized Suffix Trees (GST)

A GST is simply a ST for a set of strings, each one ending with a different marker. The leafs have two numbers, one identifiing the string and the other identifiing the position inside the string.

S_1 = **abbc$**

S_2 = **babc#**

Exercise: generalized suffix tree

1. Draw the GST for s1=abab, s2=aab

GST Solution

The generalized suffix tree of s_1=abab and s_2=aab is:

{

$	#
b$	b#
ab$	ab#
bab$	aab#
abab$	

}

ST applications

- Searching for substrings
- LCS
- Hamming distance
- Longest palindrome

Longest common substring

Let S_1 and S_2 be two string over the same alphabet. The **<u>Longest Common Substring</u>** problem is to find the longest substring of S_1 that is also a substring of S_2.

Knuth in 1970 conjectured that this problem was $\Theta(n^2)$

Building a generalized suffix tree for S_1 and S_2, to solve the problem one has to identify the nodes which belong to both suffix trees of S_1 and S_2 and choose the one with greatest **string depth** (length of the path label from the root to itself). All these operations cost $O(n)$.

Longest Common Extension

LCS(T,P) is solved in linearly using suffix trees.

Find the longest substring of T[i..] that matches a substring of P[j..], for all (i, j).

It can be solved in O(n+m) time. Build a generalized suffix tree for T and P.

For every leaf i of T and j of P, find their LCA (lowest common ancestor) in the tree (it can be done in constant time after preprocessing the tree).

Exercise: Find LCS of s1 and s2

using GST of s_1=abab and s_2=aab is:

{

$	#
b$	b#
ab$	ab#
bab$	aab#
abab$	

}

LCS solution

Every node with a leaf descendant from string s_1 and a leaf descendant from string s_2 represents a maximal common substring and vice versa.

Find such node with largest "string depth".

GST(s1=abab, s2=aab)

Finding maximal palindromes

Palindromes e.g: Dad, Radar, Malayalam

Q. To find all maximal palindromes in a string s

Let s = cba|aba

The maximal palindrome with center between i-1 and i is the LCP of the suffix at position i of s and the suffix at position m-i+1 of reverse(s).

Maximal palindromes algorithm

Algorithm:

```
• r = reverse(s).
• m = len(s);
• build GST(s,r)
• for i=1..m
•   find LCA(s[i..],r[m-i+1…])
```

Complexity: O(n) time to identify all palindromes

Exercise

Find the palindrome in cbaaba

1. Build GST(s=cbaaba$, r=abaabc#)
2. Find the LCP

Palindrome example

Let s = cbaaba$ then reverse(s) = abaabc#

Hamming and Edit Distances

Hamming Distance: two strings of the same length are aligned and the distance is the number of mismatches between them.

abbcdbaabbc

abbdcbbbaac

$H = 6$

Edit Distance: it is the minimum number of insertions, deletions and substitutions needed to trasform a string into another.

abbcdbaabbc

cbcdbaabc

abbcdbaabbc

abbcdbaabbc

$E = 3$

The k-mismatches problem

We have a text T and a pattern P, and we want to find occurences of P in T, allowing a maximum of k mismatches, i.e. we want to find all the substring T' of T such that $H(P,T') \leq k$.

We can use suffix trees, but they do not perfom well anymore: the algorithm scans all the paths to leafs, keeping track of errors, and abandons the path if this number becomes greater that k.

The algorithm is made faster by using the LCS. For every suffix of T, the pieces of agreement between the suffix and P are matched together until P is exausted or the errors overcome k. Every piece is found in costant time. The complexity of the resulting algorithm is $O(k|T|)$.

aaacaabaaaaa….

aabaab

An occurence is found in position 2 of T, with one error.

Inexact Matching

In biology, inexact matching is very important:

• Similarity in DNA sequences implies often that they have the same biological function (viceversa is not true);

• Mutations and error transcription make exact comparison not very useful.

There are a lot of algorithms that deal with inexact matching (with respect to edit distance), and they are mainly based on dynamic programming or on automata. Suffix trees are used as a secondary tools in some of them, because their structure is inadapt to deal with insertions and deletions, and even with substitutions.

The main efforts are spend in speeding up the average behaviour of algorithms, and this is justified because of the fact that random sequences often fall in these cases (and DNA sequences have an high degree of randomness).

Suffix Arrays

Suffix array

- We loose some of the functionality but we save space.

Let s = abab

Sort the suffixes lexicographically:
ab, abab, b, bab

The suffix array gives the indices of the suffixes in sorted order

3	1	4	2

How do we build it ?

- Build a suffix tree
- Traverse the tree in DFS, lexicographically picking edges outgoing from each node and fill the suffix array.
- O(n) time

How do we search for a pattern ?

- If P occurs in T then all its occurrences are consecutive in the suffix array.
- Do a binary search on the suffix array
- Takes O(m logn) time

Example

Let S = mississippi

Let P = issa

L →	11	i
	8	ippi
	5	issippi
	2	ississippi
	1	mississippi
M →	10	pi
	9	ppi
	7	sippi
	4	sisippi
	6	ssippi
R →	3	ssissippi

How do we accelerate the search ?

Maintain ℓ = LCP(P,L)

Maintain r = LCP(P,R)

If ℓ = r then start comparing M to P at ℓ + 1

How do we accelerate the search ?

If $\ell > r$ then

Suppose we know LCP(L,M)

If LCP(L,M) < ℓ we go left

If LCP(L,M) > ℓ we go right

If LCP(L,M) = ℓ we start comparing at $\ell + 1$

Analysis of the acceleration

If we do more than a single comparison in an iteration then $\max(\ell, r)$ grows by 1 for each comparison: O(logn + m) time

Finite Automata

- A *finite automaton* M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where
  - Q is a finite set of *states*
  - q_0 ε Q is the *start state*
  - $A \subseteq Q$ is a set of *accepting states*
  - Σ is a finite *input alphabet*
  - δ is the *transition function* that gives the next state for a given current state and input

How a Finite Automaton Works

- The finite automaton M begins in state q_0
- Reads characters from $\sum$ one at a time
- If M is in state q and reads input character a, M moves to state $\delta(q,a)$
- If its current state q is in A, M is said to have *accepted* the string read so far
- An input string that is not accepted is said to be *rejected*

Example

- $Q = \{0,1\}$, $q_0 = 0$, $A=\{1\}$, $\Sigma = \{a, b\}$
- $\delta(q,a)$ shown in the transition table/diagram
- This accepts strings that end in an odd number of a's; e.g., abbaaa is accepted, aa is rejected

state	input a	input b
0	1	0
1	0	0

transition table

transition diagram

Finite automata string matcher

- Construct deterministic FA for the pattern P before starting match with T.
- Processing time takes $\Theta(n)$.
- T matches P, if DFA(P) accepts T

String-Matching Automata

- Given the pattern $P[1..m]$, build a finite automaton M
  - The state set is $Q=\{0, 1, 2, \ldots, m\}$
  - The start state is 0
  - The only accepting state is m

- Time to build M can be large if Σ is large

String-Matching Automata …contd

- Scan the text string T [1..n] to find all occurrences of the pattern P [1..m]

- String matching is efficient: $\Theta(n)$
  - Each character is examined exactly once
  - Constant time for each character
- But …time to compute δ is $O(m\ |\Sigma|)$
  - δ Has $O(m\ |\Sigma|\ )$ entries

DFA matcher

Input: Text string T [1..n], δ and m

Result: All valid shifts displayed

```
FINITE-AUTOMATON-MATCHER (T, m, δ)
    n ← length[T]
    q ← 0
    for i ← 1 to n
        q ← δ (q, T [i])
        if q = m
                print "pattern occurs with shift" i-m
```

Example

state	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
$\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

The KMP Algorithm

- The Knuth-Morris-Pratt (KMP) algorithm looks for the pattern in the text in a *left-to-right* order (like the brute force algorithm).
- It differs from the brute-force algorithm by keeping track of information gained from previous comparisons.
- But it shifts the pattern more intelligently than the brute force algorithm.

Knuth-Morris-Pratt (KMP) Method

- Avoids computing δ (transition function) of DFA matcher
- Instead computes a *prefix function* π in $O(m)$ time. π has only m entries
- Prefix function stores info about how the pattern matches against shifts of itself

(to avoid testing shifts that will never match).

How much to skip?

- If a mismatch occurs between the text and pattern P at P[j], what is the *most* we can shift the pattern to avoid wasteful comparisons?

- *Answer*: the largest prefix of P[0 .. j-1] that is a suffix of P[1 .. j-1]

KMP failure function

- A failure function (f) is computed that indicates how much of the last comparison can be reused if it fails.

- f is defined to be the longest prefix of the pattern P[0,..,j] that is also a suffix of P[1,..,j]

  Note: **not** a suffix of P[0,..,j]

KMP Advantages

- KMP runs in optimal time: O(m+n)
  - fast

- The algorithm never needs to move backwards in the input text, T
  - this makes the algorithm good for processing very large files that are read in from external devices or through a network stream

KMP Disadvantages

- KMP doesn’t work so well as the size of the alphabet increases
  - more chance of a mismatch (more possible mismatches)
  - mismatches tend to occur early in the pattern, but KMP is faster when the mismatches occur later

KMP Extensions

- The basic algorithm doesn't take into account the letter in the text that caused the mismatch.

Basic KMP does **not** do this.

KMP failure function

- KMP failure function, for P = ababac

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

- This shows how much of the beginning of the string matches up to the portion immediately preceding a failed comparison.

  if the comparison fails at (4), we know the a,b in positions 2,3 is identical to positions 0,1

Knuth-Morris-Pratt Algorithm

- Achieves $\Theta(n + m)$ by avoiding precomputation of δ.
- Instead, we precompute $\pi[1..m]$ in $O(m)$ time.
- As T is scanned, $\pi[1..m]$ is used to deduce information given by δ in FA algorithm.

The KMP Algorithm (contd.)

- Time Complexity Analysis
- define k = i - j
- In every iteration through the while loop, one of three things happens.

  1) if T[i] = P[j], then i increases by 1, as does j k remains the same.

  2) if T[i] != P[j] and j > 0, then i does not change and k increases by at least 1, since k changes from i - j to i - f(j-1)

  3) if T[i] != P[j] and j = 0, then i increases by 1 and k increases by 1 since j remains the same.

The KMP Algorithm (contd.)

- Thus, each time through the loop, either i or k increases by at least 1, so the greatest possible number of loops is 2n
- This of course assumes that f has already been computed.
- However, f is computed in much the same manner as KMPMatch so the time complexity argument is analogous. KMPFailureFunction is O(m)
- Total Time Complexity: O(n + m)

KMP Matcher

Algorithm KMPMatch(T,P)

Input: Strings T (text) with n characters and P (pattern) with m characters.

Output: Starting index of the first substring of T matching P, or an indication that P is not a substring of T.

Algorithm

```
f ← KMPFailureFunction(P) {build failure function}
    i ← 0
    j ← 0
    while i < n do
        if P[j] = T[i] then
            if j = m - 1 then
                return i - m - 1 {a match}
            i ← i + 1
            j ← j + 1
        else if j > 0 then {no match, but we have advanced}
            j ← f(j-1) {j indexes just after matching prefix in
    P}
        else
            i ← i + 1
    return "There is no substring of T matching P"
```

Create failure function

```
f ← KMPFailureFunction(P) {build failure function}
    i ← 0,  j ← 0
    while i ≤ m-1 do
                if P[j] = T[i] then
                        if j = m - 1 then
                           { we have matched j+1 characters}
                        f(i) ← j + 1
                        i ← i + 1
                        j ← j + 1
                else if j > 0 then
                   j ← f(j-1) {j indexes just after matching prefix in
P}
                else {there is no match}
                        f(i) ← 0
                        i ← i + 1
```

The KMP Algorithm (contd.)

- A graphical representation of the KMP string searching algorithm

Motivating Example

Shift s is discovered to be invalid because of mismatch of 6th character of P.

By definition of P, we also know s + 1 is an invalid shift

However, s + 2 may be a valid shift.

Motivating Example

The shift s + 2. Note that the first 3 characters of T starting at s + 2 don't have to be checked again -- we already know what they are.

Motivating Example

The longest prefix of P that is also a proper suffix of P_5 is P_3. We will define $\pi[5] = 3$.

In general, if q characters have matched successfully at shift s, the next potentially valid shift is $s' = s + (q - \pi[q])$.

The Prefix Function

π is called the **prefix function** for P.

π: $\{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m-1\}$

$\pi[q]$ = length of the longest prefix of P that is a proper suffix of P_q, i.e.,

$\pi[q] = \max\{k: k < q \text{ and } P_k \underline{\text{suf}}\ P_q\}$.

```
Compute-π(P)
    m := length[P];
    π[1] := 0;
    k := 0;
    for q := 2 to m do
        while k > 0 and P[k+1] ≠ P[q] do
            k := π[k]
        od;
        if P[k+1] = P[q] then
            k := k + 1
        fi;
        π[q] := k
    od;
    return π
```

Example

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
π[i]	0	0	1	2	3	0	1

Same as our FA example

P_7 = a b a b a c a
|
a = P_1

P_6 = a b a b a c
|
$\varepsilon = P_0$

P_5 = a b a b a
| | |
a b a = P_3

P_4 = a b a b
| |
a b = P_2

P_3 = a b a
|
a = P_1

P_2 = a b
|
$\varepsilon = P_0$

P_1 = a
|
$\varepsilon = P_0$

Another Explanation …

Essentially KMP is computing a FA with epsilon moves. The "spine" of the FA is implicit and doesn't have to be computed -- it's just the pattern P. π gives the ε transitions. There are O(m) such transitions.

Recall from Comp 455 that a FA with epsilon moves is conceptually able to be in several states at the same time ("in parallel"). That's what's happening here -- we're exploring pieces of the pattern "in parallel".

Another Example

i	1	2	3	4	5	6	7	8	9	10
P[i]	a	b	b	a	b	a	a	b	b	a
π[i]	0	0	0	1	2	1	1	2	3	4

P_{10} = a b b a b a a b b a
| | | |
a b b a = P_4

P_7 = a b b a b a a
|
a = P_1

P_4 = a b b a
|
a = P_1

P_1 = a
|
$\varepsilon = P_0$

P_9 = a b b a b a a b b
| | |
a b b = P_3

P_6 = a b b a b a
|
a = P_1

P_3 = a b b
|
$\varepsilon = P_0$

P_8 = a b b a b a a b
| |
a b = P_2

P_5 = a b b a b
| |
a b = P_2

P_2 = a b
|
$\varepsilon = P_0$

Time Complexity

Amortized Analysis --

Φ_0
$\downarrow$ loop q = 2 (1st iteration)
Φ_1
$\downarrow$ loop q = 3 (2nd iteration)
Φ_2
$\downarrow$ loop q = 4 (3rd iteration)
$\vdots$
$\downarrow$ loop q = m ((m – 1)st iteration)
Φ_{m-1}

Φ = potential function = value of k

Amortized cost: $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$

iteration actual loop cost

Time Complexity (Continued)

Total amortized cost: $\sum_{i=1}^{m-1} \hat{c}_i$

$$= \sum_{i=1}^{m-1} (c_i + \Phi_i - \Phi_{i-1})$$

$$= \sum_{i=1}^{m-1} c_i + \Phi_{m-1} - \Phi_0$$

If $\Phi_{m-1} \geq \Phi_0$, then amortized cost upper bounds real cost.

We have $\Phi_0 = 0$ (initial value of k)

$\Phi_{m-1} \geq 0$ (final value of k).

We show $\hat{c}_i = O(1)$.

Time Complexity (Continued)

The value of $\hat{c}_i$ obviously depends on how many times statement 6 is executed.

Note that $k > \pi[k]$. Thus, each execution of statement 6 decreases k by at least 1.

So, suppose that statements 5..6 iterate several times, decreasing the value of k.

We have: number of iterations $\leq k_{old} - k_{new}$. Thus,

$$\hat{c}_i \leq \underbrace{O(1)}_{\text{for statements other than 5 \& 6}} + 2(k_{old} - k_{new}) + \underbrace{\Phi_i}_{= k_{new}} - \underbrace{\Phi_{i-1}}_{= k_{old}}$$

Hence, $\hat{c}_i = O(1)$. Total cost is therefore $O(m)$.

Rest of the Algorithm

```
KMP(T, P)
    n := length[T];
    m := length[P];
    π := Compute-π(P);
    q := 0;
    for i := 1 to n do
        while q > 0 and P[q+1] ≠ T[i] do
            q := π[q]
        od;
        if P[q+1] = T[i] then
            q := q + 1
        fi;
        if q = m then
            print "pattern occurs with shift i – m";
            q := π[q]
        fi
    od
```

Time complexity of loop is O(n) (similar to the analysis of Compute-π).

Total time is O(m + n).

Example

P = a b a b c

i	1	2	3	4	5
P[i]	a	b	a	b	c
π[i]	0	0	1	2	0

	1	2	3	4	5	6	7	8	9	10
T =	a	b	b	a	b	a	b	a	b	c

Start of 1st loop: q = 0, i = 1 **[a]**
2nd loop: q = 1, i = 2 **[b]**
3rd loop: q = 2, i = 3 **[b]** **mismatch detected**
4th loop: q = 0, i = 4 **[a]**
5th loop: q = 1, i = 5 **[b]**
6th loop: q = 2, i = 6 **[a]**
7th loop: q = 3, i = 7 **[b]**
8th loop: q = 4, i = 8 **[a]** **mismatch detected**
9th loop: q = 3, i = 9 **[b]**
10th loop: q = 4, i = 10 **[c]** **match detected**
Termination: q = 5

Please see the book for formal correctness proofs.
(They're *very* tedious.)

Terminology/Notations

- String *w* is a *prefix* of string *x*, if *x*=*wy* for some string *y* (e.g., “srilan” of “srilanka”)
- String *w* is a *suffix* of string *x*, if *x*=*yw* for some string *y* (e.g., “anka” of “srilanka”)
- The *k*-character prefix of the pattern *P* [1..*m*] denoted by P_k
  - E.g., $P_0 = \varepsilon$, $P_m = P = P[1..m]$

Prefix Function for a Pattern

- Given that pattern prefix $P[1..q]$ matches text characters $T[(s+1)..(s+q)]$, what is the least shift $s' > s$ such that

  $P[1..k] = T[(s'+1)..(s'+k)]$ where $s'+k=s+q$?

- At the new shift s', no need to compare the first k characters of P with corresponding characters of T
  - Since we know that they match

Prefix Function: Example 1

Compare pattern against itself; longest prefix of P that is also a suffix of P_5 is P_3; so $\pi[5]= 3$

Prefix Function: Example 2

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	**a**	**b**	**a**	**b**	**a**	**b**	**a**	**b**	**c**	**a**
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

$\pi[q] = \max \{ k \mid k<q \text{ and } P_k \text{ is a suffix of } P_q \}$

Illustration: given a String ‘S’ and pattern ‘p’ as follows:

S	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

p	a	b	a	b	a	c	a

Let us execute the KMP algorithm to find whether ‘p’ occurs in ‘S’.

For ‘p’ the prefix function, Π was computed previously and is as follows:

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	1

Initially: n = size of S = 15;
m = size of p = 7

Step 1: i = 1, q = 0
comparing p[1] with S[1]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

P[1] does not match with S[1]. 'p' will be shifted one position to the right.

Step 2: i = 2, q = 0
comparing p[1] with S[2]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

P[1] matches S[2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

Comparing p[2] with S[3] p[2] does not match with S[3]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Backtracking on p, comparing p[1] and S[3]

Step 4: i = 4, q = 0

comparing p[1] with S[4] p[1] does not match with S[4]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 5: i = 5, q = 0

comparing p[1] with S[5] p[1] matches with S[5]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 6: i = 6, q = 1

Comparing p[2] with S[6] p[2] matches with S[6]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p	a	b	a	b	a	c	a								

Step 7: i = 7, q = 2

Comparing p[3] with S[7] p[3] matches with S[7]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p	a	b	a	b	a	c	a								

Step 8: i = 8, q = 3

Comparing p[4] with S[8] p[4] matches with S[8]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p	a	b	a	b	a	c	a								

Step 9: i = 9, q = 4

Comparing p[5] with S[9] p[5] matches with S[9]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p	a	b	a	b	a	c	a								

Step 10: i = 10, q = 5

Comparing p[6] with S[10] p[6] doesn't match with S[10]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p	a	b	a	b	a	c	a								

Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

Step 11: i = 11, q = 4

Comparing p[5] with S[11] p[5] matches with S[11]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p	a	b	a	b	a	c	a								

Step 12: i = 12, q = 5

Step 13: i = 13, q = 6

Pattern ‘p’ has been found to completely occur in string ‘S’. The total number of shifts that took place for the match to be found are: i – m = 13 – 7 = 6 shifts.

KMP Example

i
T:
. . a b a a b x . . . . .
P:
a b a a b a
j = 5
j
a b a a b a
j_{new} = 2
No need to repeat these comparisons
Resume comparing here

Why

j == 5

- Find largest prefix (start) of:

  "a b a a b" (P[0..j-1])

  which is suffix (end) of:

  "b a a b" (p[1 .. j-1])

- Answer: "a b"
- Set j = 2 // the new j value

KMP Failure Function

- KMP preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself.
- j = mismatch position in P[]
- k = position before the mismatch (k = j-1).
- The *failure function* F(k) is defined as the *size* of the largest prefix of P[0..k] that is also a suffix of P[1..k].

Failure Function Example

(k == j-1)

- P: "abaaba"

  j: 012345

	0	1	2	3	4
	0	0	1	1	2

F(k) is the size of
the largest prefix.

- In code, F() is represented by an array, like the table.

Why is F(4) == 2?

P: "abaaba"

- F(4) means
  - find the size of the largest prefix of P[0..4] that is also a suffix of P[1..4]

  = find the size largest prefix of "abaab" that is also a suffix of "baab"

  = find the size of "ab"

  = 2

Using the Failure Function

- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm.
  - if a mismatch occurs at P[j]
    (i.e. P[j] != T[i]), then
    k = j-1;
    j = F(k); // obtain the new j

Example

T: a b a c a a b a c c a b a c a b a a b b

P: a b a c a b (1 2 3 4 5 6)

a b a c a b (7)

a b a c a b (8 9 10 11 12)

a b a c a b (13)

a b a c a b (14 15 16 17 18 19)

k	0	1	2	3	4
$F(k)$	0	0	1	0	1

Why is F(4) == 1?

P: "abacab"

- F(4) means
  - find the size of the largest prefix of P[0..4] that is also a suffix of P[1..4]

  = find the size largest prefix of "abaca" that is also a suffix of "baca"

  = find the size of "a"

  = 1

Boyer and Moore Strin matching Algorithm

A fast string searching algorithm. *Communications of the ACM*. Vol. 20 p.p. 762-772, 1977.

BOYER, R.S. and **MOORE, J.S.**

Boyer and Moore Algorithm

- The algorithm compares the pattern *P* with the substring of sequence *T* within a sliding window in the **right-to-left order**.
- The **bad character rule** and **good suffix rule** are used to determine the movement of sliding window.

- Very fast, used in Python library.

Bad Character Rule

Suppose that P_1 is aligned to T_s now, and we perform a pair-wise comparing between text T and pattern P from right to left. Assume that the first mismatch occurs when comparing T_{s+j-1} with P_j .

Since $T_{s+j-1} \neq P_j$, we move the pattern P to the right such that the largest position c in the left of P_j is equal to T_{s+j-1}. We can shift the pattern at least (j-c) positions right.

Rule 2-1: Character Matching Rule (A Special Version of Rule 2)

- Bad character rule uses Rule 2-1 (Character Matching Rule).
- For any character *x* in *T*, find the nearest *x* in *P* which is to the left of *x* in *T*.

Implication of Rule 2-1

- Case 1. If there is a *x* in *P* to the left of *T*, move *P* so that the two x's match.

- Case 2: If no such a x exists in P, consider the partial window defined by x in T and the string to the left of it.

- Ex: Suppose that P_1 is aligned to T_6 now. We compare pair-wise between T and P from right to left. Since $T_{16,17} = P_{11,12}$ = "CA" and T_{15} ="G" $\neq P_{10}$ = "T". Therefore, we find the rightmost position c=7 in the left of P_{10} in P such that P_c is equal to "G" and we can move the window at least (10-7=3) positions.

Good Suffix Rule 1

- If a mismatch occurs in T_{s+j-1}, we match T_{s+j-1} with $P_{j'-m+j}$, where j' ($m-j+1 \leqq j' < m$) is the **largest position** such that

  (1) $\boldsymbol{P_{j+1,m}}$ **is a suffix of** $\boldsymbol{P_{1,j'}}$

  (2) $\boldsymbol{P_{j'-(m-j)} \neq P_{j'}}$.

- We can move the window at least ($m-j'$) position(s).

Rule 2: The Substring Matching Rule

- For any substring *u* in T, find a nearest *u* in P which is to the left of it. If such a *u* in P exists, move P; otherwise, we may define a new partial window.

- Ex: Suppose that P_1 is aligned to T_6 now. We compare pair-wise between P and T from right to left. Since $T_{16,17}$ = "CA" = $P_{11,12}$ and T_{15} ="A" $\neq P_{10}$ = "T". We find the substring "CA" in the left of P_{10} in P such that "CA" is the suffix of $P_{1,6}$ and the left character to this substring "CA" in P is not equal to P_{10} = "T". Therefore, we can move the window at least *m-j'* (12-6=6) positions right.

Good Suffix Rule 2

Good Suffix Rule 2 is used only when Good Suffix Rule 1 can not be used. That is, t does not appear in P(1, j). Thus, t is **unique** in P.

- If a mismatch occurs in T_{s+j-1}, we match $T_{s+m-j'}$ with P_1, where j' ($1 \leqq j' \leqq m-j$) is **the largest position** such that

  $P_{1,j'}$ is a suffix of $P_{j+1,m}$.

P.S. : t' is suffix of substring t.

Rule 3-1: Unique Substring Rule

- The substring u appears in P exactly once.
- If the substring u matches with $T_{i,j}$, no matter whether a mismatch occurs in some position of P or not, we can slide the window by l.

The string s is the longest prefix of P which equals to a suffix of u.

Rule 1: The Suffix to Prefix Rule

- For a window to have any chance to match a pattern, in some way, there must be a suffix of the window which is equal to a prefix of the pattern.

- Note that the above rule also uses Rule 1.
- It should also be noted that the unique substring is the shorter and the more right-sided the better.
- A short *u* guarantees a short (or even empty) *s* which is desirable.

- Ex: Suppose that P_1 is aligned to T_6 now. We compare pair-wise between P and T from right to left. Since $T_{12} \neq P_7$ and there is no substring $P_{8,12}$ in left of P_8 to exactly match $T_{13,17}$. We find a longest suffix "AATC" of substring $T_{13,17}$, the longest suffix is also prefix of P. We shift the window such that the last character of prefix substring to match the last character of the suffix substring. Therefore, we can shift at least 12-4=8 positions.

- Let ***Bc*(*a*)** be the rightmost position of *a* in *P*. The function will be used for applying ***bad character rule***.

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A

Σ	A	C	G	T
B	12	11	0	10

- We can move our pattern right at least $\boldsymbol{j\text{-}B(T_{s+j\text{-}1})}$ position by above ***Bc*** function.

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T	A	G	C	T	A	G	C	C	T	G	C	A	C	G	T	A	C	A

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A

Move at least
10-*B*(*G*) = 10 positions

Let ***Gs*(*j*)** be **the largest number of shifts** by ***good suffix rule*** when a mismatch occurs for comparing P_j with some character in T.

- $gs_1(j)$ be the largest k such that $P_{j+1,m}$ **is a suffix of** $P_{1,k}$ **and** $P_{k-m+j} \neq P_j$, where $m-j+1 \leqq k < m$; 0 if there is no such k.

  (gs_1 is for Good Suffix Rule 1)

- $gs_2(j)$ be the largest k such that $P_{1,k}$ **is a suffix of** $P_{j+1,m}$, where $1 \leqq k \leqq m-j$; 0 if there is no such k.

  (gs_2 is for Good Suffix Rule 2.)

- $Gs(j) = m - \max\{gs_1, gs_2\}$, if $j = m$,$Gs(j)=1$.

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
gs_1	0	0	0	0	0	0	9	0	0	6	1	0
gs_2	4	4	4	4	4	4	4	4	1	1	1	0
Gs	8	8	8	8	8	8	3	8	11	6	11	1

$gs_1(7)=9$

$\because P_{8,12}$ is a suffix of $P_{1,9}$

and $P_4 \neq P_7$

$gs_2(7)=4$

$\because P_{1,4}$ is a suffix of $P_{8,12}$

How do we obtain $\boldsymbol{gs_1}$ and $\boldsymbol{gs_2}$?

In the following, we shall show that by constructing the **Suffix Function**, we can kill two birds with one arrow.

Suffix function *f'*

- For $1 \leqq j \leqq m-1$, let the suffix function $f'(j)$ for $\boldsymbol{P_j}$ be the **smallest** $\boldsymbol{k}$ such that $\boldsymbol{P_{k,m} = P_{j+1,m-k+j+1}}$; ($j+2 \leqq k \leqq m$)
  - If there is no such k, we set $f' = m+1$.
  - If $j=m$, we set $f'(m)=m+2$.

- Ex:

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14

- $f'(4)=8$, it means that $P_{f'(4),m} = P_{8,12} = P_{5,9} = P_{4+1,4+1+m-f'(4)}$
- Since there is no k for $13= j+2 \leqq k \leqq 12$, we set $f'(11)=13$.

Suppose that the Suffix is obtained. How can we use it to obtain gs_1 and gs_2?

gs_1 can be obtained by scanning the Suffix function from right to left.

BM Example

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

P	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14

Example

As for Good Suffix Rule 2, it is relatively easier.

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14

Question: How can we construct the Suffix function?

To explain this, let us go back to the prefix function used in the MP Algorithm.

The following figure illustrates the prefix function in the MP Algorithm.

The following figure illustrates the suffix function of the BM Algorithm.

We now can see that actually the suffix function is the same as the prefix. The only difference is now we consider a suffix. Thus, the recursive formula for the prefix function in MP Algorithm can be slightly modified for the suffix function in BM Algorithm.

- The formula of suffix function *f'* as follows :

Let $f'^{x}(y) = f'(f'^{x-1}(y))$ for $x > 1$ and $f'^{1}(y) = f'(y)$

$$f'(j) = \begin{cases} m+2, & \text{if } j = m \\ f'^{k}(j+1)-1, & \text{if } 1 \le j \le m\text{-}1 \text{ and there exists the smallest} \\ & k \ge 1 \text{ such that } P_{j+1} = P_{f'^{k}(j+1)\text{-}1}; \\ m+1, & \text{otherwise} \end{cases}$$

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'												14

j=m=12,
f'=m+2=14

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'											13	14

No k satisfies
$P_{j+1}=P_{f'^{k}(j+1)-1}$,
$f'=m+1=12+1=13$

$k=1 \rightarrow$

$P_{12} \neq P_{13}$

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'										13	13	14

No k satisfies
$P_{j+1}=P_{f'^{k}(j+1)-1}$,
$f'=m+1=12+1=13$

$k=1 \rightarrow$

$P_{11} \neq P_{12}$

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'									13	13	13	14

No k satisfies
$P_{j+1}=P_{f'^{k}(j+1)-1}$,
$f'=m+1=12+1=13$

$k=1 \rightarrow$

$P_{10} \neq P_{12}$

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'								12	13	13	13	14

$\because P_{j+1} = P_{f'(j+1)-1} \Rightarrow P_9 = P_{12}$,
$f' = f'(j+1) - 1 = 13 - 1 = 12$

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'							11	12	13	13	13	14

$\because P_{j+1} = P_{f'(j+1)-1} \Rightarrow P_8 = P_{11}$,
$f' = f'(j+1) - 1 = 12 - 1 = 11$

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'				8	9	10	11	12	13	13	13	14

$\because P_{j+1} = P_{f'^{1}(j+1)-1} \Rightarrow P_5 = P_8,$

$f' = f'(j+1) - 1 = 9 - 1 = 8$

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'			12	8	9	10	11	12	13	13	13	14

$\because P_{j+1} = P_{f'^{3}(j+1)-1} \Rightarrow P_4 = P_{f'^{3}(4)-1} = P_{12},$

$f' = f'^{3}(j+1) - 1 = 13 - 1 = 12$

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'		11	12	8	9	10	11	12	13	13	13	14

$\because P_{j+1} = P_{f'(j+1)-1} \Rightarrow P_3 = P_{f'(3)-1} = P_{11}$,

$f' = f'(j+1) - 1 = 12 - 1 = 11$

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14

$\because P_{j+1} = P_{f'(j+1)-1} \Rightarrow P_2 = P_{f'(2)-1} = P_{10}$,

$f' = f'(j+1) - 1 = 11 - 1 = 10$

- Let $G'(j)$, $1 \leqq j \leqq m$,to be the largest number of shifts by good suffix rules.

- First, we set $G'(j)$ to zeros as their initializations.

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14
G'	0	0	0	0	0	0	0	0	0	0	0	0

- **Step1:** We scan from right to left and $gs_1(j)$ is determined during the scanning, then $gs_1(j) >= gs_2(j)$

Observe:
If $P_j = P_4 \neq P_7 = P_{f'(j)-1}$, we know $\boldsymbol{gs_1(f'(j)-1) = m+j-f'(j)+1} = 9$.
If $t = f'(j)-1 \leqq m$ and $P_j \neq P_t$, $G'(t) = m - gs_1(f'(j)-1) = f'(j) - 1 - j$.
$\boldsymbol{f'^{(k)}(x) = f'^{(k-1)}(f'(x) - 1),\ k \geq 2}$

- When j=12, t=13. $t > m$.
- When j=11, t=12. Since P_{11}='C'≠ 'A'= P_{12},
  $G'(t) = m - \max\{gs_1(t), gs_2(t)\} = m - \underline{gs_1(t)}$
  $= f'(j) - 1 - j$
  => $G'(12)$=13-1-11= 1.

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14
G'	0	0	0	0	0	0	0	0	0	0	0	1

If $t = f'(j)-1 \leqq m$ and $P_j \neq P_t$, $G'(t)=f'(j) - 1 - j$.
$f'^{(k)}(x)=f'^{(k-1)}(f'(x) - 1)$, $k \geq 2$

- When j=10, t=12. Since P_{10}='T'≠'A' =P_{12}, $G'(12) \neq 0$.
- When j=9, t=12. P_9 = 'A' =P_{12}.
- When j=8, t=11. P_8 = 'C' =P_{11}.
- When j=7, t=10. P_7 = 'T' =P_{10}
- When j=6, t=9. P_6 = 'A' =P_9
- When j=5, t=8. P_5 = 'C' =P_8
- When j=4, t=7. Since P_4 = 'A' ≠ P_7 = 'T', $G'(7) = 8 - 1 - 4 = 3$

  Besides, $t = f'^{(2)}(4) - 1 = f'(f'(4) - 1) - 1 = 10$. Since P_4 = 'A'≠P_{10} = 'T', $G'(10) = f'(7) - 1 - j = 11 - 1 - 4 = 6$.

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14
G'	0	0	0	0	0	0	3	0	0	6	0	1

If $t = f'(j)-1 \leqq m$ and $P_j \neq P_t$, $G'(t)=f'(j) - 1 - j$.
$f'^{(k)}(x)=f'^{(k-1)}(f'(x) - 1)$, $k \geq 2$

- When j=3, t=11. P_3='C'=P_{11}.
- When j=2, t=10. P_2='T'=P_{10}
- When j=1, t=9. P_1='A'=P_9.

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14
G'	0	0	0	0	0	0	3	0	6	0	0	1

- By the above discussion, we can obtain the values using the Good Suffix Rule 1 by scanning the pattern from right to left.

- **Step2:** Continuously, we will try to obtain the values using ***Good Suffix Rule 2*** and those values are still zeros now and scan from left to right.

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14
G'	0	0	0	0	0	0	3	0	0	6	0	1

- Let k' be the **smallest** $\boldsymbol{k}$ in $\{1,\ldots,m\}$ such that $\boldsymbol{P_{f'^{(k)}(1)-1} = P_1}$ and $f'^{(k)}(1)-1<=m$.

Observe:
$\because P_{1,4}=P_{9,12}$, $\therefore gs_2(j)=m-(f'(1)-1)+1=4$, where $1 \leqq j \leqq f'^{(k')}(1)-2$.

- If $G'(j)$ is not determined in the first scan and $1<=j<= f'^{(k')}(1)-2$, thus, in the second scan, we set $G'(j)=m - \max\{gs_1(j), gs_2(j)\}= m - gs_2(j)= f'^{(k')}(1) - 2$. If no such k exists, set each undetermined value of G to m in the second scan.

- ***k=1=k'***, since $P_{f'(1)-1}=P_9=$"A"$=P_1$, we set $G'(j)=f'(1)-2$ for j=1,2,3,4,5,6,8.

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14
G'	8	8	8	8	8	8	3	8	0	6	0	1

- Let z be $f'^{(k')}(1)$-2. Let k'' be the **largest value k** such that $f''^{(k)}(z)$-1<=m.
- Then we set $G'(j) = m - gs_2(j) = m - (m - f''^{(i)}(z) - 1) = f''^{(i)}(z) - 1$, where $1<=i<=k''$ and $f''^{(i-1)}(z) < j <= f''^{(i)}(z)-1$ and $f''^{(0)}(z) = z$.

- For example, z=8 :
  - k=1, $f''^{(1)}(8)$-1=11≦m=12
  - k=2, $f''^{(2)}(8)$-1=12≦m=12 => ***k''=2***
  - i=1, $f''^{(0)}(8)$-1 = 7 < $\boldsymbol{j}$ ≦ $f''^{(1)}(8)$-1=11.
  - i=2, $f''^{(1)}(8)$-1 =11< $\boldsymbol{j}$ ≦ $f''^{(2)}(8)$-1=12.
  - We set $G(9)$ and $G(11)$= $f''^{(1)}(8)$ – 1= 12-1 = 11.

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14
G'	8	8	8	8	8	8	3	8	11	6	11	1

We essentially have to decide the maximum number of steps. We can move the window right when a mismatch occurs. This is decided by the following function:

$$\max\{G'(j), j\text{-}B(T_{s+j-1})\}$$

Example

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

mismatch

P	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

Shift →

P	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14
G'	8	8	8	8	8	8	3	8	11	6	11	1

Σ	A	C	G	T
B	12	11	0	10

We compare *T* and *P* from right to left. Since T_{12}="T"$\neq P_{12}$="A", the largest movement = max{$G'(j)$, j-$B(T_{s+j-1})$} = max{$G'(12)$, 12-$B(T_{12})$}= max{1,12-10} = 2.

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

mismatch

P	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

Shift →

P	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14
G'	8	8	8	8	8	8	3	8	11	6	11	1

Σ	A	C	G	T
B	12	11	0	10

After moving, we compare *T* and *P* from right to left. Since T_{14}="T"≠P_{12}="A", the largest movement = max{$G'(j)$, j-$B(Ts+j-1)$} = max{$G'(12)$, 12-$B(T_{14})$}
= max{1,12-10} = 2.

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

mismatch

P	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

Shift →

P	A	T	C	A	C	A	T	C	A	T	C	A
	1	2	3	4	5	6	7	8	9	10	11	12

j	1	2	3	4	5	6	7	8	9	10	11	12
P	A	T	C	A	C	A	T	C	A	T	C	A
f'	10	11	12	8	9	10	11	12	13	13	13	14
G'	8	8	8	8	8	8	3	8	11	6	11	1

Σ	A	C	G	T
B	12	11	0	10

After moving, we compare *T* and *P* from right to left. Since T_{12}="T"≠P_8="G", the largest movement = max{*G'*(*8*), *j*-*B*(T_{12})} = max{8,8-10} = 8.

Time Complexity

- The preprocessing phase in O(m+Σ) time and space complexity and searching phase in ***O***(mn) time complexity.

- The worst case time complexity for the ***Boyer-Moore*** method would be $O((n-m+1)m)$.

- It was proved that this algorithm has $O(m)$ comparisons when P is not in T. However, this algorithm has $O(mn)$ comparisons when P is in T.

The Boyer-Moore Algorithm

- The Boyer-Moore pattern matching algorithm is based on two techniques.

- 1. The *looking-glass* technique
  - find P in T by moving *backwards* through P, starting at its end

- 2. The *character-jump* technique
  - when a mismatch occurs at T[i] == x
  - the character in pattern P[j] is not the same as T[i]

- There are 3 possible cases, tried in order.

Case 1

- If P contains x somewhere, then try to *shift P* right to align the last occurrence of x in P with T[i].

Case 2

- If P contains x somewhere, but a shift right to the last occurrence is *not* possible, then
  shift P right by 1 character to T[i+1].

Case 3

- If cases 1 and 2 do not apply, then *shift* P to align P[0] with T[i+1].

No x in P

Boyer-Moore Example (1)

Last Occurrence Function

- Boyer-Moore's algorithm preprocesses the pattern P and the alphabet A to build a last occurrence function L()
  - L() maps all the letters in A to integers

- L(x) is defined as: // x is a letter in A
  - the largest index i such that P[i] == x, or
  - -1 if no such index exists

L( ) Example

- A = {a, b, c, d}
- P: "abacab"

P

a	b	a	c	a	b
0	1	2	3	4	5

x	a	b	c	d
$L(x)$	4	5	3	-1

L() stores indexes into P[]

Note

- In Boyer-Moore code, L( ) is calculated when the pattern P is read in.

- Usually L() is stored as an array
  - something like the table in the previous slide

Boyer-Moore Example (2)

x	a	b	c	d
$L(x)$	4	5	3	–1

Analysis

- Boyer-Moore worst case running time is $O(nm + A)$
- But, Boyer-Moore is fast when the alphabet (A) is large, slow when the alphabet is small.
  - e.g. good for English text, poor for binary
- Boyer-Moore is *significantly faster than brute force* for searching English text.

Worst Case Example

- T: "aaaaa…a"
- P: "baaaaa"

Suffix trees

Trie

- A tree representing a set of strings.

{
aeef
ad
bbfe
bbfg
c
}

Trie (Cont)

- Assume no string is a prefix of another

Each edge is labeled by a letter, no two edges outgoing from the same node are labeled the same.

Each string corresponds to a leaf.

Compressed Trie

- Compress unary nodes, label edges by strings

Suffix tree

Given a string s a suffix tree of s is a compressed trie of all suffixes of s

To make these suffixes prefix-free we add a special character, say $, at the end of s

Suffix tree (Example)

Let s=abab, a suffix tree of s is a compressed trie of all suffixes of s=abab$

{
$
b$
ab$
bab$
abab$
}

Trivial algorithm to build a Suffix tree

Put the largest suffix in

Put the suffix bab$ in

Put the suffix ab$ in

Put the suffix b$ in

Put the suffix $ in

We will also label each leaf with the starting point of the corres. suffix.

Analysis

Takes $O(n^2)$ time to build.

We will see how to do it in $O(n)$ time

What can we do with it ?

Exact string matching:

Given a Text T, |T| = n, preprocess it such that when a pattern P, |P|=m, arrives you can quickly decide when it occurs in T.

We may also want to find all occurrences of P in T

Exact string matching

In preprocessing we just build a suffix tree in O(n) time

Given a pattern P = ab we traverse the tree according to the pattern.

If we did not get stuck traversing the pattern then the pattern occurs in the text.

Each leaf in the subtree below the node we reach corresponds to an occurrence.

By traversing this subtree we get all k occurrences in O(n+k) time

Generalized suffix tree

Given a set of strings S a *generalized suffix tree* of S is a compressed trie of all suffixes of $s \in S$

To associate each suffix with a unique string in S add a different special char to each s

Generalized suffix tree (Example)

Let s_1=abab and s_2=aab here is a generalized suffix tree for s_1 and s_2

{

$	#
b$	b#
ab$	ab#
bab$	aab#
abab$	

}

So what can we do with it ?

Matching a pattern against a database of strings

Longest common substring (of two strings)

Every node with a leaf descendant from string s_1 and a leaf descendant from string s_2 represents a maximal common substring and vice versa.

Find such node with largest "string depth"

Lowest common ancetors

A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it

Why?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes

Finding maximal palindromes

- A palindrome: caabaac, cbaabc
- Want to find all maximal palindromes in a string s

Let s = cba|aba

The maximal palindrome with center between i-1 and i is the LCP of the suffix at position i of s and the suffix at position m-i+1 of s^r

Maximal palindromes algorithm

Prepare a generalized suffix tree for
s = cbaaba\$ and s^r = abaabc#

For every i find the LCA of suffix i of s
and suffix m-i+1 of s^r

Let s = cbaaba\$ then s^r = abaabc#

Analysis

O(n) time to identify all palindromes

Can we construct a suffix tree in linear time ?

Ukkonen's *linear* time construction

ACTAATC

A

ACTAATC

AC

ACTAATC

AC

ACTAATC

AC

ACTAATC

ACT

ACTAATC

ACT

ACTAATC

ACT

ACTAATC

ACT

ACTAATC

ACTA

ACTAATC

ACTA

ACTAATC

ACTA

ACTAATC

ACTA

ACTAATC

ACTAA

ACTAATC

ACTAA

ACTAATC

ACTAA

ACTAATC

ACTAA

ACTAATC

ACTAA

Phases & extensions

- Phase i is when we add character i

- In phase i we have i extensions of suffixes

ACTAATC

ACTAAT

ACTAATC

ACTAAT

ACTAATC

ACTAAT

ACTAATC

ACTAAT

ACTAATC

ACTAAT

ACTAATC

ACTAAT

Extension rules

- Rule 1: The suffix ends at a leaf, you add a character on the edge entering the leaf
- Rule 2: The suffix ended internally and the extended suffix does not exist, you add a leaf and possibly an internal node
- Rule 3: The suffix exists and the extended suffix exists, you do nothing

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

ACTAATC

Skip forward..

ACTAATCACT
A
C
T
ATCAC
CTAATCAC
TCAC
TAATCAC
AC
AATCAC
CAC
4
1
5
2
7
3
6

ACTAATCACT
A
C
T
ATCAC
CTAATCACT
TCAC
TAATCAC
AC
AATCAC
CAC
4
1
5
2
7
3
6

ACTAATCACT
A
C
T
ATCAC
CTAATCACT
TCAC
TAATCACT
AC
AATCAC
CAC
4
1
5
2
7
3
6

ACTAATCACT
A
C
T
ATCAC
CTAATCACT
TCAC
TAATCACT
AC
AATCACT
CAC
4
1
5
2
7
3
6

ACTAATCACT
A
C
T
ATCACT
CTAATCACT
TCAC
TAATCACT
AC
AATCACT
CAC
4
1
5
2
7
3
6

ACTAATCACT
A
C
T
ATCACT
CTAATCACT
TCACT
TAATCACT
AC
AATCACT
CAC
4
1
5
2
7
3
6

ACTAATCACT
A
C
T
ATCACT
CTAATCACT
TCACT
TAATCACT
AC
AATCACT
CACT
4
1
5
2
7
3
6

ACTAATCACT
A
C
T
ATCACT
CTAATCACT
TCACT
TAATCACT
ACT
AATCACT
CACT
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACT
CTAATCACT
TCACT
TAATCACT
ACT
AATCACT
CACT
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACT
CTAATCACTG
TCACT
TAATCACT
ACT
AATCACT
CACT
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACT
CTAATCACTG
TCACT
TAATCACTG
ACT
AATCACT
CACT
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACT
CTAATCACTG
TCACT
TAATCACTG
ACT
AATCACTG
CACT
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACTG
CTAATCACTG
TCACT
TAATCACTG
ACT
AATCACTG
CACT
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACTG
CTAATCACTG
TCACTG
TAATCACTG
ACT
AATCACTG
CACT
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACTG
CTAATCACTG
TCACTG
TAATCACTG
ACT
AATCACTG
CACTG
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACTG
CTAATCACTG
TCACTG
TAATCACTG
ACTG
AATCACTG
CACTG
4
1
5
2
7
3
6

ACTAATCACTG
A
C
T
ATCACTG
CT
TCACTG
TAATCACTG
ACTG
AATCACTG
CACTG
4
5
2
7
3
6
G
AATCACTG
1
8

ACTAATCACTG
A
C
T
ATCACTG
CT
TCACTG
T
ACTG
AATCACTG
CACTG
4
5
7
3
6
G
AATCACTG
AATCACTG
G
1
8
2
9

ACTAATCACTG
A
C
T
G
11
ATCACTG
CT
TCACTG
T
ACTG
AATCACTG
CACTG
G
4
5
7
3
6
10
G
AATCACTG
G
AATCACTG
1
8
2
9

Observations

At the first extension we must end at a leaf because no longer suffix exists (rule 1)

At the second extension we still most likely to end at a leaf.

We will not end at a leaf only if the second suffix is a prefix of the first

Say at some extension we do not end at a leaf

Then this suffix is a prefix of some other suffix (suffixes)

We will not end at a leaf in subsequent extensions

Is there a way to continue using i^{th} character ?

(Is it a prefix of a suffix where the next character is the ith character ?)

Rule 3

Rule 2

Rule 3

Rule 2

If we apply rule 3 then in all subsequent extensions we will apply rule 3

Otherwise we keep applying rule 2 until in some subsequent extensions we will apply rule 3

Rule 3

In terms of the rules that we apply a phase looks like:

1 1 1 1 1 1 1 2 2 2 2 3 3 3 3

We have nothing to do when applying rule 3, so once rule 3 happens we can stop

We don't really do anything significant when we apply rule 1 (the structure of the tree does not change)

Representation

- We do not really store a substring with each edge, but rather pointers into the starting position and ending position of the substring in the text

- With this representaion we do not really have to do anything when rule 1 applies

How do phases relate to each other

1 1 1 1 1 1 1 2 2 2 2 3 3 3 3

The next phase we must have:

1 1 1 1 1 1 1 1 1 1 1 2/3

So we start the phase with the extension that was the first where we applied rule 3 in the previous phase

Suffix Links

ACTAATCACTG
A
C
T
G
11
ATCACTG
CT
TCACTG
T
ACTG
AATCACTG
CACTG
G
4
5
7
3
6
10
G
AATCACTG
AATCACTG
G
1
8
2
9

ACTAATCACTG
A
C
T
G
11
ATCACTG
CT
TCACTG
T
ACTG
AATCACTG
CACTG
G
4
5
7
3
6
10
G
AATCACTG
AATCACTG
G
1
8
2
9

ACTAATCACTG
A
C
T
G
11
ATCACTG
CT
TCACTG
T
ACTG
AATCACTG
CACTG
G
4
5
7
3
6
10
G
AATCACTG
AATCACTG
G
1
8
2
9

ACTAATCACTG
A
C
T
G
11
ATCACTG
CT
TCACTG
T
ACTG
AATCACTG
CACTG
G
4
5
7
3
6
10
G
AATCACTG
AATCACTG
G
1
8
2
9

Suffix Links

- From an internal node that corresponds to the string $a\beta$ to the internal node that corresponds to β (if there is such node)

- Is there such a node ?

Suppose we create v applying rule 2. Then there was a suffix aβx… and now we add aβy

So there was a suffix βx…

- Is there such a node ?

Suppose we create v applying rule 2. Then there was a suffix aβx… and now we add aβy

So there was a suffix βx…

If there was also a suffix βz…

Then a node corresponding to β is there

- Is there such a node ?

Suppose we create v applying rule 2. Then there was a suffix aβx… and now we add aβy

So there was a suffix βx…

If there was also a suffix βz…

Then a node corresponding to β is there

Otherwise it will be created in the next extension when we add βy

Inv: All suffix links are there except (possibly) of the last internal node added

You are at the (internal) node corresponding to the last extension

Remember: we apply rule 2

You start a phase at the last internal node of the first extension in which you applied rule 3 in the previous iteration

1) Go up one node (if needed) to find a suffix link

2) Traverse the suffix link

3) If you went up in step 1 along an edge that was labeled δ then go down consuming a string δ

Create the new internal node if necessary

Create the new internal node if necessary

Create the new internal node if necessary, add the suffix

Create the new internal node if necessary, add the suffix and install a suffix link if necessary

Analysis

Handling all extensions of rule 1 and all extensions of rule 3 per phase take O(1) time ➔ O(n) total

How many times do we carry out rule 2 in all phases ?

O(n)

Does each application of rule 2 takes constant time ?

No ! (going up and traversing the suffix link takes constant time, but then we go down possibly on many edges..)

β
aβ
δ
δ
y
v
x..
y

So why is it a linear time algorithm ?

How much can the depth change when we traverse a suffix link ?

It can decrease by at most 1

Punch line

Each time we go up or traverse a suffix link the depth decreases by at most 1

When starting the depth is 0, final depth is at most n

So during all applications of rule 2 together we cannot go down more than 3n times

THM: The running time of Ukkonen's algorithm is O(n)

Another application

- Suppose we have a pattern P and a text T and we want to find for each position of T the length of the longest substring of P that matches there.

- How would you do that in O(n) time ?

Drawbacks of suffix trees

- Suffix trees consume a lot of space
- It is O(n) but the constant is quite big
- Notice that if we indeed want to traverse an edge in O(1) time then we need an array of ptrs. of size $|\Sigma|$ in each node

Suffix array

- We loose some of the functionality but we save space.

Let s = abab

Sort the suffixes lexicographically:
ab, abab, b, bab

The suffix array gives the indices of the suffixes in sorted order

3	1	4	2

How do we build it ?

- Build a suffix tree
- Traverse the tree in DFS, lexicographically picking edges outgoing from each node and fill the suffix array.

- O(n) time

How do we search for a pattern ?

- If P occurs in T then all its occurrences are consecutive in the suffix array.
- Do a binary search on the suffix array
- Takes O(mlogn) time

Example

Let S = mississippi

Let P = issa

L →	11	i
	8	ippi
	5	issippi
	2	ississippi
	1	mississippi
M →	10	pi
	9	ppi
	7	sippi
	4	sisippi
	6	ssippi
R →	3	ssissippi

How do we accelerate the search ?

Maintain l = LCP(P,L)
Maintain r = LCP(P,R)

If l = r then start comparing M to P at l + 1

How do we accelerate the search ?

If l > r then

Suppose we know LCP(L,M)

If LCP(L,M) < l we go left

If LCP(L,M) > l we go right

If LCP(L,M) = l we start comparing at l + 1

Analysis of the acceleration

If we do more than a single comparison in an iteration then max(l, r) grows by 1 for each comparison ➔ $O(\log n + m)$ time

Suffix Trees

Charles Yan

2008

Suffix Trees: Motivations

Substring problem: One is given a text T of length m. After O (m) preprocessing time, one must be prepared to take a pattern P of length n as input and find an occurrence of P in T or determine P does not exist in T in O(n) time.

- m is a larger number, e.g. the size of human genome.
- Multiple patterns input by different users. Thus, can not use exact set matching.
- O(m) preprocessing time. After that, each search of P must be done in O(n) time.

- Boyer-Moore alg. requires O (m+n) for each input pattern.
- Using a suffix tree, it only requires O(n) to find the occurrence of P in T for each P.

Suffix Trees: Motivations

The text T is a fixed *set* of strings. The goal is to determine whether an input pattern P is a substring of any of the fixed strings in T.

Dictionary problem using keyword tree: whether the input string match a *full* string in the dictionary. It won't work in this case.

Suffix trees …

Suffix Trees

Suffix trees can be used to solve in linear time

- exact matching problem.
- many string problems more complicated than exact matching.
- "We know of no other single data structure that allows efficient solutions to such a wide range of complex string problems"

Suffix Trees

A suffix tree T for an m-character string S

- A rooted directed tree with exactly m leaves numbered from 1 to m.
- Each internal node, other than root, has at least two children and each edge is labeled with a non-empty substring of S.
- No two edges out of a node can have edge-labels beginning with the same character.
- For any leave i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of S that starts at position i, that is, spells out S[i,…,m]

Suffix Trees

The suffix tree for string xabxac

Suffix Trees

What is the suffix tree for string xabxa ?

If one suffix of S matches a prefix of another suffix of S, then no suffix tree satisfying the above definition exists.

Suffix Trees

To avoid the problem, we add a special character $ to the end of string S.

$ does not appear in S. Thus, no suffix of S$ can be prefix of another suffix of S$.

In this chapter, string S is assumed to be extended with $ even if the symbol is not explicitly shown.

xabxa$

Suffix Trees

Differences between a suffix tree and a keyword tree:

Keyword Trees vs. Suffix Trees

A keyword tree for a set P is a rooted directed tree k satisfying three conditions: (1) each edge is labeled with one character; (2) any two edges out of the same node have distinct labels; and (3) every pattern P_i in P maps to some node v of K such that the characters on the path from the root of K to v exactly spell out P_i and every leaf of K is mapped to by some pattern in P.

A suffix tree T for an m-character string S

- A rooted directed tree with exactly m leaves numbered from 1 to m.
- Each internal node, other than root, has at least two children and each edge is labeled with a non-empty substring of S.
- No two edges out of a node can have edge-labels beginning with the same character.
- For any leave i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of S that starts at position i, that is, spells out S[i,…,m]

Keyword Trees vs. Suffix Trees

P={potato, poetry, pottery, science, school}

The suffix tree for string xabxac

Keyword Trees vs. Suffix Trees

Relationships between a suffix tree and a keyword tree:

For string S, P is the *set* of suffixes of S.

- Construct the keyword tree for set P.
- Merge any path of non-branching nodes into a single edge
- Then we get the suffix tree of S.

S=xabxac, P={xabxac, abxac, bxac, xac, ac, c}

Suffix Trees

$|S|=m$, the total lengths of patterns in P is $(m+1)*m/2$.
The algorithm is $O(m^2)$ time.

Suffix Trees

Label of path: from the root to a node (or a point) is the concatenation of all the substrings labeling the edges of that path.

Path-label of a node (Label of a node): The label of the path from the root of T to that node.

String-depth of a node v: the number of characters in v's label.

Motivating Example

How to use suffix trees for exact matching?

- Given a pattern P of length n and a text T of length m.
- Build a suffix tree $\mathcal{T}$ for text T in O(m) time.
- Match the characters of P along the unique path in $\mathcal{T}$, until either (1) P is exhausted or (2) no more matches are possible.
- Case 1: Every leaf in the subtree below the point of the last match shows a starting position of P in T
- Case 2: P does not occurs in T.

Motivating Example

T: xabxac

P: xa

Motivating Example

Time complexity

- Build the suffix tree: O(m)
  - To be done.
- Match P to the unique path: O(n)
  - Assume the size of the alphabet is finite.
- Traverse the tree below the last matching point: O(k), where k is the number of occurrences, i.e., the number of leaves below the last matching point.
  - Easy to prove.
  - The substree having k leaves has at most 2k-1 edges.
- Overall O(m+n+k).

Suffix Trees

Substring problem: One is given a text T of length m. After O (m) preprocessing time, one must be prepared to take a pattern P of length n as input and find an occurrence of P in T or determine P does not exist in T in O(n) time.

The text T is a fixed *set* of strings. The goal is to determine whether an input pattern P is a substring of any of the fixed strings in T.

Suffix Trees

String S with length of m.

N_i: is the intermediate tree consisting of all suffixes from 1 to i.

Then, N_m is the suffix tree we want.

A naïve algorithm to build a suffix tree for string S:

Create a single edge for suffix 1, i.e. S[1,…,m]$

For i=2;i<m;i++

Add suffix i into tree N_{i-1} to create N_i

$O(m^2)$

Suffix Trees

S=xabxa$

Suffix Trees

Ukkonen's algorithm: Linear time construction of suffix trees.

An implicit suffix tree for string S is a tree obtained from the suffix tree for S$ by

(1) removing $ from every leaf;

(2) removing any edge that has no label;

(3) removing any node that has less than two children.

I_i : The implicit suffix tree of substring S[1,…i]

Suffix Trees

I_5 for xabxa$

Suffix Trees

The implicit suffix tree has fewer leaves than the corresponding suffix tree is and only if some suffixes of S is a prefix of another suffix.

Even though an implicit tree may not have a leave for each suffix, it does encode all the suffixes of S.

Each suffix is spelled out by a path from the root to a leaf or the middle of an edge (no marker).

An implicit suffix tree is less informative than the corresponding suffix tree.

Suffix Trees

Construct an implicit suffix tree I_i for each prefix S[1,…,i], starting from I_1 and incrementing i by one until I_m is built.

The suffix tree for S is constructed from I_m.

Ukkonen Algorithm

Input: String S
Output: A suffix tree of S
Ukkonen Alogrithm

```
Construct tree I1.
For (i=1;i<m;i++) do
      begin {phase i+1}
      For (j=1;j<i+1;j++) do
          begin {extension j}
          Find the end of the path from the root labeled S[j…i] in the
          current tree. If needed extend that path by adding character
          S[i+1], thus ensuring that string S[j,…,i+1] is in the tree.
      end;
end;
```

Ukkonen Algorithm

I_1 is a tree with a single edge labeled with character S[1].

In phase i+1, tree I_{i+1} is constructed from I_i.

In extension j of phase i+1, substring S[j,…,i+1] is added (by extending S[j,…,i]).

After i+1 extensions, S[1,…,i+1], S[2,…,i+1], S[3,…,i+1],…,S[i+1], are added. Thus I_{i+1} is constructed.

Ukkonen Algorithm

In extension j of phase i+1, substring S[j,…,i+1] is added by extending S[j,…,i].

Let β= S[j,…,i],

Rules of extensions

Rule 1: β ends at a leaf in the current tree (I_i), add character S[i+1] to the end of β.

Rule 2: At least one labeled path continues from the end of β, but no path starts with character S[i+1], create a new leaf edge starting from the end of β and label the edge with character S[i+1] and the leave with j.

Rule 3: Some labeled path from the end of β starts with character S[i+1]. Do nothing.

Ukkonen Algorithm

S=axabxb

I_5

Phase i+1=6,
extension j=1

Phase i+1=6,
extension j=2

Phase i+1=6,
extension j=3

Phase i+1=6,
extension j=4

Phase i+1=6,
extension j=5

Phase i+1=6,
extension j=6

I_6

Ukkonen Algorithm

In phase i+1, extension j, once the end of β is found, only constant time is needed to execute the extension rules.

How to locate the end of β?

Naive approach: Start from the root find the end of the path that spell out β.

O(|β|) for a suffix β (each extension).

O(i+1-j) for extension j of phase i+1.

$$\sum_{j=1}^{i+1} O(i+1-j) = O(i^2)$$ for phase i+1

$$\sum_{i=1}^{m} O(i^2) = O(m^3)$$ for m phases (construction of I_m from I_1)

Suffix Trees

Construct an implicit suffix tree I_i for each prefix S[1,…,i], starting from I_1 and incrementing i by one until I_m is built.
O (m^3) !!! Need to be speeded up to O(m).

The suffix tree for S is constructed from I_m.

Ukkonen Algorithm

Suffix links

Let xα denote an arbitrary string, where x denotes a single character and α denotes a (possible empty) substring. For an internal node v with path-label xα, if there is another node s(v) with path-label α, then a pointer from v to s(v) is called a suffix link, denoted as (v,s(v)).

The root has no suffix link from it.

If α is empty, then the suffix link points the root.

Failure Links

v: a node in keyword tree K

L(v): the label on v, that is, the concatenation of characters on the path from the root to v.

lp(v): the length of the longest proper suffix of string L(v) that is a prefix of some pattern in P. Let this substring be α.

Lemma. There is a unique node in the keyword tree that is labeled by string α. Let this node be n_v. Note that n_v can be the root.

The ordered pair (v, n_v) is called a **failure link**.

Failure Links

P={potato, tattoo, theater, other}

Failure Links

Ukkonen Algorithm

Suffix links

Let xα denote an arbitrary string, where x denotes a single character and α denotes a (possible empty) substring. For an internal node v with path-label xα, if there is another node s(v) with path-label α, then a pointer from v to s(v) is called a suffix link, denoted as (v,s(v)).

The root has no suffix link from it.

If α is empty, then the suffix link points to the root.

This definition does not guarantee every internal node has a suffix link from it.

Ukkonen Algorithm

Every internal node in a implicit suffix tree has a suffix link from it.

Lemma 6.1.1 If a new internal node v with path-label $x\alpha$ is created in extension j of phase i+1, then an internal node w with path-label α already exists or will be created in extension j+1 in the same phase i+1.

Ukkonen Algorithm

Phase i+1
Extension j

Phase i+1
Extension j+1

Ukkonen Algorithm

Any newly created internal node, will have an suffix link from it at the end of next extension.

The extension (j=i+1) (the last extension) of phase i+1 does not create new internal node.

In any implicit suffix tree, every internal node v will have a s(v), i.e., has a suffix link from it.

In any implicit suffix tree I_i , if internal node v has a a path-label $x\alpha$, then there is node s(v) of I_i with path-label α.

Ukkonen Algorithm

In phase i+1, extension j, once the end of β is found, only constant time is needed to execute the extension rules.

How to locate the end of β?

Naive approach: Start from the root find the end of the path that spell out β. $O(m^3)$

Use the suffix link.

Ukkonen Algorithm

In the construction of I_i, keep a pointer P to leaf 1.

In I_i , the path-label of leaf 1 is S[1,…,i]

In the construction of I_{i+1}, the edge leading to leaf 1 will be extended by rule 1.

Leaf 1 in I_i will become leaf 1 in I_{i+1}.

The pointer to leaf 1 does not need to be updated.

S=axabxb

S[1..5]=axabx

Phase i+1=6,
extension j=1

S[1..6]=axabxb

Ukkonen Algorithm

For phase i+1,

In extension 1, pointer P indicates the end of β.

Ukkonen Algorithm

Phase i+1
Extension j=1
To add (1,i+1)=xαabcd
β=S(j,i)=xαabc

I_i

Label (1)=xαabc

Label (1)=xαabcd

Ukkonen Algorithm

For phase i+1,

In extension 1, pointer P indicates the end of β.

Let be a pointer pointing to P.

For extension j=2,…i+1, find the end of β by:

- Start with the node (k) that is pointed to by w.
- Walk up one edge and reach node v. let γ be the label of the edge (v,k)
- Follow the suffix link from v and reach s(v). If v is the root, then s(v) is also the root.
- Walk down the path that spells out γ .
- The end of the path is the end of β
- Move w to the end of β

Ukkonen Algorithm

Phase i+1
Extension j=2
Need to add
S(2,i+1)=αabcd
β=S(j,i)=αabc

Ukkonen Algorithm

Phase i+1
Extension j=3
Need to add S(3,i+1)=λabcd
β=S(j,i)=λabc

I_i

Ukkonen Algorithm

For phase i+1,

In extension 1, pointer P indicates the end of β.

Let pointer w point to P.

For extension j=2,…i+1, we find the end of β by:

- Starting with the node (k) that is pointed to by w.
- Walk up one edge and reach node v. let γ be the label of the edge (v,k)
  - if g is an internal node, there is no need to walk up. v=k
- Follow the suffix link from v and reach s(v).
- Walk down the path that spells out γ .
  - If v is the root, (there is no suffix link from the root) then walk down a path that spells out β.
- The end of the path is the end of β
- Move w to the end of β
- If a new node (z) was created in extension j-1, then create the suffix link for z. s(z) is the first internal node above or at pointer w in the current tree.

Ukkonen Algorithm

Phase i+1
Extension j=3
Need to add S(3,i+1)=λabcd
β=S(j,i)=λabc

I_i

Ukkonen Algorithm

Input: String S
Output: A suffix tree of S
Ukkonen Alogrithm

```
Construct tree I_1.
For (i=1;i<m;i++) do
        begin {phase i+1}
        For (j=1;j<i+1;j++) do
            begin {extension j}
            Find the end of the path from the root labeled S[j…i] in the
                    current tree. If needed extend that path by adding character S[i+1],
                    thus ensuring that string S[j,…,i+1] is in the tree.
        end;
end;
```

How to locate the end of β?

Naive approach: Start from the root find the end of the path that spell out β. $O(m^3)$

Use suffix links: When the tree has no internal at all,

the running time is still $O(m^3)$!!!! ☹

Ukkonen Algorithm

We will be able to reduce the running to O(m) by applying three tricks.

Trick 1. Skip/count trick

The down walk from s(v) takes time proportional to $|\gamma|$, i.e. the number of characters that γ consists of.

g be the number of characters that the algorithm needs to walk down.

g starts with $|\gamma|$.

h be the index of the character in γ that the edge (e) to be traversed should start with. h starts with 1.

g` be the number of characters on the edge (e) to be traversed.

Ukkonen Algorithm

If g≥g`, skip to next node; g=g-g`; h=h+g'; e be the edge starts with γ[h]

else, go to the gth character on edge e.

Achievement: the walk down take time proportional to the number of nodes on the path, in stead of the number of characters.

Keep track of the number of characters on each edge.

Move from one node to the other node of an edge in constant time (Adjacency list).

Ukkonen Algorithm

Achievement: the walk down take time proportional to the number of nodes on the path, in stead of the number of characters.

Theorem 6.1.1. Using the skip/count trick, any phase of Ukkonen Algorithm takes O(m) time.

This is an improvement over the naive approach, which takes $O(m^2)$ for each phase.

Ukkonen Algorithm

In phase i+1, extension j, once the end of β is found, only constant time is needed to execute the extension rules.

How to locate the end of β?

Naive approach: Start from the root find the end of the path that spell out β.

O(|β|) for a suffix β (each extension).

O(i+1-j) for extension j of phase i+1.

$$\sum_{j=1}^{i+1} O(i+1-j) = O(i^2)$$ for phase i+1

$$\sum_{i=1}^{m} O(i^2) = O(m^3)$$ for m phases (construction of I_m from I_1)

Ukkonen Algorithm

Still need to prove Theorem 6.1.1

The node depth of a node u is the number of nodes on the path from the root to u.

Lemma 6.12. Let (v, s(v)) be a suffix link, then the node depth of v is at most one greater than the node depth of s(v).

Ukkonen Algorithm

Label (v)=xα Label (s(v))=a
Label (u)=xβ Label (s(u))=β

(v,s(v)) is a suffix link.

The suffix link from any internal ancestor of v goes to an ancestor of s(v).

Ukkonen Algorithm

(v,s(v)) is a suffix link.

The suffix link from any internal ancestor of v goes to an ancestor of s(v).

The only extra ancestor that v can have is the internal node whose label consists of only one character.

Thus, v can have at most one more ancestor than s(v).

The node depth of v is at most one greater than the node depth of s(v).

On the other hand, s(v) have more ancestors than v.

Ukkonen Algorithm

As the algorithm proceeds, the current node depth (cd) of the algorithm is the node depth of the node most recently visited by the algorithm.

Theorem 6.1.1. Using the skip/count trick, any phase of Ukkonen Algorithm takes O(m) time.

Only need to analyze the time for down-walks.

In extension j of phase i+1

- Up-walk
  - decreases by at most 1
- Traverse
  - decreases by at most 1
- down-walk
  - increase by n_j, the number of nodes walk down.

Ukkonen Algorithm

As the algorithm proceeds, the current node depth (cd) of the algorithm is the node depth of the node most recently visited by the algorithm.

Theorem 6.1.1. Using the skip/count trick, any phase of Ukkonen Algorithm takes O(m) time.

Then in the entire phase i+1,

The total decrement is at most 2*(i+1)

The total increment is $\sum_{j=1}^{i+1} n_j$

Since cd≤m at any time,

$$\sum_{j=1}^{i+1} n_j - 2*(i+1) \le m \qquad \sum_{j=1}^{i+1} n_j \le 3m$$

Walks down O(m) nodes in phase i+1

The total time for all down-walks in a phase is O(m)

Ukkonen Algorithm

Ukkonen algorithm can be implemented with suffix links to run in $O(m^2)$ time.

Keyword Trees vs. Suffix Trees

Relationships between a suffix tree and a keyword tree:

For string S, P is the *set* of suffixes of S.

- Construct the keyword tree for set P.
- Merge any path of non-branching nodes into a single edge
- Then we get the suffix tree of S.

S=xabxac, P={xabxac, abxac, bxac, xac, ac, c}

Suffix Trees

|S|=m, the total lengths of patterns in P is (m+1)*m/2.
The algorithm is $O(m^2)$ time.

$O(m^2) \rightarrow O(m^2)$ ☹!!!

Ukkonen Algorithm

For a string S with m characters, its suffix tree has $O(m^2)$ characters!

S=abcdefghij…xyz

To output a $O(m^2)$ tree in O(m) time…

Mission impossible!

Need to find alternate representation of the tree that takes only O(m) space.

Edge label compression: Label each edge with a pair of indices, specifying the beginning and end positions of the edge label in S

Ukkonen Algorithm

S=abcdefabcuvw

At most m leaves
At most 2m-1 edges
O(m) space

Ukkonen Algorithm

In extension j of phase i+1, substring S[j,…,i+1] is added by extending S[j,…,i].

Let β= S[j,…,i],

Rules of extensions

Rule 1: β ends at a leaf in the current tree (I_i), add character S[i+1] to the end of β. Change the edge label from (p,q) to (p, q+1)

q=i, i.e., from (p,i) to (p,i+1)

Rule 2: At least one labeled path continues from the end of β, but no path starts with character S[i+1], create a new leaf edge starting from the end of β and label the edge with character S[i+1] and the leave with j. The newly created edge is labeled with (i+1,i+1)

Rule 3: Some labeled path from the end of β starts with character S[i+1]. Do nothing.

Ukkonen Algorithm

Observation 1. Rule 3 is a show stopper.

In any phase i+1, if rule 3 applies in extension j, it will also apply in all further extensions (j+1 to i+1).

Ukkonen Algorithm

Phase i+1
Extension j
S(j,i+1)=xyzαabcd
β_1 =S(j,i)=xyzαabc

Phase i+1
Extension j+1
S(j+1,i+1)=yzαabcd
β_2 =S(j+1,i)=yzαabc

Phase i+1
Extension j+2
S(j+2,i+1)=zαabcd
β_3 =S(j+2,i)=zαabc

Rule 3 → β_1d is in I_i

β_2d is in I_i→ Rule 3

β_3d is in I_i→ Rule 3

Ukkonen Algorithm

Observation 1. Rule 3 is a show stopper.

In any phase i+1, if rule 3 applies in extension j, it will also apply in all further extensions (j+1 to i+1).

Trick 2.

In any phase i+1, if rule 3 applies in extension j, then end that phase and go to the next phase.

Extensions j+1,…,i+1 are done implicitly.

Extensions 1, …, j are done explicitly. Explicit extensions.

Ukkonen Algorithm

Phase	Extension													
	1	2	3	4	5	6	7	8	9	…	i	i+1	i+2	…
i	1	2	1	2	3	3	3	3	3	…	3			
i+1	1	1	1	1	2	1	3	3	3	…	3	3		
i+2	1	1	1	1	1	1	2	2	3	…	3	3		
…														

Ukkonen Algorithm

Observation 2. Once a leaf always a leaf.

If at some point in the Ukkonen algorithm a leaf is created and labeled with j in extension j of phase i (rule 2 applies), then in extension j of phase i+1, that leaf will be extended by adding a new character to the edge label (rule 1 applies).

- Rule 2 applies in extension j of phase i → rule 1 applies in extension j of phase i+1.

Ukkonen Algorithm

S=axabxbd

Phase i+1=6,
extension j=1

Phase i+1=6,
extension j=2

Phase i+1=6,
extension j=3

Phase i+1=6,
extension j=4

Phase i+1=6,
extension j=5

Phase i+1=6,
extension j=6

Phase i+1=7,
extension j=5

Ukkonen Algorithm

Observation 2. Once a leaf always a leaf.

If at some point in the Ukkonen algorithm a leaf is created and labeled with j in extension j of phase i (rule 2 applies), then in extension j of phase i+1, that leaf will be extended by adding a new character to the edge label (rule 1 applies).

- Rule 2 applies in extension j of phase i → rule 1 applies in extension j of phase i+1.

Once leaf j is created, it will remain leaf j in all successive trees created.

- Rule 1 applies in extension j of phase i → rule 1 applies in extension j of phase i+1.

Ukkonen Algorithm

S=axabxbd

Phase i+1=6,
extension j=1

Phase i+1=6,
extension j=2

Phase i+1=6,
extension j=3

Phase i+1=6,
extension j=4

Phase i+1=6,
extension j=5

Phase i+1=6,
extension j=6

Phase i+1=7,
extension j=1,…4

Ukkonen Algorithm

Phase	Extension													
	1	2	3	4	5	6	7	8	9	…	i	i+1	i+2	…
i	1	2	1	2	3	3	3	3	3	…	3			
i+1	1	1	1	1	2	1	3	3	3	…	3	3		
i+2	1	1	1	1	1	1	2	2	3	…	3	3	3	
…														

Ukkonen Algorithm

In any phase i, there is an initial sequence of consecutive extensions (starting with extension 1) where rule 1 or 2 applies. Let j_i denotes the last extension of this sequence.

This sequence can not shrink in successive phases. $j_{i+1} \geq j_i$

Ukkonen Algorithm

Phase	Extension													
	1	2	3	4	5	6	7	8	9	…	i	i+1	i+2	…
i	1	2	1	2 j_i	3	3	3	3	3	…	3			
i+1	1	1	1	1	2	1 j_{i+1}	3	3	3	…	3	3		
i+2	1	1	1	1	1	1	2	2 j_{i+2}	3	…	3	3	3	
…														

Ukkonen Algorithm

Trick 3. Keep a global variable e. When the Ukkonen algorithm steps into phase i+1, e is set to i+1.

In phase i+1, a leaf edge would normally be labeled with substring S[p, i+1], instead of writing indices (p, i+1) on the edge, write (p,e).

Thus, in phase i+1, once e is updated, all leaf edges are updated implicitly

Ukkonen Algorithm

S=axabxbd

Phase i+1=6,
extension j=1

Phase i+1=6,
extension j=2

Phase i+1=6,
extension j=3

Phase i+1=6,
extension j=4

Ukkonen Algorithm

Trick 3. Keep a global variable e. When the Ukkonen algorithm steps into phase i+1, e is set to i+1.

In phase i+1, a leaf edge would normally be labeled with substring S[p, i+1], instead of writing indices (p, i+1) on the edge, write (p,e).

Thus, in phase i+1, once e is updated, all leaf edges are updated implicitly

All the extensions in which rule 1 applies will be done implicitly by updating e.

In phase i+1, extensions 1 through j_i will be done implicitly by updating e.

Ukkonen Algorithm

Phase	Extension													
	1	2	3	4	5	6	7	8	9	…	i	i+1	i+2	…
i	1	2	1	2 j_i	3	3	3	3	3	…	3			
i+1	1	1	1	1	2	1 j_{i+1}	3	3	3	…	3	3		
i+2	1	1	1	1	1	1	2	2 j_{i+2}	3	…	3	3	3	
…														

Ukkonen Algorithm

In phase i+1,

- Extensions 1 through j_i are done explicitly by updating e;
- Explicitly compute extensions starting from j_i+1, until the first extension (let it be j*) where rule 3 applies;
- Set j_{i+1}=j*-1 to prepare for the next phase.

Ukkonen Algorithm

Input: String S of m characters
Output: Implicit suffix tree I_m of S
Ukkonen Alogrithm

```
e=1;   j1=1;
Construct tree I1.
For (i=1;i<m;i++) do
        begin {phase i+1}
        e=i+1; (this will correctly implement extension 1 through ji)
        j*=i+2;
        For (j=ji+1; j <i+1; j++) do
            begin {extension j}
            Find the end of the path from the root labeled S[j…i]. (up-walk, traversal,
                    down-walk)
            Apply one of the extension rules to ensure that string S[j,…,i+1] is in the tree.
            If rule 3 is applies, then go to next phase and j*=j;
                    (Explicitly compute extensions starting from ji until the first extension
                    where rule 3 applies).
        end;
        ji+1=j*-1;
end;
```

Ukkonen Algorithm

Phase	Extension													
	1	2	3	4	5	6	7	8	9	…	i	i+1	i+2	…
i	1	2	1	2 j_i	3 j^*	3	3	3	3	…	3			
i+1	1	1	1	1	2	1 j_{i+1}	3 j^*	3	3	…	3	3		
i+2	1	1	1	1	1	1	2	2 j_{i+2}	3 j^*	…	3	3	3	
…														

Ukkonen Algorithm

In phase i+1, extension j_i+1 (the last extension that are explicitly computed in phase i),

- How to find the end of β (i.e., S [j_i+1, i]) ? (we can not start from pointer p that points to 1).
- The end of β is where the algorithm stops in the previous phase (pointed to by w). (in extension j_i+1 of phase i, S [j_i+1, i] was added to the tree).
- There is no need to search for it explicitly.

Ukkonen Algorithm

Phase i
Extension $j = j_i+1$
add $S(j_i+1,i)=\alpha abc$

Phase i+1
Extension $j=j_i+1$
Need to add
$S(j_i+1,i+1)=\alpha abcd$
$\beta=S(j_i+1,i)=\alpha abc$

Ukkonen Algorithm

Input: String S of m characters
Output: Implicit suffix tree I_m of S
Ukkonen Alogrithm

```
e=1;   j1=1;
Construct tree I1.
For (i=1;i<m;i++) do
        begin {phase i+1}
        e=i+1; (this will correctly implement extension 1 through ji)
        j*=i+2;
        For (j=ji+1; j <i+1; j++) do
            begin {extension j}
            if (j>ji+1) then find the end of the path from the root labeled S[j…i]. (up-walk,
                                traversal, down-walk)
            Apply one of the extension rules to ensure that string S[j,…,i+1] is in the tree.
            If rule 3 is applies, then go to next phase and j*=j;
                    (Explicitly compute extensions starting from ji until the first extension
                    where rule 3 applies).
        end;
        ji+1=j*-1;
end;
```

Ukkonen Algorithm

Using suffix links and tricks 1,2, and 3, the Ukkonen algorithm builds implicit suffix trees I_1 through I_m in O(m) total time.

Implicit extensions: Constant time O(1)

Explicit extensions:

- At most 2m explicit extensions
- Up-walks, traversal O(m)
- Down-walks O(m)

Ukkonen Algorithm

How to create the true suffix tree?

Add $ to the end of S. O(1)

Create I_1 through I_{m+1} using the Ukkonen algorithm. O(m)

Replace each index e on every leaf edge with m. O(m)

Ukkonen algorithm can build a true suffix tree for S along with all its suffix links in O(m) time.

Suffix tree and suffix array techniques for pattern analysis in strings

Esko Ukkonen

Univ Helsinki

Erice School 30 Oct 2005

tttttttttttttgagacggagtctcgctctg
tcgcccaggctggagtgcagtggcggg
atctcggctcactgcaagctccgcctcc
cgggttcacgccattctcctgcctcagcc
tcccaagtagctgggactacaggcgcc
cgccactacgcccggctaatttttgtattt
ttagtagagacggggtttcaccgttttagc
cgggatggtctcgatctcctgacctcgtg
atccgcccgcctcggcctcccaaagtgc
tgggattacaggcgt

Algorithms for combinatorial string matching?

- deep beauty? +-
- shallow beauty? +
- applications? ++
- intensive algorithmic miniatures
- sources of new problems:
  text processing, DNA, music,…

Analysis of a string of symbols

- T = h a t t i v a t t i ’*text*’
- P = a t t ’*pattern*’

- Find the occurrences of P in T:

  h a t t i v a t t i

- Pattern synthesis: #(t) = 4 #(atti) = 2
  #(t****t) = 2

Pattern finding & synthesis problems

- $T = t_1 t_2 \ldots t_n$, $P = p_1 p_2 \ldots p_n$, strings of symbols in finite alphabet

- Indexing problem: Preprocess T (build an <u>index</u> structure) such that the occurrences of different patterns P can be found fast
  - static text, any given pattern P

- Pattern synthesis problem: Learn from T new patterns that occur surprisingly often

- What is a pattern? Exact substring, approximate substring, with generalized symbols, with gaps, ...

1. **Suffix tree**
2. Suffix array
3. Some applications
4. Finding motifs

The suffix tree Tree(T) of T

- data structure suffix tree, Tree(T), is <u>compacted trie</u> that represents all the suffixes of string T
- linear size: |Tree(T)| = O(|T|)
- can be constructed in linear time O(|T|)
- has *myriad virtues* (A. Apostolico)
- is well-known: 366 000 Google hits

Suffix trie and suffix tree

abaab
baab
aab
ab
b

Trie(abaab)

Tree(abaab)

Trie(T) can be large

- $|Trie(T)| = O(|T|^2)$
- bad example: $T = a^n b^n$
- Trie(T) can be seen as a DFA: language accepted = the suffixes of T
- minimize the DFA => directed cyclic word graph (’DAWG’)

Tree(T) is of linear size

- only the internal branching nodes and the leaves represented explicitly
- edges labeled by substrings of T
- v = node(α) if the path from root to v spells α
- one-to-one correspondence of leaves and suffixes
- |T| leaves, hence < |T| internal nodes
- |Tree(T)| = O(|T| + size(edge labels))

Tree(hattivatti)

hattivatti
attivatti
ttivatti
tivatti
ivatti
vatti
atti
tti
ti
i

Tree(hattivatti)

hattivatti
attivatti
ttivatti
tivatti
ivatti
vatti
atti
tti
ti
i

substring labels of edges represented as pairs of pointers

Tree(hattivatti)

hattivatti
attivatti
ttivatti
tivatti
ivatti
vatti
atti
tti
ti
i

Tree(T) is *full* text index

P occurs in T at
locations 8, 31, …

P occurs in T ⇔ P is a prefix of some suffix of T
⇔ Path for P exists in Tree(T)

All occurrences of P in time O(|P| + #occ)

Find att from Tree(hattivatti)

hattivatti
attivatti
ttivatti
tivatti
ivatti
vatti
atti
tti
ti
i

Linear time construction of Tree(T)

hattivatti

attivatti

ttivatti

tivatti

ivatti

vatti

atti

tti

ti

i

Weiner 1973. (algorithm of the year)

McCreight 1976.

'on-line' algorithm - Ukkonen 1992.

On-line construction of Trie(T)

- $T = t_1 t_2 \ldots t_n \$$
- $P_i = t_1 t_2 \ldots t_i$ i:th prefix of *T*
- <u>on-line idea</u>: update *Trie*(P_i) to *Trie*(P_{i+1})
- => very simple construction

Trie(abaab)

Trie(a) *Trie(ab)* *Trie(aba)*

aba**a**
ba**a**
a**a**
ε**a**
ε

chain of links suffixes connects the end points of current

Trie(abaab)

Trie(abaa)

Trie(abaab)

Trie(abaa)

Add next symbol = b

Trie(abaab)

Trie(abaa)

Add next symbol = b

From here on b-arc already exists

Trie(abaab)

Trie(abaab)

What happens in *Trie*(P_i) => *Trie*(P_{i+1}) ?

Before

After

What happens in $Trie(P_i) \Rightarrow Trie(P_{i+1})$?

- time: O(size of Trie(T))
- suffix links:

  slink(node($a\alpha$)) = node(α)

On-line procedure for suffix trie

1. Create *Trie*(t_1)*:* nodes *root* and *v*, an arc *son(root,* t_1*)* = *v,* and suffix links *slink(v) := root* and *slink(root) := root*

2. **for** *i := 2* to *n* **do begin**

3. v_{i-1} := leaf of *Trie*($t_1 \ldots t_{i-1}$) for string $t_1 \ldots t_{i-1}$ (i.e., the deepest leaf)

4. $v := v_{i-1}$; $v' := 0$

5. **while** node *v* has no outgoing arc for t_i **do begin**

6. Create a new node v'' and an arc $son(v,t_i) = v''$

7. **if** $v' \neq 0$ **then** $slink(v) := v''$

8. $v := slink(v)$; $v' := v''$ **end**

9. **for** the node v'' such that $v'' = son(v,t_i)$ **do**
   if $v'' = v'$ **then** *slink(v') := root* **else** $slink(v') := v''$

Suffix trees on-line

- ’compacted version’ of the on-line trie construction: simulate the construction on the linear size tree instead of the trie => time O(|T|)
- all trie nodes are conceptually still needed => *implicit* and *real* nodes

Implicit and real nodes

- Pair (v,*α*) is an *implicit node* in Tree(T) if v is a node of Tree and *α* is a (proper) prefix of the label of some arc from *v.* If *α* is the empty string then *(v, α)* is a *'real'* node (= *v*).
- Let *v* = *node(α´)* in Tree(T). Then implicit node *(v, α)* represents *node(α´α)* of Trie(T)

Implicit node

Suffix links and open arcs

label [i,*] instead of [i,j] if w is a leaf and j is the scanned position of T

Big picture

suffix link path traversed: total work O(n)

new arcs and nodes created:
total work O(size(Tree(T))

On-line procedure for suffix tree

Input: string $T = t_1 t_2 \ldots t_n\$$

Output: *Tree(T)*

Notation: $son(v,\alpha) = w$ iff there is an arc from v to w with label α

$son(v,\varepsilon) = v$

Function *Canonize(v, α):*

while $son(v, \alpha') \neq 0$ where $\alpha = \alpha' \alpha''$, $|\alpha'| > 0$ **do**

$v := son(v, \alpha')$; $\alpha := \alpha''$

return (v, α)

Suffix-tree on-line: main procedure

Create *Tree*(t_1); *slink(root) := root*

(v, *α) := (root, ε)* /* *(v, α)* is the start node */

for *i := 2* **to** *n+1* **do**

v´ := 0

while there is no arc from *v* with label prefix αt_i **do**

if *α ≠ ε* **then** /* divide the arc *w = son(v, αη)* into two */

son(v, α) := v´´; son(v´´,t_i*) := v´´´; son(v´´,η) := w*

else

son(v,t_i*) := v´´´; v´´ := v*

if *v´ ≠ 0* **then** *slink(v´) := v´´*

v´ := v´´; v := slink(v); (v, α) := Canonize(v, α)

if *v´ ≠ 0* **then** *slink(v´) := v*

(v, α) := Canonize(v, αt_i*)* /* *(v, α) = start node of the next round* */

The actual time and space

- |Tree(T)| is about 20|T| in practice
- brute-force construction is O(|T|log|T|) for random strings as the average depth of internal nodes is O(log|T|)
- difference between linear and brute-force constructions not necessarily large (Giegerich & Kurtz)
- truncated suffix trees: k symbols long prefix of each suffix represented (Na et al. 2003)
- alphabet independent linear time (Farach 1997)

1. Suffix tree
2. **Suffix array**
3. Some applications
4. Finding motifs

Suffix array: example

hattivatti	**ε**	**11**
attivatti	**atti**	**7**
ttivatti	**attivatti**	**2**
tivatti	**hattivatti**	**1**
ivatti	**i**	**10**
vatti	**ivatti**	**5**
atti	**ti**	**9**
tti	**tivatti**	**4**
ti	**tti**	**8**
i	**ttivatti**	**3**
ε	**vatti**	**6**

- suffix array = lexicographic order of the suffixes

Suffix array

- suffix array SA(T) = an array giving the lexicographic order of the suffixes of T
- space requirement: 5|T|
- practitioners like suffix arrays (simplicity, space efficiency)
- theoreticians like suffix trees (explicit structure)

Pattern search from suffix array

hattivatti	**ε**	**11**
attivatti	**atti**	**7**
ttivatti	**attivatti**	**2**
tivatti	**hattivatti**	**1**
ivatti	**i**	**10**
vatti	**ivatti**	**5**
atti	**ti**	**9**
tti	**tivatti**	**4**
ti	**tti**	**8**
i	**ttivatti**	**3**
ε	**vatti**	**6**

att

binary search

Recent suffix array constructions

- Manber&Myers (1990): O(|T|log|T|)
- linear time via suffix tree
- January / June 2003: direct linear time construction of suffix array
    - Kim, Sim, Park, Park (CPM03)
    - Kärkkäinen & Sanders (ICALP03)
    - Ko & Aluru (CPM03)

Kärkkäinen-Sanders algorithm

1. Construct the suffix array of the suffixes starting at positions $i \bmod 3 \neq 0$. This is done by reduction to the suffix array construction of a string of two thirds the length, which is solved recursively.
2. Construct the suffix array of the remaining suffixes using the result of the first step.
3. Merge the two suffix arrays into one.

Notation

- string $T = T[0,n) = t_0 t_1 \ldots t_{n-1}$
- suffix $S_i = T[i,0) = t_i t_{i+1} \ldots t_{n-1}$
- for C \subset [0,n]: $S_C = \{S_i \mid i \text{ in } C\}$

- *suffix array* SA[0,n] of T is a permutation of [0,n] satisfying $S_{SA[0]} < S_{SA[1]} < \ldots < S_{SA[n]}$

Running example

	0	1	2	3	4	5	6	7	8	9	10	11		
• T[0,n) =	y	a	b	b	a	d	a	b	b	a	d	o	0	0 …

- SA = (12,1,6,4,9,3,8,2,7,5,10,11,0)

Step 0: Construct a sample

- for $k = 0,1,2$

  $$Bk = \{i \in [0,n] \mid i \bmod 3 = k\}$$

- $C = B1 \cup B2$ *sample positions*
- S_C *sample suffixes*

- Example: $B1 = \{1,4,7,10\}$, $B2 = \{2,5,8,11\}$, $C = \{1,4,7,10,2,5,8,11\}$

Step 1: Sort sample suffixes

- for k = 1,2, construct

  $Rk = [t_k t_{k+1} t_{k+2}]\ [t_{k+3} t_{k+4} t_{k+5}] \ldots\ [t_{maxBk} t_{maxBk+1} t_{maxBk+2}]$

  R = R1 ^ R2 concatenation of R1 and R2

  Suffixes of R correspond to S_C: suffix $[t_i t_{i+1} t_{i+2}] \ldots$ corresponds to S_i ; correspondence is order preserving.

  Sort the suffixes of R: radix sort the characters and rename with ranks to obtain R´. If all characters different, their order directly gives the order of suffixes. Otherwise, sort the suffixes of R´ using Kärkkäinen-Sanders. Note: |R´| = 2n/3.

Step 1 (cont.)

- once the sample suffixes are sorted, assign a rank to each: rank(S_i) = the rank of S_i in S_C; rank(S_{n+1}) = rank(S_{n+2}) = 0
- Example:

  R = [abb][ada][bba][do0][bba][dab][bad][o00]

  R´ = (1,2,4,6,4,5,3,7)

  $SA_{R'}$ = (8,0,1,6,4,2,5,3,7)

  rank(S_i) - 1 4 - 2 6 - 5 3 – 7 8 – 0 0

Step 2: Sort nonsample suffixes

- for each non-sample $S_i \in S_{B0}$ (note that $\text{rank}(S_{i+1})$ is always defined for $i \in B0$):

  $$S_i \leq S_j \quad \leftrightarrow \quad (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1}))$$

- radix sort the pairs $(t_i, \text{rank}(S_{i+1}))$.

- Example: $S_{12} < S_6 < S_9 < S_3 < S_0$ because $(0,0) < (a,5) < (a,7) < (b,2) < (y,1)$

Step 3: Merge

- merge the two sorted sets of suffixes using a standard comparison-based merging:
- to compare $S_i \in S_C$ with $S_j \in S_{B0}$, distinguish two cases:

- $i \in B1$: $S_i \leq S_j \leftrightarrow (t_i, rank(S_{i+1})) \leq (t_j, rank(S_{j+1}))$
- $i \in B2$: $S_i \leq S_j \leftrightarrow (t_i, t_{i+1}, rank(S_{i+2})) \leq (t_j, t_{j+1}, rank(S_{j+2}))$

- note that the ranks are defined in all cases!
- $S_1 < S_6$ as $(a,4) < (a,5)$ and $S_3 < S_8$ as $(b,a,6) < (b,a,7)$

Running time O(n)

- excluding the recursive call, everything can be done in linear time
- the recursion is on a string of length 2n/3
- thus the time is given by recurrence

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

- hence $T(n) = O(n)$

Implementation

- about 50 lines of C++
- code available e.g. via Juha Kärkkäinen's home page

LCP table

- Longest Common Prefix of successive elements of suffix array:
- LCP[i] = length of the longest common prefix of suffixes $S_{SA[i]}$ and $S_{SA[i+1]}$
- build inverse array SA^{-1} from SA in linear time
- then LCP table from SA^{-1} in linear time (Kasai et al, CPM2001)

Suffix tree vs suffix array

- **suffix tree ⇔ suffix array + LCP table**

1. Suffix tree

2. Suffix array

3. **Some applications**

4. Finding motifs

Substring motifs of string T

- string T = $t_1 \ldots t_n$ in alphabet A.
- Problem: what are the frequently occurring (ungapped) substrings of T? Longest substring that occurs at least q times?

- Thm: Suffix tree *Tree(T)* gives complete occurrence counts of all substring motifs of T in $O(n)$ time (although T may have $O(n^2)$ substrings!)

Counting the substring motifs

- internal nodes of Tree(T) $\leftrightarrow$ repeating substrings of T
- number of leaves of the subtree of a node for string P = number of occurrences of P in T

Substring motifs of hattivatti

Counts for the *O(n)* <u>maximal motifs</u> shown

Finding repeats in DNA

- human chromosome 3
- the first 48 999 930 bases
- 31 min cpu time (8 processors, 4 GB)

- Human genome: $3x10^9$ bases
- Tree(HumanGenome) feasible

Longest repeat?

<u>Occurrences</u> at: 28395980, 28401554r <u>Length</u>: 2559

ttagggtacatgtgcacaacgtgcaggtttgttacatatgtatacacgtgccatgatggtgtgctgcacccattaactcgtcatttagcgttaggtatatctccgaat
gctatccctcccccctcccccacccacaacagtccccggtgtgtgatgttccccttcctgtgtccatgtgttctcattgttcaattcccacctatgagtgagaac
atgcggtgtttggtttttgtccttgcgaaagtttgctgagaatgatggtttccagcttcatccatatccctacaaaggacatgaactcatcattttttatggctgcata
gtattccatggtgtatatgtgccacattttcttaacccagtctacccttgttggacatctgggttggttccaagtctttgctattgtgaatagtgccgcaataaacatac
gtgtgcatgtgtctttatagcagcatgatttataatccttttgggtatatacccagtaatgggatggctgggtcaaatggtatttctagttctagatccctgaggaatca
ccacactgacttccacaatggttgaactagtttacagtcccagcaacagttcctatttctccacatcctctccagcacctgttgtttcctgactttttaatgatcgcca
ttctaactggtgtgagatggtatctcattgtggttttgatttgcatttctctgatggccagtgatgatgagcatttttttcatgtgtttttttggctgcataaatgtcttcttttga
gaagtgtctgttcatatccttcgcccacttttgatggggttgtttgtttttttcttgtaaatttgttggagttcattgtagattctgggtattagccctttgtcagatgagtag
gttgcaaaaattttctcccattctgtaggttgcctgttcactctgatggtggtttcttctgctgtgcagaagctctttagtttaattagatcccatttgtcaattttggctttt
gttgccatagcttttggtgttttagacatgaagtccttgcccatgcctatgtcctgaatggtattgcctaggttttcttctagggtttttatggttttaggtctaacatgta
agtctttaatccatcttgaattaattataaggtgtatattataaggtgtaattataaggtgtataattatatattaattataaggtgtatattaattataaggtgtaaggaag
ggatccagtttcagctttctacatatggctagccagttttccctgcaccatttattaaatagggaatcctttccccattgcttgtttttgtcaggtttgtcaaagatcaga
tagttgtagatatgcggcattatttctgagggctctgttctgttccattggtctatatctctgttttggtaccagtaccatgctgttttggttactgtagccttgtagtatag
tttgaagtcaggtagcgtgatggttccagctttgttcttttggcttaggattgacttggcaatgtgggctctttttggttccatatgaactttaaagtagtttttttccaatt
ctgtgaagaaattcattggtagcttgatggggatggcattgaatctataaattaccctgggcagtatggccattttcacaatattgaatcttcctacccatgagcgt
gtactgttcttccatttgtttgtatcctcttttatttcattgagcagtggtttgtagttctccttgaagaggtccttcacatcccttgtaagttggattcctaggtattttattct
ctttgaagcaattgtgaatgggagttcactcatgatttgactctctgtttgtctgttattggtgtataagaatgcttgtgatttttgcacattgattttgtatcctgagacttt
gctgaagttgcttatcagcttaaggagattttgggctgagacgatggggttttctagatatacaatcatgtcatctgcaaacagggacaatttgacttcctcttttcc
taattgaatacccgttatttccctctcctgcctgattgccctggccagaacttccaacactatgttgaataggagtggtgagagagggcatccctgtcttgtgcca
gttttcaaagggaatgcttccagtttttgtccattcagtatgatattggctgtgggtttgtcatagatagctcttattattttgagatacatcccatcaatacctaatttatt
gagagtttttagcatgaagagttcttgaattttgtcaaaggccttttctgcatcttttgagataatcatgtggtttctgtctttggttctgtttatatgctggagtacgtttat
tgattttcgtatgttgaaccagccttgcatcccagggatgaagcccacttgatcatggtggataagctttttgatgtgctgctggattcggtttgccagtattttattg
aggatttctgcatcgatgttcatcaaggatattggtctaaaattctctttttttgttgtgtctctgtcaggctttggtatcaggatgatgctggcctataaaatgagtta
gg

Ten occurrences?

**tttttttttttttgagacggagtctcgctctgtcgcccaggctggagtgcagtggcgggat
ctcggctcactgcaagctccgcctcccgggttcacgccattctcctgcctcagcctcc
caagtagctgggactacaggcgcccgccactacgcccggctaatttttgtatttttagt
agagacggggtttcaccgttttagccgggatggtctcgatctcctgacctcgtgatccg
cccgcctcggcctcccaaagtgctgggattacaggcgt**

<u>Length</u>: 277

<u>Occurrences</u> at: 10130003, 11421803, 18695837, 26652515, 42971130, 47398125
In the reversed complement at: 17858493, 41463059, 42431718, 42580925

Using suffix trees: plagiarism

- find longest common substring of strings X and Y
- build Tree(X$Y) and find the deepest node which has a leaf pointing to X and another pointing to Y

Using suffix trees: approximate matching

- edit distance: insertions, deletions, changes
- STOCKHOLM vs TUKHOLMA

String distance/similarity functions

STOCKHOLM vs TUKHOLMA

STOCKHOLM_
_TU_ KHOLMA

=> 2 deletions, 1 insertion, 1 change

Approximate string matching

- A: S T O C K H O L M

  B: T U K H O L M A

- minimum number of ’mutation’ steps:

  a -> b a -> ϵ ϵ -> b …

- $d_{ID}(A,B) = 5$ $d_{Levenshtein}(A,B) = 4$
- mutation costs => probabilistic modeling
- evaluation by dynamic programming => alignment

Dynamic programming

$d_{i,j}$ = min(if $a_i = b_j$ then $d_{i-1,j-1}$ else ∞,
$d_{i-1,j} + 1$,
$d_{i,j-1} + 1$)

= distance between i-prefix of A and j-prefix of B (substitution excluded)

mxn table **d**

$$d_{i,j} = \min(\text{if } a_i = b_j \text{ then } d_{i-1,j-1} \text{ else } \infty,\ d_{i-1,j} + 1,\ d_{i,j-1} + 1)$$

A\B		s	t	o	c	k	h	o	l	m
	0	1	2	3	4	5	6	7	8	9
t	1	2	1	2	3	4	5	6	7	8
u	2	3	2	3	4	5	6	7	8	9
k	3	4	3	4	5	4	5	6	7	8
h	4	5	4	5	6	5	4	5	6	7
o	5	6	5	4	5	6	5	4	5	6
l	6	7	6	5	6	7	6	5	4	5
m	7	8	7	6	7	8	7	6	5	4
a	8	9	8	7	8	9	8	7	6	5

optimal alignment by trace-back

$d_{ID}(A,B)$

Search problem

- find approximate occurrences of pattern P in text T: substrings P’ of T such that d(P,P’) small
- dyn progr with small modification: O(mn)
- lots of (practical) improvement tricks

Index for approximate searching?

- dynamic programming: P x Tree(T) with backtracking

1. Suffix tree
2. Suffix array
3. Some applications
4. **Finding motifs**

Gapped motifs

a##bc

Gapped motifs

a##bc a##bc

ababbbcccbcaaabca

Gapped motifs of T

- gapped pattern: P in (A U {#})*

- <u>gap symbol</u> # matches any symbol in A
- aa##bb#b
- L(P) = occurrence locations of P in T

- P is called a motif of T if |L(P)| > 1 and a motif with quorum q if |L(P)| ≥ q.
- Problem: find occurrence count |L(P)| for all gapped motifs P of T

- $a^n b a^n$ has exponentially many motifs!

Maximal motifs

- motif P´ is the maximal version of motif P if P´ has the largest possible number of non-# symbols among motifs that have the same set of occurrence locations as P

- every motif has unique maximal version

- unfortunately still exponential number of different maximal motifs

Blocks of maximal motifs

- aaa##b##ba has blocks aaa, b, ba

- Lemma: Maximal 1-block motifs ↔ (branching) nodes of Tree(S)

- Thm: Each block of a maximal motif of T is a maximal substring motif of T. Hence there are O(n) different strings that can be used as a block of a maximal motif.

- => There are $O(n^{2k-1})$ different maximal motifs with k blocks [$O(n^{2k})$ unrestricted motifs].

Two-block maximal pattern: X###Y

Counting 2-block maximal motifs

- Thm: The occurrence counts for all maximal motifs with two blocks can be found in (optimal) time $O(n^3)$.
- c.f., Arimura et al (2000), Apostolico et al (2004), …

Algorithm (very simple)

2-block motif *(X,d,Y)*

```
for each maximal substring motif X
    for each distance d = 1,2, …
        mark the leaves of Tree(T) that correspond to
          locations L(X) + d
        for each maximal substring motif Y,
          find the number h(Y) of marked leaves in
          its subtree in Tree(T)
        the occurrence count of motif (X,d,Y) is h(Y)
```

Algorithm (very simple)

2-block motif *(X,d,Y)*

for each maximal substring motif X — ***O(n)***

  for each distance d = 1,2, … — ***O(n)***

    mark the leaves of Tree(T) that correspond to locations L(X) + d

    for each maximal substring motif Y, find the number h(Y) of marked leaves in its subtree in Tree(T) — ***O(n)***

    the occurrence count of motif (X,d,Y) is h(Y)

Counting 2-block maximal motifs (cont)

- Thm: The occurrence counts for all maximal motifs with two blocks can be found in (optimal) time $O(n^3)$.

- flexible gaps:
  x*y * = gap of any length

- Thm: The occurrence counts for all maximal motifs with two blocks and one flexible gap can be found in (optimal) time $O(n^2)$.
- k-block case?

Conclusion

- suffix structures provide very efficient algorithmic tools for finding and learning potentially interesting patterns in strings

General case

- **Q1**: Given q and W, has T a motif with at least W non-gap symbols and at least q occurrences?
- In k-block case, is $O(n^{2k-1})$ (or even better) time possible?
- related work: H.Arimura&al, A. Apostolico &al, M-F. Sagot, L. Parida, N. Pisanti, …

Suffix trees and suffix arrays

Trie

- A tree representing a set of strings.

{
aeef
ad
bbfe
bbfg
c
}

Trie (Cont)

- Assume no string is a prefix of another

Each edge is labeled by a letter, no two edges outgoing from the same node are labeled the same.

Each string corresponds to a leaf.

Compressed Trie

- Compress unary nodes, label edges by strings

Suffix tree

Given a string s a suffix tree of s is a compressed trie of all suffixes of s

To make these suffixes prefix-free we add a special character, say $, at the end of s

Suffix tree (Example)

Let s=abab, a suffix tree of s is a compressed trie of all suffixes of s=abab$

{
$
b$
ab$
bab$
abab$
}

Trivial algorithm to build a Suffix tree

Put the largest suffix in

Put the suffix bab$ in

Put the suffix ab$ in

Put the suffix b$ in

Put the suffix $ in

We will also label each leaf with the starting point of the corres. suffix.

Analysis

Takes $O(n^2)$ time to build.

We will see how to do it in O(n) time

What can we do with it ?

Exact string matching:

Given a Text T, |T| = n, preprocess it such that when a pattern P, |P|=m, arrives you can quickly decide when it occurs in T.

W e may also want to find all occurrences of P in T

Exact string matching

In preprocessing we just build a suffix tree in O(n) time

Given a pattern P = ab we traverse the tree according to the pattern.

If we did not get stuck traversing the pattern then the pattern occurs in the text.

Each leaf in the subtree below the node we reach corresponds to an occurrence.

By traversing this subtree we get all k occurrences in O(n+k) time

Generalized suffix tree

Given a set of strings S a generalized suffix tree of S is a compressed trie of all suffixes of $s \in S$

To make these suffixes prefix-free we add a special char, say $, at the end of s

To associate each suffix with a unique string in S add a different special char to each s

Generalized suffix tree (Example)

Let s_1=abab and s_2=aab here is a generalized suffix tree for s_1 and s_2

{

$	#
b$	b#
ab$	ab#
bab$	aab#
abab$	

}

So what can we do with it ?

Matching a pattern against a database of strings

Longest common substring (of two strings)

Every node with a leaf descendant from string S_1 and a leaf descendant from string S_2 represents a maximal common substring and vice versa.

Find such node with largest "string depth"

Lowest common ancestors (LCA)

A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it

Why?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes

Finding maximal palindromes

Palindromes: Dad, Radar, Malayalam

Q. To find all maximal palindromes in a string s

Let s = cba|aba

The maximal palindrome with center between i-1 and i is the LCP of the suffix at position i of s and the suffix at position m-i+1 of s^r

Maximal palindromes algorithm

Algorithm:

- `r = reverse(s).`
- `m = len(s);`
- `build GST(s,r)`
- `for i=1..m`
- `  find LCA(s[i..],r[m-i+1…])`

Complexity: O(n) time to identify all palindromes

e.g. s = cbaaba$, reverse(s) = abaabc#

Palindrome example

Let s = cbaaba$ then sr = abaabc#

Analysis

O(n) time to identify all palindromes

Drawbacks

- Suffix trees consume a lot of space
- It is O(n) but the constant is quite big
- Notice that if we indeed want to traverse an edge in O(1) time then we need an array of ptrs. of size $|\Sigma|$ in each node

Suffix array

- We loose some of the functionality but we save space.

Let s = abab

Sort the suffixes lexicographically:
ab, abab, b, bab

The suffix array gives the indices of the suffixes in sorted order

3	1	4	2

How do we build it ?

- Build a suffix tree
- Traverse the tree in DFS, lexicographically picking edges outgoing from each node and fill the suffix array.

- O(n) time

How do we search for a pattern ?

- If P occurs in T then all its occurrences are consecutive in the suffix array.
- Do a binary search on the suffix array
- Takes O(mlogn) time

Example

Let S = mississippi

Let P = issa

L →	11	i
	8	ippi
	5	issippi
	2	ississippi
	1	mississippi
M →	10	pi
	9	ppi
	7	sippi
	4	sisippi
	6	ssippi
R →	3	ssissippi

How do we accelerate the search ?

Maintain ℓ = LCP(P,L)

Maintain r = LCP(P,R)

If ℓ = r then start comparing M to P at ℓ + 1

How do we accelerate the search ?

If $\ell > r$ then

Suppose we know LCP(L,M)

If LCP(L,M) < ℓ we go left

If LCP(L,M) > ℓ we go right

If LCP(L,M) = ℓ we start comparing at $\ell + 1$

Analysis of the acceleration

If we do more than a single comparison in an iteration then max(ℓ, r) grows by 1 for each comparison ➔ O(logn + m) time

Suffix Tree

Example: Suffix tree of xabxac

Suffix Tree Representation

S=xabxac

Represent every edge using its start and end text location

Implicit => Explicit

1. No suffix of S is a prefix of a different suffix of S.
2. There is a leaf for each suffix of S.

History

History of linear-time suffix tree

- Weiner's algorithm
  - "The algorithm of 1973" called by Knuth
  - First algorithm of linear time, but much space
- McGreight's algorithm
  - Linear time and quadratic space
  - More readable
- Ukkonen's algorithm
  - On-line algorithm of linear time and less space

Ukkonen

String S$=	S(1)	S(2)	….	S(m-1)	S(m)	$

S$ Prefixes:

Pref(1) = S(1)
Pref(2) = S(1)S(2)
. . .
Pref(i) = S(1)S(2)…S(i)
. . .
Pref(m-1) = S(1)S(2)….S(m-1)
Pref(m) = S(1)S(2)….S(m-1)S(m)
Pref(m+1) = S(1)S(2)….S(m-1)S(m)$ = S$

Ukkonen's insertion order:

Suffixes(Pref(1))
Suffixes(Pref(2))
…
Suffixes(Pref(i))
…
Suffixes(Pref(m-1))
Suffixes(Pref(m))
Suffixes(Pref(m+1))

Implicit suffix tree

The intermediate Ukkonen Suffix Tree will be in the implicit form, until the last prefix insertion, which transform it to the explicit one.

Straightforward Construction

Input: string S[1…m]

```
1. Construct T(1), the Suffix tree of S[1]
2. for ( i = 1 ; i <= m-1 ; i++ )
   {
     //   Convert T to Suffix tree of S[1..i+1]
     for ( j = 1 ; j <= i+1 ; j++ )
     {
          // Find the end of path for S[j…i].
          // Extend the path, if needed, to S[j..i+1].
     }
   }

3. Convert T(m) into the real suffix tree.
```

Time: $O(m^3)$

Extended rule 1

Extending path S[j..i] to S[j..i+1]

Case 1: Path S[j..i] ends at a leaf.

- Extend the string on the last edge by one character S[i+1]
- Constant time

Extended rule 2

Extending path S[j..i] to S[j..i+1]

Case 2: Path S[j..i] has an extension that starts with S[i+1].

- Nothing need to be done, since we are working on the on the implicit suffix tree.
- Also constant time

Extended rule 3

Extending path S[j..i] to S[j..i+1]

Case 3: Path S[j..i] has extensions but none of them start with S[i+1]

- Create a new internal node if needed.
- Add a new edge to a new leaf j

Extended rules (example)

S = axabxb….

Important improvement

Suffix link

- For an internal node v with path-label $x\alpha$, if there is another node $s(v)$ with path-label α, then a pointer from v to $s(v)$ is called a suffix link.

- Same as in Weiner, except the direction of the links
- No need for associating with characters
- Still use and create suffix links during construction

Useful lemmas

Lemma 1: If a new internal node v with path-label xα is added to the current tree in extension j of some phase i + 1, then either the path labeled α already ends at an internal node of the current tree or an internal node at the end of string α will be created (by the extension rules) in extension j + 1 in the same phase i + 1.

Lemma 2: In Ukkonen's algorithm, any newly created internal node will have a suffix link from it by the end of the next extension.

Lemma 3: In any implicit suffix tree T(i), if internal node v has path-label xα, then there is a node s(v) of T(i) with path-label α.

Algorithm using suffix links

Single extension algorithm: extension $j > 2$ of phase $i + 1$

1. Find the first node v at or above the end of S[j -1..i] that either has a suffix link from it or is the root. This requires walking up at most one edge from the end of S[j - 1..i] in the current tree. Let γ (possibly empty) denote the string between v and the end of S[j - 1..i].

2. If v is not the root, traverse the suffix link from v to node s(v) and then walk down from s(v) following the path for string γ. If γ is the root, then follow the path for S[j..i] from the root (as in the naive algorithm).

3. Using the extension rules, ensure that the string S[j..i]S(i + 1) is in the tree.

4. If a new internal node w was created in extension j - 1 (by extension rule 3), then by Lemma 1, string α must end at node s(w), the end node for the suffix link from w. Create the suffix link (w, s(w)) from w to s(w).

Single Extension algorithm (example)

Figure 6.5: Extension j >1 in phase i + 1. Walk up at most one edge (labeled γ) from the end of the path labeled S[j-1 ..i] to node v; then follow the suffix link to s(v); then walk down the path specifying substring γ then apply the appropriate extension rule to insert suffix S[I + 1].

Skip/Count Trick

Improvement for looking γ from the previous process

When the algorithm identifies the next edge on the path, it compares the current value of g to the number of characters g' on that edge.

When g is at least as large as g' the algorithm skips to the node at the end of the edge, sets g to $g - g$, sets h to $h + g'$, and finds the edge whose first character is character h of γ and repeats.

When an edge is reached where g is smaller than or equal to g', then the algorithm skips to character g on the edge and quits, assured that the γ path from s(v) ends on that edge exactly g characters down its label.

The total time to traverse the path is proportional to the number of nodes on it rather than the number of characters on it.

Skip/Count Trick (Example)

Figure 6.6: The skip/count trick. In phase i + I, substring γ has length ten. There is a copy of substring γ of node s(v); it is found three characters down the last edge, after four node skips are executed.

Time Improvement

Lemma 4: Let (v , s(v)) be any suffix link traversed during Ukkonen's algorithm. At that moment, the node-depth of v is at most one greater than the node depth of s(v).

Theorem 1: Using the skip/count trick any phase of Ukkonen's algorithm takes 0(m) time.

Skip iterations trick 1

Observation 1: Once a leaf, always a leaf
If Case 1 applies during a particular (i,j) iteration, it will also apply for all iterations with a larger i and same j.

Proof:
Path S[j..i] ends at a leaf. Extend the string on the last edge by 1 character (S[i+1]). Now the Path S[j..i+1] ends at the same leaf and it will be the same for every extension of it to S[j..i+2] etc.

Skip iterations trick 2

Observation 2: Extensions stopper

If Case 2 applies during a particular (i,j) iteration, it will also apply for all iterations with the same i and larger j.

Proof:

Path S[j..i] has at least one extension that starts with S[i+1]. Since S[j..i+1] is already in the tree, S[j+1..i+1] must also be in the tree.

Skip iterations trick 3

Observation 3: Make a node, be a leaf

If Case 3 applies during a particular (i,j) iteration, Case 1 will apply for all iterations with the a larger i and same j.

Proof:

Path S[j..i] has extensions but none of them start with S[i+1]. Add a new branch to a new leaf labeled j. Now the path S[j..i+1] ends at a leaf, and Case 1 will apply for every extension of it to S[j..i+2] etc.

Possible execution

Extension Case distribution

j=	1	2	3	4	5	6	7	8	9=m
i=1:	1	3							
i=2:	1	1	2						
i=3:	1	1	2	2					
i=4:	1	1	3	2	2				
i=5:	1	1	1	2	2	2			
i=6:	1	1	1	3	3	2	2		
i=7:	1	1	1	1	1	3	2	2	
i=8:	1	1	1	1	1	1	3	2	2

Need to compute:

At most one Case 2 per i

At most one Case 1 or 3 per j

Creating a true suffix tree

- Run another iteration of Ukkonen algorithm on S$
- No suffix is now a prefix of any other suffix.
- As a result, each suffix will end at a leaf.
- Replace each index on every leaf edge with the number m.

Total Algorithm time O(m)

Computational Number Theory

Number-Theoretic Algorithms

Topics:

- **Primes**
- **Modular math**
- **GCD**
- **Extended GCD**
- **Inverse mod**
- **Power mod**
- **Chinese Remainder Theorem**
- **Large Primes**
- **RSA**
- **Discrete Log**

Prime factorization

1. There are infinite number of primes

Primes = { 2, 3, 5, 7, 11, …}

2. Every number can be uniquely written as a product of its primes factors:

Eg. 100 = 4*25 = 2*2*5*5

Homework: primes.c

Write a program **primes.c** to print the first n primes numbers

```
$ primes 100
Prime numbers less than 100:
2 3 5 7 11 13 17 19 23 29 31 37
  41 43 47 53 59 61 67 71 73 79 83
  89 97
```

Homework: write is-prime.c, factor.c

```
$ is-prime 100
100 is not a prime
$ is-prime 229
229 is a prime

$ factor 100
100: 2 2 5 5
$ factor 101
101: 101
```

Modular math

In C,we write: 5 % 3 == 2, read as 5 mod 3 is 2.

In Math, we write: $(5 \equiv 2) \bmod 3$, or $5 \equiv_3 2$

5 mod 3 = 2

3 * 3 mod 5 = 4

3 * 3 * 3 mod 5 = 2

(3 * 3 mod 5) * 3 mod 5 = 2

(3 ^ 4) mod 5 = 1 = ((3^2) mod 5)^2 mod 5) = 1

= (9 mod5)^2 mod 5 = 1

GCD (Greatest Common Divisor)

Examples:

1. gcd(10,15)=gcd(2*5, 3*5) = 5 * gcd(2, 3) = 5
2. gcd(20,24)=gcd(2*2*5, 2*2*2*3) = 2*2*gcd(5,2*3) = 4
3. gcd(2,2)=2

Numbers are called **relatively prime** if their gcd(a,b) is 1.

E.g. gcd(9,8)=1

Any two prime numbers are relatively prime.

E.g. gcd(13,17)=1

Properties of gcd

$\gcd(a,0)=a$

$\gcd(a,ka)=a$

$\gcd(a,b)=\gcd(b,a)$

$\gcd(a,-b)=\gcd(a,b)$

$\gcd(ka,kb) = k \gcd(a,b)$

$\gcd(a, b) = \gcd(b, a \bmod b)$

$\gcd(\gcd(a,b),c) = \gcd(a,\gcd(b,c)) = \gcd(a,b,c)$.

Euclid's GCD Algorithm

To compute gcd(a,b) where $a \geq 0$ and $b \geq 0$.
We exploit the following property of gcd to write an recursive an algorithm:

gcd(a, b) = gcd(b, a mod b)

```
int gcd(a, b){
     if (b = 0)
       return a
     else
       return gcd(b, a mod b)
}
```

GCD Algorithm Example

Example:

gcd(30, 21) = gcd(21, 9) // 9 is 30 mod 21
= gcd(9, 3) // 3 is 21 mod 9
= gcd(3, 0) // 0 is 9 mod 3
= 3

Modular math mod n

In modular arithmetic the set of congruence classes relatively prime to the modulus number n, form a group under multiplication called the multiplicative group of integers modulo *n*.

It is also called the group of primitive residue classes modulo *n*.

Since $\gcd(a, n) = 1$ and $\gcd(b, n) = 1$ implies $\gcd(ab, n) = 1$, the set of classes relatively prime to *n* is closed under multiplication.

Prime, mod 5 multiplicative group

%5	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

Composite, mod 6

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

Mod n group examples:

Extended Euclid

ext-gcd(a,b) = [g, x, y]

Where:

g = gcd(a,b) and

g = a*x + b*y, for some x, y.

Example:

gcd(30, 21) = 3 = 30*5 + 21*(-7).

ext-gcd(30,21) is [3, 5, -7]

Compute gcd(181, 39)

181 = 39 * 4 + 25		(25 = 181 – 39*4).
39 = 25 * 1 + 14		(14 = 39 – 25)
25 = 14 * 1 + 11		(11 = 25 – 14)
14 = 11 * 1 + 3		(3 = 14 – 11)
11 = 3 * 3 + 2		(2 = 11 – 3*3)
3 = 2 * 1 + 1		.. Hence gcd(181,39) is 1.
2 = 2 * 1 + 0		

Now use back-substitution to find x, y such that:

181 * x+39 * y = 1.

Use back-substitution to find x and y:

181*x+39*y = gcd 1.

gcd 1 = 3 – 2

= 3 - (11 - 3 * 3)

= 4 * 3 – 11

= 4 * (14 - 11) - 11 = 4 * 14 - 5 * 11

= 4 * 14 - 5 * (25 - 14) = 9 * 14 - 5 * 25

= 9 * (39 - 25) - 5 * 25 = 9 * 39 - 14 * 25

= 9 * 39 – 14 * (181 - 39 * 4) = 65 * 39 - 14 * 181

Hence gcd 1 = 65 * 39 - 14 * 181

x = 65, y = 14.

Exercise on computing gcd

Find the gcd(55, 700) and write the gcd as an integer combination of 55 and 700.

Solution to gcd(55, 700)

700 = 55 * 12 + 40

55 = 40 * 1 + 15

40 = 15 * 2 + 10

15 = 10 * 1 + 5 .. hence gcd(55,700)=5

10 = 5 * 2 + 0

Solution: back substituting to find x, y

Back substituting, we get

$5 = 15 - 10$

$= 15 - (40 - 15 * 2)$

$= 3 * 15 - 40$

$= 3(55 - 40) - 40$

$= 3 * 55 - 4 * 40$

$= 3 * 55 - 4(700 - 55 * 12)$

$= 51 * 55 - 4 * 700 = x * 55 + y * 700$

Hence $\gcd(55,700) = 5 = 51 * 55 - 4 * 700$.

Extended Euclid Algorithm

```
Extended-Euclid (a, b)
    if b = 0 then
        return(a, 1, 0)
    fi;
    (d′, x′, y′) := Extended-Euclid (b, a mod b);
    (d, x, y) := (d′, y′, x′ – ⌊a/b⌋ y′);
    return (d, x, y)
```

$\gcd(a, 0) = a = 1 \cdot a + 0 \cdot b$

$d' = bx' + (a \bmod b)\, y'$

$d = bx' + (a - \lfloor a/b \rfloor b)\, y'$

$= ay' + b\,(x' - \lfloor a/b \rfloor y')$

$= ax + by$

Example: ext-gcd(30,21)=(3,-2,3)

```
C:\> eea 30 21
 Calculating eea_rec(30,21):
  Calculating eea_rec(21, 9):
  Calculating eea_rec( 9, 3):
  Calculating eea_rec( 3, 0):
   1 *  3 +  0 *  0 =  3 = gcd( 3, 0),
   0 *  9 +  1 *  3 =  3 = gcd( 9, 3),
   1 * 21 + -2 *  9 =  3 = gcd(21, 9),
  -2 * 30 +  3 * 21 =  3 = gcd(30,21),
```

Exercises

1. Compute eea(7, 3)
2. Compute eea(31, 20)

Solution

Q. Compute eea(7,3)

A. Calculating eea_rec(7, 3):

Calculating eea_rec(3, 1):

Calculating eea_rec(1, 0):

1 * 1 + 0 * 0 = 1 = gcd(1, 0)

0 * 3 + 1 * 1 = 1 = gcd(3, 1)

1 * 7 + -2 * 3 = 1 = gcd(7, 3)

Solution

Q. Compute eea(31, 20)

A. Calculating eea_rec(31,20):

Calculating eea_rec(20,11):

Calculating eea_rec(11, 9):

Calculating eea_rec(9, 2):

Calculating eea_rec(2, 1):

Calculating eea_rec(1, 0):

1 * 1 + 0 * 0 = 1 = gcd(1, 0), inv(1) is 1 mod 0

0 * 2 + 1 * 1 = 1 = gcd(2, 1), inv(2) is 0 mod 1

1 * 9 + -4 * 2 = 1 = gcd(9, 2), inv(9) is 1 mod 2

-4 * 11 + 5 * 9 = 1 = gcd(11, 9), inv(11) is -4 mod 9

5 * 20 + -9 * 11 = 1 = gcd(20,11), inv(20) is 5 mod 11

-9 * 31 + 14 * 20 = 1 = gcd(31,20), inv(31) is -9 mod 20

Inverse mod n

$a * a^{-1} = 1 \bmod n$

If a and b are inverses mod n:

$a * b = 1 \bmod n$

$1/a = b \bmod n$

$1/b = a \bmod n$

Example: $5 * 2 = 10 = 1 \bmod 9$

5 is inverse of 2 mod 9.

Inverse modulo primes

All non-zero b have an **inverse** (mod prime p) where b is not a multiple of p.

For example:

1 * 1 = 1 mod 5

2 * 3 = 1 mod 5

3 * 2 = 1 mod 5

4 * 4 = 1 mod 5

Example, inverse mod 7

The multiplicative inverse of 1 is 1,

the multiplicative inverse of 2 is 4 (and vice versa)

the multiplicative inverse of 3 is 5 (and vice versa)

and the multiplicative inverse of 6 is 6.

Note that 0 doesn't have a multiplicative inverse. This corresponds to the fact in ordinary arithmetic that 1 divided by 0 does not have an answer.

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

Computing inverse mod n

Given a and n, with $\gcd(a, n) = 1$,

Finding x satisfying $ax \equiv 1 \pmod{n}$

is the same as solving for x in $ax + ny = 1$.

We can use ext-gcd eea(a,n) = (g,x,y), to get x.

Not all numbers have inverse mod n

if gcd(a,n)=1, then a has an inverse mod n

The inverse can be found from ext-gcd(a,n):

```
Let gcd(a,n)= 1
a * k + n * y = 1
a * k + 0     = 1, mod n
a * k         = 1, mod n
```

Inverse from ext-gcd

```
C:\> ext-euclid2.sh 2 5
Computing ext_euclid_algo(2, 5)
# 2 = 0 * 5 + 2
# 5 = 2 * 2 + 1
# 2 = 2 * 1 + 0
  res: 2 * -2 + 5 * 1 = 1
  coprime: 2 and 5,
   so  2 * 3 = 1 (mod 5),
   and 3 is the inverse of 2 (mod 5)
```

Power mod, compute a^p mod n

Brute force:

```
int z = 1;
for(i=1;i<=p;i++)
  z = z * a;
z = z % n
```

Problems:

1. z overflows
2. Slow.

Power mod, compute a^p mod n

Brute force:

```
int z = 1;
for(i=1;i<=p;i++)
  z = z * a mod n;
```

Problems:

1. Slow.

Power mod, compute a^p mod n

Brute force:

```
int z = 1;
for(i=1;i<=p;i++)
  z = z * a mod n;
```

Problems:

1. Slow.

Fast Power mod O(log(p))

```
// Return: b^p mod n, complexity O(log2(p)).
// Usage: pow-mod(2,100, 5) for 2^100 mod 5
int pow-mod(int b, int p, int n){
  int r = 1;
  while (p > 0) {
    if (p & 1) { // same as p % 2.
      r = (r * b) % n;
    }
    p >> = 1; // same as: p = p/2
    b = (b * b) % n;
  }
  return r;
}
```

CRT (chinese remainder theorem)

Problem: solve for x in

$x = a1 \bmod m1$

$x = a2 \bmod m2$

$x = a3 \bmod m3$

…

Where m1, m2, m3,… are mutually relatively prime, ie. gcd(m1,m2)=1, gcd(m1,m3)=1,…

CRT

Compute M = m1 * m2 * m3 …

Compute Mj = M/mj

Note: Mj is an integer and gcd(Mj, mj)=1

Compute inverses bj such that:

Mj * bj = 1 mod mj,

Mj * bj = 0 mod mi (i!=j).

Compute x = sum_{j=1..}(Mj * bj * aj) mod M.

CRT exercise

From Niven and Zuckerman Number theory book:

Example 1, find the least positive integer x:

$x = 5 \pmod{7}$,

$x = 7 \pmod{11}$,

$x = 3 \pmod{13}$.

CRT solution

$x = 5 \pmod 7$, $x = 7 \pmod{11}$, $x = 3 \pmod{13}$.

So $a1=5$, $a2=7$, $a3=3$; $m1=7$, $m2=11$, $m3=13$

$M = 7*11*13=1001$

$M1 = M/m1 = 1001/7=143$

$M2 = M/m2 = 1001/11=91$

$M3 = M/m3 = 1001/13=77$

$b1 = inv(M1) \bmod m1 = inv(143) \bmod 7=5= -2$

$b2 = inv(91) \bmod 11 = 4$

$b3 = inv(77) \bmod 13 = 12= -1$

$x = M1*b1*a1+M2*b2*a2+M3*b3*a3$

$x = 143*(-2)*5 + 91*4*(7)+77*(-1)*3 = 887$, also

$x = (143*5*5 + 91*4*7 + 77*12*3) \% 1001 = 887$

CRT program

```
M=1001
  m0. mi= 7, Mi=     143 =M/(mi= 7), bi=    -2 =inv(Mi=   143) mod (mi=7).
  m1. mi=11, Mi=      91 =M/(mi=11), bi=     4 =inv(Mi=    91) mod (mi=11).
  m2. mi=13, Mi=      77 =M/(mi=13), bi=    -1 =inv(Mi=    77) mod (mi=13).

test 0.
crtin=887
  887 mod 7 is 5
  887 mod 11 is 7
  887 mod 13 is 3
  crt_inv:+(5*-2*143)+(7*4*91)+(3*-1*77)= 887
```

Euler phi function

phi(n) = number of numbers relatively prime to n.

phi(6)=2 … {1,5}

phi(5)=4 … {1,2,3,4}

phi(p) = p-1, if p is a prime.

Perfect number

A **perfect number** is a positive integer that is equal to the sum of its proper positive divisors.

Eg. 6 = 1+2+3, and

28 = 1 + 2 + 4 + 7 + 14.

Primality testing

Primality testing is easier than factoring.

Factoring is hard.

Worse case n will have 2 factors near sqrt(n),
because n = sqrt(n) * sqrt(n),

Brute force will try to divide n by each factor,
from 1 to sqrt(n).

Is Prime, naive algorithm

```
IsPrime(n) {
  for f in 2..sqrt(n) {
    if (n mod f == 0) // f divides n?
      return false; // n is f * p/f
  }
  return true; // n is a prime.
}
```

Complexity: O(sqrt(n))

Euler and Fermat's theorem

Euler's theorem: a ^ phi(n) =1 mod n.

Proof from Lagrange's theorem about groups.

Fermat's theorem: If p is a prime, we have phi(p) = p-1, and so: a^(p-1) = 1 mod p..

So if a number passes Fermat's test, it is likely to be a prime.

Pseudo-prime

```
Pseudo-prime(n)
   if( 2^(n-1) != 1 mod n )
      return composite; // sure.
  else // pseudo prime.
    return almost prime
```

Exceptions: Carmichael numbers, rare composite numbers that pass above test.

Rabin Miller primality test

```
Rabin-miller-pseudo-prime(n){
   for j = 1 .. s {
     a = random(1..n-1)
     if( a^(n-1) != 1 mod n ) {
       also detect squareroots of 1.
       return composite;
     }
   }
   return almost prime;
   // error rate is 1/2^s
}
```

Factoring with Pollard Rho heuristic

Inputs: n, the integer to be factored; and $f(x)$, a pseudo-random function modulo n

Output: a non-trivial factor of n, or failure.

$x \leftarrow 2$, $y \leftarrow 2$; $d \leftarrow 1$

While $d = 1$:

$x \leftarrow f(x)$

$y \leftarrow f(f(y))$

$d \leftarrow \mathrm{GCD}(|x - y|, n)$

If $d = n$, return failure.

Else, return d.

Pollard Rho

If the function prints a factor, it is correct.

If it doesn't print anything, it may be still searching.

Instead of sqrt(n) complexity, pollard-rho one has complexity sqrt(sqrt(n)).

RSA steps

1. **Choose two large primes, p and q, kept secret.**
2. **Compute n = p*q, assume factoring n is hard.**
3. **Compute $\varphi(n) = (p - 1)(q - 1)$**
4. **Pick prime e, such that $\gcd(e,\varphi(n)) = 1$.**
5. **Determine $d = e^{-1} \pmod{\varphi(n)}$**
6. **Publish *public-key* (e,n)**
7. **Hide *private-key* (d, n)**

RSA Encrypt and Decrypt

1. Encrypt plaintext=m using public-key to:

   Ciphertext = $c = m^e \pmod{n}$.

2. Decrypt c using private key: $m = c^d \pmod{n}$.

RSA example

1. **Choose two large prime, p = 61 and q = 53.**
2. **n = p * q = 61 * 53 = 3233**
3. **φ(p * q) = (61 − 1)*(53 − 1) = 3120.**
4. **Let e = 17, gcd(e, n)=1.**
5. **Compute d=2753 (17*d=1 mod φ(3233))**
6. **Hard without knowing factors of n.**
7. **public key is (n = 3233, e = 17),**
   encryption function is m^{17}(mod 3233).
8. **private key is (n = 3233, d = 2753)**
   decryption function is c^{2753}(mod 3233).

RSA encrypt/decrypt:

1. Encrypt ‘A’ = 65, using (e=17,n=3233)

   $c = 65^{17} (\text{mod } 3233) = 2790.$

2. Decrypt 2790, using (d=2753,n=3233)

   $m = 2790^{2753} (\text{mod } 3233) = 65$, to get ‘A’.

Discrete Logarithms (dlog) Definition

Discrete logarithms are logarithms defined with regard to multiplicative cyclic groups.

If *G* is a multiplicative cyclic group and *g* is a generator of *G*, then from the definition of cyclic groups, we know every element *h* in *G* can be written as:

$g^x = h$, for some x in G.

$x = \text{dlog}_g(h)$

dlog example

In the group is Z5*, 2 is a generator

2^4 ≡ 1 mod 5, so

$\mathrm{dlog}_2(1) = 4$ in Z5.

Diffie hellman

Diffie hellman is based on the principle that computing dlog is hard.

ie. It is hard to compute a, from g and g^a.

Diffie Hellman

$$K = A^b \bmod p = (g^a \bmod p)^b \bmod p = g^{ab} \bmod p = (g^b \bmod p)^a \bmod p = B^a \bmod p$$

Diffie Hellman: shared-secret over insecure network

Alice and Bob want to come up with a shared-secret password.
Alice and Bob agree on: g and modulus p (hacker Catie sees g, p).

Alice picks secret: a.	Bob picks secret: b.
Alice computes: $A \equiv_p g^a$	Bob computes: $B \equiv_p g^b$
Alice sends A to Bob.	Bob sends B to Alice.

Alice computes: $K \equiv_p B^a = (g^b)^a$
Bob computes: $K \equiv_p A^b = (g^a)^b$

Alice and Bob use K as their common shared-secret password.
Catie knows A and B, g, p but cannot compute a, b or K.

Diffie Hellman example

Diffie Hellman example

Alice and Bob choose: p=23 (prime), g=11 (generator).

Alice choses secret a=6, computes A=g^a=11^6=9 (mod 23)

Bob choses secret b=5. computes B=g^b=11^5=5 (mod 23).

Alice sends Bob: A=9

Bob sends Alice: B=5.

Alice computes: K=B^a=5^6=8 (mod 23)

Bob computes: K=A^b=9^5=8 (mod 23).

Applications of

Data Encryption

- SSL
- RSA
- Diffie Hellman

Protecting data with Encryption

- We need to lock the data
- Make sure it is not tampered
- Make sure it available only to the recipients.

Encryption

Simple explanation of encryption and decryption methods

Encryption before computers

German Lorenz cipher machine, used in World War II to encrypt very-high-level general staff messages

Symmetric key encryption

- Symmetric key encryption uses one key, called secret key - for both encryption and decryption. Users exchanging data must keep this key secret. Message encrypted with a secret key can be decrypted only with the **same** secret key.
- Examples: DES - 64 bits, 3DES - 192 bits, AES - 256 bits, IDEA - 128 bits, Blowfish, Serpent

Symmetric vs Public key encryption

Public-Key
ZU
ZU
AUF
Private-Key
Public
Private
YALE

Public key cryptography

Public key Cryptography

- The two main branches of public key cryptography are:

- Public key encryption: a message encrypted with a recipient's public key cannot be decrypted by anyone except by a matching private key. This is used for confidentiality.

- Digital signatures: a message signed with a sender's private key can be verified by anyone using sender's public key, thereby verifying the sender and message is untampered.

Mailbox and wax seal analogy

- An analogy to public-key encryption is that of a locked mailbox with a mail slot. The mail slot is exposed and accessible to the public; its location (the street address) is in essence the public key. Anyone knowing the street address can go to the door and drop a written message through the slot; however, only the person who possesses the key can open the mailbox and read the message.

- An analogy for digital signatures is the sealing of an envelope with a personal wax seal. The message can be opened by anyone, but the presence of the seal authenticates the sender.

Web of trust

- A central problem for use of public-key cryptography is confidence (ideally proof) that a public key is correct, belongs to the person or entity claimed (i.e., is 'authentic'), and has not been tampered with or replaced by a malicious third party. The usual approach to this problem is to use a public-key infrastructure (PKI), in which one or more third parties, known as certificate authorities, certify ownership of key pairs. Another approach, used by PGP, is the "web of trust" method to ensure authenticity of key pairs.

Alice makes a public key for locking

An unpredictable (typically large and random) number is used to begin generation of an acceptable pair of keys suitable for use by an asymmetric key algorithm.

PK Encryption: Anyone can lock, only Alice can unlock

In an asymmetric key encryption scheme, anyone can encrypt messages using the public key, but only the holder of the paired private key can decrypt. Security depends on the secrecy of that private key.

PK Signing: Only Alice can lock anyone can unlock

In some related signature schemes, the private key is used to sign a message; but anyone can check the signature using the public key. Validity depends on private key security.

Key exchange over insecure network

Merkle Hellman and Diffie

Diffie Hellman key exchange

Diffie Hellman overview

In the Diffie–Hellman key exchange scheme, each party generates a public/private key pair and distributes the public key. After obtaining an authentic copy of each other's public keys, Alice and Bob can compute a shared secret offline. The shared secret can be used as the key for a symmetric cipher.

Diffie Hellman overview

Diffie Hellman example 1

Diffie Hellman example 2

The RSA Cryptosystem

Clifford Cocks
(Born 1950)

- A public key cryptosystem, now known as the RSA system was introduced in 1976 by three researchers at MIT.

Ronald Rivest
(Born 1948)

Adi Shamir
(Born 1952)

Leonard
Adelman
(Born 1945)

- It is now known that the method was discovered earlier by Clifford Cocks, working secretly for the UK government.
- The public encryption key is (n,e), where $n = pq$ (the modulus) is the product of two large (200 digits) primes p and q, and an exponent e that is relatively prime to $(p-1)(q-1)$. The two large primes can be quickly found using probabilistic primality tests, discussed earlier. But $n = pq$, with approximately 400 digits, cannot be factored in a reasonable length of time.

HTTPS security guarantees

from eff.org

- **Server authentication** allows the browser and the user to have some confidence that they are talking to the true application server. Without this guarantee, there can be no guarantee of confidentiality or integrity.
- **Data confidentiality** means that eavesdroppers cannot understand the communications between the user's browser and the web server, because the data is encrypted.
- **Data integrity** means that a network attacker cannot damage or alter the content of the communications between the user's browser and the web server, because they are validated with a cryptographic *message authentication code*.

Certificate

Digital Signatures

Cryptographic hash functions

- One way functions - given output, very hard to compute input.

- Hard to find another input that gives same output.

- Hard to find 2 different inputs with same output.

•Examples:
MD5 128 bits
SHA 160
SHA2 256,512

A cryptographic hash function (specifically, SHA-1) at work. Note that even small changes in the source input (here in the word "over") drastically change the resulting output, by the so-called avalanche effect.

Signing using SHA and RSA

Digital Signatures

Sign
Document for signing
Hash calculation
1AFFF12CF
Document Fingerprint
The signer's PKI objects
Signer
Private Key
Public Key
Certificate
SIGN:
Encrypt fingerprint
using signer's
Private key
1AFFF12CF
Signed Fingerprint
Public Key
Certificate
Digital Signature object can be embedded inside the document
Digitally Signed Document
Verify
Current Document
Digital Signature
1AFFF12CF
Calculated Fingerprint
?
Match Results
1AFFF12CF
Original Fingerprint
Invalid Signature
Valid Signature
copyright © 2008 ARX (www.arx.com) All Rights Reserved

SSL

- SSL is secure socket layer
- Used by https
- to secure e-commerce and logins

https website

SBI SSL certificate

https: browser warning on fake ssl certificate

It not possible to verify that you are communicating with **www.facebook.com** instead of an attacker who generated his own certificate claiming to be **www.facebook.com**. You should not proceed past this point.

Firewall proxy claiming to be facebook over ssl

Good ssl certificate, with security flaws

Number theory application

GPG

GPG (Gnu privacy guard)
PGP (pretty good privacy)

- GPG
- C:\> gpg --version
- C:\> gpg –gen-key
  - » gpg: key 43F2B829 marked as ultimately trusted
  - » public and secret key created and signed.
- ~/.gnupg

GPG

- gpg --export --armor > Public-key.asc
- gpg --import file.asc
- gpg --sign-key RedHat
- gpg –list-keys

See http://www.gnupg.org/gph/en/manual.html for more help.

GPG usage

- Generate a private key:

```
gpg --gen-key
```

- Get your public key as ascii text:

```
gpg --armor --output pubkey.txt –export
 you@gmail.com
```

GPG usage

Send your keys to a key-server

```
gpg --send-keys youremail –keyserver
  hkp://subkeys.pgp.net
```

Import Friend's key

gpg --import friend.asc OR

**gpg --search-keys friend@gmail.com **

--keyserver hkp://subkeys.pgp.net

GPG usage

Encrypt message.txt for your friend

Check you have his/her key, else get it:

$ **gpg –list-keys | grep friend**

…

```
$ gpg --encrypt –recipient \
  friend@gmail.com message.txt
```

Reading mail from your friend

```
$ gpg --decrypt reply.txt
```

GPG usage

Signing a file

```
$ gpg --armor --detach-sign
  myfile.zip
```

Verify the signature

```
$ gpg --verify myfile.asc myfile.zip
```

Keyserver: http://pgp.mit.edu/

MIT PGP Public Key Server

Help: Extracting keys / Submitting keys / Email interface / About this server / FAQ
Related Info: Information about PGP / MIT distribution site for PGP

Extract a key

Search String: Do the search!

Index: Verbose Index:

Show PGP fingerprints for keys

Only return exact matches

Submit a key

Enter ASCII-armored PGP key here:

Clear Submit this key to the keyserver!

Homework

- Read http://www.pgpi.org/doc/pgpintro/
- Install gpg (if you don’t have it)
- Create your public key
- Upload your key to mit keyserver
- Sign your friends public key
- Send a signed/encrypted test email to a friend.
- Decrypt and verify signature in the email you receive

Primes

```
$ OpenSSL
>  prime 17 (is 17 a prime?)
   11 is prime
  (Hex 11 = 16 + 1 = 17 Dec).
$ gpg --gen-prime 1 30
    # 30 bit prime.
     3756E197 # Hex
     # Decimal 928440727
```

Factoring

Using cygwin on windows or linux:

```
$ factor 111111111111111
  3 31 37 41 271 2906161
$ factor 928440727
  928440727
$ factor 928440729
  3 3 337 443 691
```

Factoring with OpenSSL

```
$ factor 928440727
   928440727: 928440727
$ factor 928440729
   928440729: 3 3 337 443 691
```

OpenSSL to generate public key

Generate a new public/private keypair in openssl

```
$ openssl genrsa -out key.pem
   Generating RSA private key, 512 bit long
  modulus
    ..+++++++++++.........++++++++++++
    e is 65537 (0x10001)
```

Extract the modulus, e, primes from your key:

```
$ openssl rsa -in key.pem -noout -text
  publicExponent: 65537 (0x10001)
  Modulus=....long-string-of-hex-digits...
```

Random numbers

- PRNG are predictable, /dev/urandom
  - Use a fixed function and time to generate random numbers.
  - Example: r = md5(time + hostname + process id)
- RNG /dev/random, uses system entropy
  - Use physical system to generate random number
  - Example: r = md5(mouse and keyboard delays)

Brute force cracking passwords

WORD,	SIZE,	BITS,	CRACK-TIME
1 Single word,	8 char,	24-bits,	Seconds
2 Random [a-z]	8 char,	37-bits,	Minutes
3 Random [a-z]	16 char,	75-bits,	Decades

Completely random printable string

SIZE,	BITS,	CRACK-TIME
• 6 char,	40-bits,	Minutes
• 8 char,	52-bits,	Hours
• 12 char,	78-bits,	Decades
• 15 char,	97-bits,	Centuries
• 20 char,	130-bits,	Un-crackable

Homework, number theory

Homework

Write a program **primes.c** to print the first n primes numbers

```
$ primes 100
Prime numbers less than 100:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
  53 59 61 67 71 73 79 83 89 97
```

Print primes, sieve of Erosthenes algorithm

```
Print_Primes( n ) {
   int composite[n]
   for( i = 2 ; i < n; i++ )
      composite[i] = 0;
   for( i = 2 ; i < n; i++ ){
     if( composite[i] )
         continue;
     print i
     for(k=i*2; k  < n; k += i)
        composite[k] = 1;
   }
}
```

Homework: Write is-prime.c, factor.c

```
$ is-prime 100
100 is not a prime

$ is-prime 101
101 is a prime

$ factor 100
100: 2 2 5 5

$ factor 101
101: 101
```

Is Prime, naive algorithm

```
is_prime(n) {
  for f in 2 to sqrt(n) {
    if (n mod f == 0)
      return "n is composite, factor = f"
  }
  return "n is a prime, no factors found"
}
```

Homework: Write Euclid's GCD, Ext-GCD, Modular Inverse in C

```
def gcd a, b:
        if (b == 0): return a
        else:  return gcd(b, a mod b)

def egcd(a, b):
    if a == 0: return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def mod_inv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:  Error, no modular inverse
    else: return x % m
```

Code fast power mod in C

```
// Return: b^p mod n
// Usage: pow-mod(2,100, 5) for 2^100 mod 5
pow_mod(int b, int p, int n)
  int result = 1
  while (p > 0) {
    if (p % 2)
      result = (result * b) % n
    p /= 2
    b = (b * b) % n
  }
  return result
```

Homework, compute 1000! exactly in C

1. Store bigint as an array of digits
2. Start with

```
void fact(int n)
   bigint f = 1;
   for(i = 1; i<=1000; i++)
      mult(f, i, f)
   print(f);
void mult(bigint *a, int i) { ??? }
void print(bigint *n)  /* homework */
```

1. 1000!

```
/*
What: compute n! exactly using bigint.
Date: 2013-03-22
By moshahmed/at/gmail
Usage:
  $ gcc -g -Wall factorial-bigint.c –lm -o fact
  $ fact 1000
  1000!= 402387260077093773543702….000
*/
```

2. 1000!

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 10000
typedef struct {  char digits[N]; } bigint;
   // To make it easy to extend a bigint
   // on right side during multiplication.
   // we store bigint value as reverse(digits),
   // so digits="321" => value=123.
```

3. 1000!

```
void print(bigint *b) {
  int len=strlen(b->digits);
  while(len>=0)
        printf("%c", b->digits[len--]);
  printf("\n");
}
```

4. 1000!

```
void mult(bigint *a, int b) {
  int carry=0, len = strlen(a->digits), k;
  // HOME WORK
  // Sample computation to do:
  // Input: a = "027\0", b=7
  // means: compute: 720 * 7
  // output: a = 5040
}
```

Homework, algorithm for bigint multiply

```
mult( bigint *a, int b) // Do: a = a  * b
   int carry=0
   for( i= a->len; i>0; i--)
        x = b * a->digit[i] + carry
        b->digit[i] = x % 10
        carry = x / 10
   while carry > 0 do
       b->digit[i++] = carry % 10
       carry /= 10
```

Main function to compute 1000!

```
int main(int argc, char *argv[]){
  int i, input = 1000;
  bigint fact;
  fact.digits[0]='1';
  fact.digits[1] = '\0'; // initialize fact=1
  if ( argc > 1 ) input = atoi(argv[1]);
  for(i=1;i<=input;i++)
    mult(&fact, i);
  printf("%d!=",input); print(&fact);
  return 0;
}
```

Homework, C code for bigint arithmetic functions

Write funtions to

1. bigint *add(bigint *a, bigint *b):
   // return result = a + b
2. bigint * mult(bigint *a, bigint *b)
   // return result = a *b; multiply a digit at a time.
3. minus(bigint *a, bigint *b)
   // return result = a – b; subtract using borrow
4. bigint *random( ) // return a random bigint
5. print(bigint *a)
6. Use these to multiply 2 random hundred digit numbers, and print the result.

Verify this Diffie Hellman calculation

Alice and Bob choose: p=23 (prime), g=11 (generator).
Alice choses secret a=6, computes A=g^a=11^6=9 (mod 23)
Bob choses secret b=5. computes B=g^b=11^5=5 (mod 23).
Alice sends Bob: A=9
Bob sends Alice: B=5.
Alice computes: K=B^a=5^6=8 (mod 23)
Bob computes: K=A^b=9^5=8 (mod 23).

Repeat above calculations and compute K,
i.e. "(g^a)^b mod p"
"(g^b)^a mod p"
where p=101, g=7, a=4, b=12,

Solution using Pari and Maple

Q. Compute K, with g=7, a=4, b=12, p=101

i.e. "(g^a)^b mod p" and "(g^b)^a mod p"

```
c:\> pari
gp > lift(Mod((7^4)^12, 101))
68
gp > lift(Mod((7^12)^4, 101))
68
c:\> maple
(7**4)**12 mod 101;
68
(7**12)**4 mod 101;
68
```

Verify this RSA calculation using Maple / Pari / bigint / math package

1. Choose two large prime, p = 61 and q = 53.
2. n = p * q = 61 * 53 = 3233
3. phi = φ(p * q) = (61 – 1)*(53 – 1) = φ(3233) = 3120
4. Let e = 17, gcd(e, n)=1.
5. Compute d=2753 = inv(17, 3120) (inverse modulus n).

   i.e. (d* 17 = 1 mod 3120)

1. Hard to find d, without knowing factors of n.
2. public key is (n = 3233, e = 17),

   encryption function is m^{17}(mod 3233).
3. private key is (n = 3233, d = 2753)

   decryption function is c^{2753}(mod 3233).

Repeat above: e=23, p=11, q=13

ie. n = p*q, phi=(p-1)*(q-1), d = inv(e,phi)

Solution for RSA keys using Pari, Mathematica and Maple

Given: e=23, p=11, q=13, compute RSA keys:

n = p*q = 11*13=143

phi=(p-1)*(q-1)=10*12=120

d = inv(e,n)=inv(23,120) = 47

```
c:\> pari
  gp> bezout(23,120)[1]
  47
c:\> mathematica
  in[1] PowerMod[23, -1, 120]
  out[1] 47
c:\> maple
  23^(-1) mod 120
  47
```

public key(n, e) = (143, 23)

private key(n, d) = (143, 47)

RSA encrypt/decrypt character 'C'

1. Encrypt '*A*' = 65, using (*e*=17,*n*=3233)

   $c = 65^{17} (\text{mod } 3233) = 2790.$

2. Decrypt 2790, using (d=2753,n=3233)

   $m = 2790^{2753} (\text{mod } 3233) = 65$, to get '*A*'.

3. Repeat above, encrypt '*C*' and decrypt it. Show all the numbers.

Compute RSA encrypt/decrypt:

Solution: Using the ascii code of 'C' = 67

c = 67^e mod n = 67^23 mod 143 = 111

d = 111^d mod n = 111^47 mod 143 =67='C'

```
c:\> pari
  gp> lift(Mod(67^23, 143)) = 111
  gp> lift(Mod(111^47, 143) =  67
c:\> maple
  > 67^23 mod 143;
  111
  > 111^47 mod 143
```

Computational Geometry

References: Cormen, Shamos, Berge.

Applications

- Graphics, Video
- GIS, maps, layered maps
- Transportation
- CAD/CAM (computer aided design/manufacturing).
- Robotics, motion planning
- Routing VLSI chips
- Molecular chemistry

Geometry

Plane Geometry: the geometry that deals with figures in a two-dimensional PLANE.

Solid Geometry: the geometry that deals with figures in three-dimensional space.

Spherical Geometry: the geometry that deals with figures on the surface of a sphere.

Euclidean Geometry: the geometry (plane and solid) based on Euclid's postulates.

Non-Euclidean Geometry: any geometry that changes Euclid's postulates.

Analytic Geometry: the geometry that deals with the relation between ALGEGRA and geometry, using GRAPHS and EQUATIONS of lines, curves, and surfaces to develop and prove relationships.

Non-Euclidean Geometry

Models for hyperbolic geometry

Models for elliptic geometry

Points, Lines, Vectors

- We will use x-y plane coordinates for points.
- p1 = (x1, y1), p0 = (x0, y0), etc..
- Lines segments: have start and end points
- Vectors: have direction and length (magnitude).
- |v| is the magnitude of vector v.

Basic Geometric Objects in the Plane

point : denoted by a pair of coordinates (x,y)

segment : portion of a straight line between two points

polygon : circular sequence of points (vertices) and segments (edges)

Some Geometric Problems

Segment intersection Given two segments, do they intersect

Simple closed path: given a set of points, find a non-intersecting polygon with vertices on the points.

Inclusion in polygon Is a point inside or outside a polygon ?

Q1 Lines: clockwise/ccw?

- Given directed lines p0p1 and p0p2,
- Is p0-p1 clockwise (cw) turn from p0p2?
- Clockwise (cw) is considered negative angle.
- Anti-clockwise is also called counter clockwise (ccw), is considered positive angle.

Orientation in the Plane

- The orientation of an ordered triplet of points in the plane can be
  - **counterclockwise (left turn)**
  - **clockwise (right turn)**
  - **collinear (no turn)**
- Examples:

c

clockwise (right turn)

counter clockwise (left turn)

collinear no turn

CW and CCW angles

Q2 Line: Left/Right turn?

- Given two line segments p0p1 and p1p2, if we go from p0p1 to p1p2,
- Is there a left or a right turn at p1?

Q3 Lines: intersect?

Do line segments p0p1 and p2p3 intersect?

Cross product of 2 vectors

- Given vectors p1, p2, their *cross-product* p3 is
- p3 = p1 × p2 = (x1.y2) –(x2.y1) = – p2 × p1
- Use the "*right hand rule*" for direction of p3.
- *Magnitude* of p3 is area of the parallelogram formed by p1 and p2, area= (|p1|.|p2|.sin(t)), where t is angle between the vectors.

$$p_1 \times p_2 = \det\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}$$
$$= x_1 y_2 - x_2 y_1$$
$$= -\, p_2 \times p_1$$

Example: cross product

- p1 x p2 = (x1.y2) –(x2.y1)
  = 2*2-1*1=4-1=3, (ccw).
- p2 x p1 = -3 (clockwise)

Left or Right turn?

- Given p0p1 and p1p2,
- Going from p0 to p1 to p2,
- Do we turn left/right at p1?
- Compute d = (p1 – p0) $\times$ (p2 – p0)
- if d > 0, then right turn
- if d < 0, then left turn

Right turn,
counter clockwise
sin(t) > 0

Bounding box

- bounding box is the smallest x-y rectangle containing the object

Intersection of rectangles

- Rect(p1,p2) X Rect(p3,p4) iff
  - ((x1 <= x3 <= x2 <= x4) ||
  - (x3 <= x1 <= x4 <= x2)) &&
  - Similar conditions for y1..y4.

Quick Intersection check

- Collision detection
- Approximate objects by boxes.
  - If bounding box of two objects don't intersect, the objects cannot intersect. (common case).
  - But, If boxes intersect, more complex test is required for interection.

Point Inclusion

- Given a polygon and a point p,
- Is the point inside or outside the polygon?
- Orientation helps solving this problem in linear time

Point Inclusion: algorithm

- Draw a horizontal line to the right of each point and extend it to infinity
- Count the number of times a line intersects the polygon.
  even $\Rightarrow$ point is outside
  odd $\Rightarrow$ point is inside

Degeneracy: What about points d and g (inside/outside)?

Degeneracy

- Degeneracies are input configurations that involve tricky special cases.
- When implementing an algorithm, degeneracies should be taken care of separately -- the general algorithm might fail to work.
- E.g. whether two segments intersect, we have degeneracy if two segments are collinear.
- E.g. The general algorithm of checking for orientation would fail to distinguish whether the two segments intersect.

colinear segments

multiple intersections

Computational Aspects

- We will use +,-,*,< on real numbers.
- Avoid real divisions, causes instability.
- Avoid real equality, as the representation is approximation.
- Instead of **`(x1 == x2)`** use **`eq(x1,x2)`**

```
#include <math.h>
#define eps          1.e-10
#define eq(x1,x2)    (fabs(x1-x2)< eps)
```

2 line intersection

From Cormen chapter 33

Two Segments Intersect in 5 ways

(a) p_1p_2, p_3p_4 intersect

(b) p_1p_2, p_3p_4 do not intersect

(c) p_1p_2, p_3p_4 intersect

(d) p_1p_2, p_3p_4 intersect

(e) p_1p_2, p_3p_4 do not intersect

Two line segments intersect

- check whether each segment straddles the line containing the other.
- A segment p1p2 *straddles* a line if point *p1* lies on one side of the line and point *p2* lies on the other side.
- Two line segments intersect iff either hold:
  - Each segment straddles other.
  - endpoint of one segment lies on the other segment. (boundary case.)

Direction

DIRECTION(*p0*, *p1*, *p2*)

return (*p2* – *p0*) × (*p1* – *p0*)

On-Segment

ON-SEGMENT($p1, p2, p3$)

return

$\min(x1, x2) \leq x3 \leq \max(x1,x2)$

&&

$\min(y1, y2) \leq y3 \leq \max(y1, y2)$

Algorithm

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
d1 ← DIRECTION(p3, p4, p1)
d2 ← DIRECTION(p3, p4, p2)
d3 ← DIRECTION(p1, p2, p3)
d4 ← DIRECTION(p1, p2, p4)
if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0))
   and ((d3 > 0 and d4 < 0) or (d3 < 0 and d4 > 0))
then return TRUE
elseif d1 = 0 and ON-SEGMENT(p3, p4, p1)  // p1 on p3p4?
then return TRUE
elseif d2 = 0 and ON-SEGMENT(p3, p4, p2) // p2 on p3p4?
then return TRUE
elseif d3 = 0 and ON-SEGMENT(p1, p2, p3) // p3 on p1p2?
then return TRUE
elseif d4 = 0 and ON-SEGMENT(p1, p2, p4)  // p4 on p1p2?
then return TRUE
else return FALSE
```

Figure 33.3 Cases in the procedure SEGMENTS-INTERSECT. **(a)** The segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ straddle each other's lines. Because $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$, the signs of the cross products $(p_3 - p_1) \times (p_2 - p_1)$ and $(p_4 - p_1) \times (p_2 - p_1)$ differ. Because $\overline{p_1 p_2}$ straddles the line containing $\overline{p_3 p_4}$, the signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ differ. **(b)** Segment $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$, but $\overline{p_1 p_2}$ does not straddle the line containing $\overline{p_3 p_4}$. The signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ are the same. **(c)** Point p_3 is colinear with $\overline{p_1 p_2}$ and is between p_1 and p_2. **(d)** Point p_3 is colinear with $\overline{p_1 p_2}$, but it is not between p_1 and p_2. The segments do not intersect.

N line intersection

From Cormen chapter 33

Segment intersection

- Given: a set of n distinct segments $s_1 \ldots s_n$, represented by coordinates of endpoints

- Detect if any pair $s_i \neq s_j$ intersects.

- Report all intersection.

N segment intersection

- Brute force, check all nxn intersections in O(*n*^*2*) time.
- worst-case is O(n^2)
- However: if the number of *intersections* m is usually small, we can compute in O(n log n).
- Idea: *Sweep* left to right, looking for intersections.
- Ignoring segments that are too far away for speedup.

Sweeping

- **Sweeping**: An imaginary ***sweep line*** passes through the given set of geometric objects.
- *Two segments* s1,s2 are *comparable* if some vertical sweep line v(*x*) intersects both of them.
- $(s_1 >_x s_2)$ if s1 is above s2 at v(x).

Order segments

- For example, we have the relationships $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$, and $b >_u c$. Segment d is not comparable with any other segment. When segments e and f **intersect**, their orders are reversed: we have $e >_v f$ but $f >_w e$.

Moving the sweep line

- Sweeping algorithms typically manage two sets of data:
  - The ***sweep-line status*** gives the relationships among the objects intersected by the sweep line.
  - The ***event-point schedule*** is a sequence of x-coordinates, ordered from left to right, that defines the halting positions of the sweep line.
  - We call each such halting position an ***event point***.
  - Changes to the sweep-line status occur only at event points.

- The sweep-line status is a total order T, for which we require the following operations:
  - INSERT(T, s): insert segment s into T.
  - DELETE(T, s): delete segment s from T.
  - ABOVE(T, s): return the segment immediately above segment s in T.
  - BELOW(T, s): return the segment immediately below segment s in T.
  - If there are n segments in the input, each of the above operations take $O(\lg n)$ time using red-black trees.

Segment-intersection pseudo-code

```
ANY-SEGMENTS-INTERSECT(S)
T ← { }
EP = sort the endpoints of the segments in S from left to right, breaking
     ties by putting left endpoints before right endpoints and breaking
     further ties by putting points with lower y-coordinates first
for each point p in EP (the sorted list of endpoints)
  if p is the left endpoint of a segment s  then  INSERT(T, s)
       if  (ABOVE(T, s) exists and intersects s) or
              (BELOW(T, s) exists and intersects s)  then return TRUE
  if p is the right endpoint of a segment s  then
  if both ABOVE(T, s) and BELOW(T, s) exist and
     ABOVE(T, s) intersects BELOW(T, s)  then return TRUE
     DELETE(T, s)
return FALSE // no intersections
```

Example

Each dashed line is the *sweep line* at an event point, and the ordering of segment names below each sweep line is the total order *T* at the end of the *for* loop in which the corresponding event point is processed. The intersection of segments *d* and *b* is found when segment *c* is deleted.

Running time

- If there are n segments in set S, then ANY-SEGMENTS-INTERSECT runs in time $O(n \lg n)$.
- Line 2 takes $O(n \lg n)$ time, using merge sort or heapsort. Since there are $2n$ event points, the **for** loop iterates at most $2n$ times.
- Each iteration takes $O(\lg n)$ time, since each red-black-tree operation takes $O(\lg n)$ time and, using the method of line intersection, each intersection test takes $O(1)$ time.

- The total time is thus $O(n \lg n)$.

Closest pair of points

From Cormen chapter 33

Distance

Euclidean distance: ||p –q|| is the shortest distance between points p and q in the plane, ed(p,q) = sqrt(sqr(p_x-q_x) + sqr(p_y-q_y))

Mahanttan distance:

M(p,q) = |px –qx| + |py –qy|

The closet pair problem

Given: a set of points $P=\{p_1 \ldots p_n\}$ in the plane, such that $p_i=(x_i,y_i)$.

Find a pair $p_i \neq p_j$ with minimum $||p_i - p_j||$,

where $|| p-q|| = \text{sqrt}(\text{sqr}(p_x-q_x) + \text{sqr}(p_y-q_y))$

Closest Pair Brute force

- Find a closest pair among $p_1 \ldots p_n$
- Easy to do in $O(n^2)$ time
  - For all $p_i \neq p_j$, compute $\| p_i - p_j \|$ and choose the minimum

- Want $O(n \log n)$ time

Divide and conquer

- Divide:
  - Compute the median of *x*-coordinates
  - Split the points into P^L and P^R, each of size $n/2$
- Conquer: compute the closest pairs for P^L and P^R
- Merge the results (the hard part)

Merge

- Let $d=\min(d_1,d_2)$
- Observe:
  - Need to check only pairs which cross the dividing line
  - Only interested in pairs within distance $< d$
- Suffices to look at points in the 2d-width strip around the median line

Scanning the strip

- S = Points in the strip.
  Q = Sort S by their y-coordinates, Let $q_i = (x_i, y_i)$
  Q = $\{q_1 \ldots q_k : k \leq n$, sorted points in strip by Y-coord $\}$
- $d_{min} = d$
- **For** i=1 **to** k **do**
  j=i-1
  While ($y_i - y_j < d$) **do** // is this O(n^2) ?
  If $||q_i - q_j|| < d$ then
  $d_{min} = ||q_i - q_j||$
  j--
- Report d_{min} (and the corresponding pair)

Merge complexity is not O(n^2)

- Can we have O(n^2) q_j's that are within distance d from q_i ?
- NO
- Theorem: there are at most 7 q_j's such that $y_i - y_j \leq d$.
- Proof by packing argument

Proof: merge complexity.

- Theorem: there are at most 7 q_j's such that $y_i - y_j \leq d$.
- Proof: Each such q_j must lie either in the left or in the right $d \times d$ square
- Note that within each square, all points have distance $\geq d$ from others
- We can pack at most 4 such points into one square, so we have 8 points total (incl. q_i)

Convex Hull

From Cormen chapter 33

What is convex

- *A polygon P is said to be* ***convex*** *if*:
  - *P* is non-intersecting; and
  - for any two points *p* and *q* on the boundary of *P*, segment (*p*,*q*) lies entirely inside *P*

convex hull

- The ***convex hull*** CH of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.
- Example. CH(Q) = $\{p_0, p_1, p_3, p_{10}, p_{12}\}$

Convex Hull examples

Robot motion planning

- *In motion planning for robots, sometimes there is a need to compute convex hulls.*

Graham Scan

- ***Graham Scan*** *algorithm.*
  - *Phase 1:* Solve the problem of finding the non-crossing closed path visiting all points

Finding non-crossing path

- *How do we find such a non-crossing path:*
  - Pick the bottommost point *a* as the anchor point
  - For each point p, compute the angle θ(p) of the segment (a,p) with respect to the *x*-axis.
  - Traversing the points by **increasing angle** yields a simple closed path

Sorting by angle

- *How do we sort by increasing angle*?
  - *Observation*: We do not need to compute the actual angle!
  - We just need to compare them for sorting

$\theta(p) < \theta(q) \Leftrightarrow$
orientation(a,p,q) =
counterclockwise

Rotational sweeping

- *Phase 2 of Graham Scan:* ***Rotational sweeping***
  - The anchor point and the first point in the polar-angle order have to be in the hull
  - Traverse points in the sorted order:
    - Before including the next point *n* check if the new added segment makes a right turn
    - If not, keep discarding the previous point (*c*) until the right turn is made

Implementation and analysis

- *Implementation:*
  - *Stack to store vertices of the convex hull*
- *Analysis*:
  - Phase 1: O(n log n)
    - points are sorted by angle around the anchor
  - Phase 2: O(n)
    - each point is pushed into the stack once
    - each point is removed from the stack at most once
  - Total time complexity O(n log n)

Graham's scan algorithm

- ***Graham's scan*** solves the convex-hull problem by maintaining a stack S of candidate points.
- Each point of the input set Q is pushed once onto the stack, and the points that are not vertices of CH(Q) are eventually popped from the stack.
- When the algorithm terminates, stack S contains exactly the vertices of CH(Q), in counterclockwise order of their appearance on the boundary.
- Graham scan takes $\boldsymbol{O(n \lg n)}$

GRAHAM-SCAN

- The procedure GRAHAM-SCAN takes as input a set Q of points, where $|Q| \geq 3$.
- It calls the functions TOP(S), which returns the point on top of stack S without changing S, and NEXT-TO-TOP(S), which returns the point one entry below the top of stack S without changing S.
- The stack S returned by GRAHAM-SCAN contains, from bottom to top, exactly the vertices of CH(Q) in counterclockwise order.

Graham scan algorithm

GRAHAM-SCAN(Q)

let p_0 be the point in Q with the min y-coord,
  or the leftmost such point in case of a tie

let $[p_1, p_2, ..., p_m]$ be the remaining points in Q,
  sorted by polar angle in counterclockwise order around p_0
    (if more than one point has the same angle,
    keep only the farthest point from p_0)

.. PUSH(p_0, S) ; PUSH(p_1, S) ; PUSH(p_2, S)

for $i \leftarrow 3$ **to** m

do while the angle formed by points NEXT-TO-TOP(S), TOP(S),
  and p_i makes a nonleft turn

do { POP(S) }

PUSH(p_i, S)

return S

Illustration:

sorts the remaining points of Q by polar angle relative to p_0, using the method comparing cross products

Right turn, pop(p4)

Illustration continued

Step 8: while ..
pop p8,
pop p7

Graham Scan - Example

Graham Scan - Example

Graham Scan - Example

Graham Scan - Example

Graham Scan - Example

Graham Scan - Example

Graham Scan - Example

Graham Scan - Example

Graham Scan - Example

Graham Scan - Example

Graham Scan - Example

Graham Scan - Example

Graham Scan - Example

Proof.

- Focus on this moment just before p_i is pushed, to decide whether pj stays in the CH
- Because p_i's polar angle relative to p_0 is greater than p_j's polar angle, and because the angle p(k,j,i) makes a left turn (otherwise we would have popped p_j)

Running time

- Line 1 takes $\Theta(n)$ time.
- Line 2 takes $O(n \lg n)$ time, using merge sort or heapsort to sort the polar angles and the cross-product method of to compare angles. (Removing all but the farthest point with the same polar angle can be done in a total of $O(n)$ time.)
- Lines 3-5 take $O(1)$ time. Because $m \leq n - 1$, the **for** loop of lines 6-9 is executed at most $n - 3$ times. Since PUSH takes $O(1)$ time, each iteration takes $O(1)$ time exclusive of the time spent in the **while** loop of lines 7-8, and thus overall the **for** loop takes $O(n)$ time exclusive of the nested **while** loop.
- We use aggregate analysis to show that the **while** loop takes $O(n)$ time overall.
- continued ..

Running time continued

- For $i = 0, 1, ..., m$, each point p_i is pushed onto stack S exactly once. We observe that there is at most one POP operation for each PUSH operation.
- At least three points-p_0, p_1, and p_m-are never popped from the stack, so that in fact at most $(m – 2)$ POP operations are performed in total.
- Each iteration of the **while** loop performs one POP, and so there are at most m - 2 iterations of the **while** loop altogether.
- Since the test in line 7 takes $O(1)$ time, each call of POP takes $O(1)$ time, and since $m \leq n$ - 1, the total time taken by the **while** loop is $O(n)$.
- Thus, the running time of GRAHAM-SCAN is $\boldsymbol{O(n \lg n)}$.

Convex Hull – Jarvis march

From Cormen chapter 33

Jarvis's March (JM)

- JM computes the convex hull of a set Q of points by a technique known as package wrapping (or gift wrapping).
- The algorithm runs in time $O(n\ h)$, where h is the number of vertices of CH(Q).
- When h is $o(\lg n)$, JM is asymptotically faster than Graham's scan.
- JM wraps Q from left and right sides, going from the bottom to the highest point.

Convex hull problem

- *Convex hull problem:* Let *S* be a set of *n* points in the plane. Compute the convex hull of these points.
- *Intuition*: rubber band stretched around the pegs
- *Formal definition:* the ***convex hull*** of *S* is the smallest convex polygon that contains all the points of *S*

Idea

- JM wraps Q from two sides:
  - right (going from the bottom to top)
  - left (going top to bottom)

JM right chain

- JM builds a sequence $H = (p_0, p_1, ..., p_{h-1})$ of the vertices of CH(Q).
- Start with p_0, the next convex hull vertex p_1 has the smallest polar angle with respect to p_0. (In case of ties, choose farthest point from p_0.)
- Similarly, p_2 has the smallest polar angle with respect to p_1, and so on. (break ties by choosing the farthest such vertex).
- When we reach the highest vertex, say p_k we have constructed the ***right chain*** of CH(Q).

JM left chain

- To construct the ***left chain***, we start at p_k and choose p_{k+1} as the point with the smallest polar angle with respect to p_k, but *from the negative x-axis*.
- We continue on, forming the left chain by taking polar angles from the negative *x*-axis, until we come back to our original vertex p_0.

JM Illustration

Jarvis March - Example

Jarvis March - Example

Jarvis March - Example

Jarvis March - Example

Jarvis March - Example

Jarvis March - Example

Jarvis March - Example

Running time

- If implemented properly, JM has a running time of $O(nh)$.
- where h is size of CH(Q).
- For each of the h vertices of CH(Q), we find the vertex with the minimum polar angle.
- Each comparison between polar angles takes $O(1)$ time
- We can compute the minimum of n values in $O(n)$ time if each comparison takes $O(1)$ time.
- Thus, Jarvis's march takes $O(nh)$ time.

Convex hull - few other methods

Divide & Conquer

- Sort the points from left to right
- Let A be the leftmost $\lceil n/2 \rceil$ points
- Let B be the rightmost $\lfloor n/2 \rfloor$ points
- Compute convex hulls $H(A)$ and $H(B)$
- Compute $H(A \cup B)$ by merging $H(A)$ and $H(B)$

Merging is tricky, but can be done in linear time

[Adapted from K.Lim Low]

Convex hull - few other methods

Need to find the upper and lower tangents

They can be found in linear time

[Adapted from K.Lim Low]

Convex hull – Divide and conquer in 3D

Merging still can be done in linear time!

We have O(n log n) in 3D

[Adapted from K.Lim Low]

Convex hull - few other methods

<u>Idea:</u> incrementally add a point to a convex polyhedra *P*

Two cases:

1. The new point is inside *P* → do nothing
2. The new point is outside of *P* → fix *P*!

[Adapted from F.Joskowitz]

Convex hull

Naive implementation: $O(n^2)$
O(n log n) randomized algorithm
Can be made deterministic with the same running time.

[Adapted from F.Joskowitz]

Complexity in 3 and more dimensions?

How many “faces” complex hull might have?

2D	up to	n
3D	up to	?
3D	up to	n
kD	up to	?

- Unfortunately, convex hull in *d* dimensions can have $\Omega(n^{\lfloor d/2 \rfloor})$ facets (proved by Klee, 1980)

- No $O(n \lg n)$ algorithm possible for $d > 3$

Complexity in 3 and more dimensions?

- Gift wrapping method requires $O(n^{\lfloor d/2 \rfloor+1})$ time.

- There are algorithms that works in $O(n \log n + n^{\lfloor d/2 \rfloor})$ time.

Polygon Triangulation

Applications of Polygon Triangulation

- Guarding Art galleries
- Partitioning of shapes/polygons into triangular pieces for FEM (finite element modeling in CAD/CAM)

Guarding art gallery

- Minimum number of guards needed to keep watch on every wall?

Guarding art galleries

- The gallery is represented by a simple polygon
- A guard is represented by a point within the polygon
- Guards have a viewport of 360°
- A polygon is completely guarded, if every point within the polygon is guarded by at least one of the watchmen

Guarding art galleries

- Even if two polygons have the same number of vertices, one may be easier to guard than the other.

- Determine minimum number of guards for an arbitrary polygon with n vertices.

Polygon Triangulation

- Given a polygon, we want to decompose it into triangles by adding *diagonals*: new line segments between the vertices that don't cross the boundary of the polygon.

Triangulation of simple polygons

Simple polygon: polygon does not cross itself.
Does every simple polygon have a triangulation?
How many triangles in a triangulation?
Minimize number of triangles.

Theorem

Theorem: Every simple polygon admits a triangulation, and any triangulation of a simple polygon with n vertices consists of exactly n-2 triangles.

Proof. by induction.

- The base case n = 3 is trivial: there is only one triangulation of a triangle, and it obviously has only one triangle.
- Let P be a polygon with n edges. Draw a ***diagonal*** between two vertices. This splits P into two smaller polygons.
- One of these polygons has k edges of P plus the diagonal, by the induction hypothesis, this polygon can be broken into k - 1 triangles.
- The other polygon has n - k + 1 edges, and so by the induction hypothesis, it can be broken into n - k -1 tirangles.
- Putting the two pieces back together, we have a total of (k - 1) + (n - k - 1) = n - 2 triangles.

Existence of diagonal

- How do we know that every polygon has a diagonal? [Meisters in 1975]
- Lemma. Every polygon with more than three vertices has a diagonal.
- Proof. Let P be a polygon with more than three vertices. Let q be the leftmost vertex. Let p and r be two neighboring vertices of q.

Proof of Existence of diagonal continued

Case 1: pr completely in P. Then segement pr is a diagonal

Case 2: pr not completely in P. Let s be the vertex furthest away from the segment pr. Then the line qs is a diagonal.

Partitioning problem

Given: n points are scattered inside a convex polygon *P* (in 2D) with *m* vertices.

Does there exist a partition of *P* into *n* sub-regions satisfying the following:

- Each sub-region is a convex polygon
- Each sub-region contains at least one point
- All sub-regions have equal area

Partitioning result

Given: n points are scattered inside a convex polygon P (in 2D) with m vertices. Does there exist a partition of P into n sub-regions satisfying the following:

An equitable partition always exists, we can find it in running time $O(N\, n \log N)$, where $N = m + n$.

Partitioning Applications

Example: Broadcast Network problems, to connect *clients* to *servers*, in a fixed underlying network topology.

Example: *Multi-Depot Vehicle Routing Problem* (*MDVRP*).

Definition: A set of *vehicles* located at *depots* in the plane must visit a set of *customers* such that the maximum TSP cost is minimized (min-max MDVRP).

Voronoi Diagrams and Delaunay Triangulations

Voronoi Diagrams

Voronoi diagrams

- Given a set of n points.
- Split the space into regions of points closest to the given points.

Applications

- **Nearest-neighbor queries**: Given a point, find which region it belongs.
- **Site assignment**: Given a set of hospital locations, decide which is the nearest hospital for each home.

Voronoi diagrams and clustering

- A Voronoi diagram stores proximity among points in a set

Voronoi Diagram: Corner cases

- If all the sites are colinear, the Voronoi diagram will look like this:

- Otherwise, the diagram is a connected planar graph, in which all the edges are line segments or rays

Voronoi Diagram Complexity

- A Voronoi diagram of n distinct sites contains n cells.
- One cell can have complexity n-1, but not all the cells can.
  - The number of vertices $V \leq 2n-5$
  - The number of edges $E \leq 3n-6$
  - The number of faces $F = n$

Voronoi Diagram Algorithms

- **Input:** A set of points locations (*sites*) in the plane.
- **Output:** A planar subdivision into cells. Each cell contains all the points for which a certain site is the closest.

Computing Voronoi diagrams

- Naïve, brute force - edge bisectors.
- By plane sweep.
- By randomized incremental construction
- By divide-and-conquer

All are $O(n \log n)$ time except brute force

Voronoi Diagram Properties

- A vertex of a Voronoi diagram is the center of a circle passing through three sites.
- Each point on an edge is the center of a circle passing through two sites.

Voronoi Diagram: naive algorithm

- The ♈ *bisector* of two points is a line.

all points on this side
are closer to p1

this line is perpendicular
bisector of p1 and p2

all points on this side
are closer to p2

Voronoi Diagram: naïve algorithm

- Assume no four sites (green) lie on a circle.
- The Voronoi diagram is
  - a planar-graph (blue),
  - whose vertices (pink) are equidistant from three sites
  - edges equidistant from two sites.
- The convex hull of the (outer) sites are those who have an unbounded cell.

Voronoi Diagram – Naïve algorithm

- Construct bisectors between each site and all others (n^2).
- A **Voronoi cell** is the intersection of all half-planes defined by the bisectors.
- **Corollary:** Each cell in a Voronoi diagram is a convex polygon, possibly unbounded.
- **Time complexity**: O(*n*^2 log*n*)

Computing Voronoi diagrams

- Study the geometry, find properties
  - 3-point empty circle ↔ Voronoi vertex
  - 2-point empty circle ↔ Voronoi edge

Computing Voronoi diagrams

- Some geometric properties are needed, regardless of the computational approach
- Other geometric properties are only needed for some approach

Computing Voronoi diagrams

- Fortune's sweep line algorithm (1987)
  - An imaginary line moves from left to right
  - The Voronoi diagram is computed while the known space expands (left of the line)

Computing Voronoi diagrams: Beach front

- Beach line: boundary between known and unknown → sequence of parabolic arcs
  - Geometric property: beach line is *y*-monotone → it can be stored in a balanced binary tree

Computing Voronoi diagrams

- Events: changes to the beach line = discovery of Voronoi diagram features
  - Point events

Computing Voronoi diagrams

- Events: changes to the beach line = discovery of Voronoi diagram features
  - Point events

Computing Voronoi diagrams

- Events: changes to the beach line = discovery of Voronoi diagram features
  - Circle events

Computing Voronoi diagrams

- Events: changes to the beach line = discovery of Voronoi diagram features
  - Circle events

Computing Voronoi diagrams

- Events: changes to the beach line = discovery of Voronoi diagram features
  - Only point events and circle events exist

Computing Voronoi diagrams

- For n points, there are
  - n point events
  - at most $2n$ circle events

Computing Voronoi diagrams

- Handling an event takes $O(\log n)$ time due to the balanced binary tree that stores the beach line → in total $O(n \log n)$ time

Divide-and-Conquer

- Divide the points into two parts.

Fig. 5-17: Two Voronoi Diagrams After Step 2

Merging two Voronoi diagrams

- Merging along the piecewise linear hyperplane

Fig. 5-18: The Piecewise Linear Hyperplane for the set of Points Shown in Fig. 5-17.

The final Voronoi diagram

- After merging

Fig. 5-19: The Voronoi Diagram of the Points in Fig. 5-17.

Time Complexity

- Merging takes O(n) time (**This is the Key!**)
- $T(n) = 2 * T(n/2) + O(n)$
- $T(n) = O(n \log n)$

Intermediate summary

- Voronoi diagrams are useful for clustering (among many other things)
- Voronoi diagrams can be computed efficiently in the plane, in $O(n \log n)$ time
- The approach is plane sweep (by Fortune)

Figures from the on-line animation of Allan Odgaard & Benny Kjær Nielsen

Dual graphs of Voronoi diagrams

- Delaunay triangulation is a dual graph to a Voronoi diagram.

Delaunay traingulation

- also called tessalonation

Delaunay traingulation

- An *edge e* is called *Delaunay edge* ***iff*** there exist an ***empty*** circle passing through e.
- Delaunay triangulation - a set of Delaunay edges

Delaunay triangulations and convex hulls

Project the 2D point set onto the 3D paraboloid → Compute the 3D lower convex hull → Project the 3D facets Back to the plane.

The 2D triangulation is Delaunay!

Intractable (Hard) Problems

Combinatorial puzzles

- Hard to solve: exponential number of choices, too many to try one by one.
- But easy to check if a given solution is correct or not.

Finding a Hamiltonian cycle on a dodecahedron, old puzzle

- Left: Dodecahedron with a solution
- Right: dodecahedron flattened into a planar graph

How many choices is too many?

- Number of subsets of n elements is $O(2^n)$.
- Size of truth table of n boolean variables, is $O(2^n)$
- Choose k out of N is also exponential, $O(n^k)$.

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \quad \text{for } 0 \le k \le n.$$

- Number of paths in a graph with n nodes, is n!, $O(n^n)$.

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Hamiltonian Cycle is NPC

Is there a Hamilton cycle (tour that visits each vertex exactly once)?

Number of cycles to check is n!

Hamiltonian Path problem is NPC
(no need to return to start).

Euler Path is in P

- **Eulerian path** is a trail in a graph which visits every edge exactly once.
- Examples:

G1. This Königsberg Bridges graph does not have a Euler circuit.

G2. Every vertex of this graph has an even degree, therefore this is an Eulerian graph. Following the edges in alphabetical order gives an Eulerian **circuit**/cycle

k-Clique problem is NPC

k-Clique problem: Does G have a clique of size k?

Number of k-cliques to test is C(n,k).

Max-clique: What is the largest number of nodes in G forming a **complete** (clique) subgraph?

Subgraph isomorphism problem

- K is called a subgraph of G,
  - If K is embedded inside G, i.e. by deleting some vertices and edges of G, we get K.
- Deciding if graph K is a subgraph of G is NPC
  - It is as hard as finding a clique K in G.

Chromatic number of a Graph

Given a graph G, how many colors are required to paint the vertices? such that adjacent vertices have different colors.

A torus requires 7 colors!

4 colors suffice for planar graphs

[Appel and Haken] "A proof of the 4-color theorem." *Discrete Mathematics*, 16, 1976.

Graph 3-Coloring is NPC

Given a graph G, Can we color the vertices with 3 colors? where adjacent vertices have different colors.

Exercise: 3 coloring a graph

- Is it possible to color G with 3 colors {Green, Red, Blue}?
- Each node must be colored differently from its neighbours.
- Use minimal colors
- In G, vertex 1 is red, 3 is green, 6 is red.
- What are the possible colors for the nodes {2,4,5,7,8,9}?

Vertex Cover (VC) is NPC

- Given a graph G and an integer k, is there a subset of k vertices (called cover) such that every edge is connected to a vertex in the cover?

Independent set is NPC

Independent set is a subset of vertices in the graph with no edges between any *pair* of nodes in the independent set.

Given a graph G=(V,E), find the *largest independent set.*

Examples:

Exercises

1. Find a maximum independent set S
2. Find a minimal vertex cover
3. Find a hamiltonian path
4. Find an Euler circuit
5. 3 Color the nodes

G2

G1

Knapsack problem is NPC

Given a set A integers a1..an, and another integer K in binary.

Q. Is there a subset B of A, of these integers that sum exactly to K?

Brute force packing

Bin packing problem is NPC

Can k of bins of capacity C each, store a finite collection of weights W = {w1,..,wn}?

Example:
We have 4 bins with capacity 10 each.
k=4 bins, C=10, W = {1,9,3,7,6,4,1,2,2,5}

Integer subset sum (partition) is NPC

Given a set of numbers,can you divide it into 2 sets of equal sums?

Restaurant ordering is NP Complete
(to divide the bill exactly)

3D matching is NPC

3D-Matching Problem: Is there a matching of size k?

Given 3 sets {X,Y,Z} and their compatibilities.

But 2D matching is easy, using bipartite graph matching.

Language of Boolean formulas

1. Constants: True, False (1, 0).
2. Boolean Variables: {x1, x2, x3,…}
3. Literals are the variables {x1, x2, x3,…} or its negation {¬x1, ¬x2, ¬x3, …}.
4. Negation (NOT, !, ~, ¬, operator).
5. Conjunction is (AND, ∧, &, operator) of literals are called clause.
6. Disjunctions is (OR,v,|, operator)

Language of Boolean formulas

1. OR(AND(literals)) is called DNF formula (disjunctive normal form).
2. AND(OR(literals)) is called CNF formula (conjunctive normal form).
3. CNF formula, with 3 literals in each clause is called 3CNF.

e.g. of 3CNF $(X_1 \vee \neg X_2 \vee \neg X_3) \wedge (\neg X_2 \vee \neg X_3 \vee X_4)$.

e.g. of 2CNF $(x_3 \vee x_6) \wedge (x_4 \vee x_6) \wedge (x_5 \vee x_6)$.

Language of Boolean Logic

1. A formula is satisfiable, if it is true for some truth assignment to its variables. e.g. S="(x or y or not(z))".
2. A formula is valid (tautology), if it is true for all (any) truth assignment to is variables. e.g. V="(x or not(x))"
3. A formula is unsatisfiable (invalid), if it is false for all values of its variables, e.g. U="(x and not(x))"

Boolean Algebra

- Distributive Laws:

  x or (y & z) = (x or y) & (x &z)

  x & (y or z) = (x & y) or (x & z)
- D'Morgan's laws:

  !(x & y) = !x or !y

  !(x or y) = !x & !y
- Any boolean formula can be converted into standard form called CNF:
  (...or...)&(..or..)&(..or..)...

  E.g. $P = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4)$

Boolean formulas

Consider the Boolean expression:

$$P = (x_1 \vee \neg x_2 \vee x_3) \quad \text{.. clause 1}$$
$$\wedge\ (x_2 \vee \neg x_3 \vee x_4) \quad \text{.. clause 2.}$$

1. P is a conjunction (AND) of 2 clauses.
2. Each clause is a disjunctions (OR) of 3 literals (called 3CNF)
3. The literals are the variables {x1, x2, x3} or its negation (not) {¬x1, ¬x2, ¬x3}.
4. P is true, if every clause is true, easy enough?

Satisfiability (SAT)

- Is this boolean expression satisfiable?

  $Q = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee x_4 \vee x_5)$
- Can you assign some truth values to all the x_i's such that Q = true?
- Easy solution: set x_1, x_2, and x_4 to True
- More solutions: $\{x_1, x_2, x_5\}$ ={True}, …
- But how to solve a formula with 5000 variables and 300 clauses? Truth table will have $O(2^{5000}) = O(10^{1500})$ rows to check.

Satisfiability problem (SAT) is NPC

Given a boolean formula, determine if there exists an assignment to the variables, that satisfies the formula?

3SAT is also NPC

- 3CNF is a boolean formula with just 3 literals per clause.
- 3SAT problem: is a given 3CNF formula is satisfiable? NP Complete.

- But 2-SAT is in P (polynomial algorithm).

Examples of NP-complete problems

some NPC problems

find more NP-complete problems in
http://en.wikipedia.org/wiki/List_of_NP-complete_problems

Some NPC Problems

NP Complete list?

- There are hundreds of NP complete problems
- It has been shown that if you can solve one of them efficiently, then you can solve them all efficiently (polynomial time).

NPC Problems are all related

TSP: The traveling saleperson problem

Given a graph with N cities and edges with costs (the costs of traveling from any city to any other city)

What is the cheapest round-trip route that visits each city once and then returns to the starting city?

This is an optimization problem (find the minimum costs: not a decision yes/no problem)

Traveling Salesman vs Hamiltonian Path

1. The **Decision** problem: Is there a Hamiltonian circuit of a given cost **NP-Complete**.
2. The **Optimization** problem: Traveling Salesman Problem is to find the cheapest circuit is NP Hard, Not **NP-Complete**.

Graph isomorphism problem is ??

- Two graphs G and H are isomorphic, if they have the same structure (connections) but their vertices may have different names.
- Example:

Unresolved complexity: deciding if two graphs isomorphic?

Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$

Example

Unresolved complexity Integer Factoring.

- Not all algorithms that are $O(2^n)$ are NP complete.
- Integer factorization (also $O(2^n)$) is not thought to be NP complete.
- Note: Size of the problem is measured by the number of digits in n (not the value of n). E.g. n=999, size of n is 3.

"Is Primes" is in P

- "Primes is in P" [Manindra Agrawal. Kayal, Saxena, IIT Kanpur, 2002]
- They give an unconditional deterministic polynomial-time algorithm
- The algorithm decides whether an input number is **prime** or composite (but not its factors).

P = NP?
NP Completeness

General Definitions

P, NP, NP-hard, NP-easy, and NP-complete

- Problems

  - Decision problems (yes/no)

  - Optimization problems (solution with best score)

- P

  - Decision problems (decision problems) that can be solved in polynomial time (efficiently).

- NP

  - Decision problems whose "*YES*" answer can be verified in polynomial time, if we already have the *proof* (or *witness*)

- co-NP

  - Decision problems whose "*NO*" answer can be verified in polynomial time, if we already have the *proof* (or *witness*)

P = NP?

- P: The complexity class of decision problems that can be solved on a deterministic Turing machine in polynomial time.

- NP: The complexity class of decision problems that can be solved on a non-deterministic Turing machine in polynomial time.

- P=NP is the biggest unsolved problem in CS

- Text book: Garey and Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, 1979.

NP-Completeness

- How would you define NP-Complete?
- They are the “hardest” problems in NP

Definition: S is called NP-Complete if

1. Problem S in NP and
2. S is NP hard: Every other NP problem is polynomial time reducible to S

If P ≠ NP

If P = NP

Definition: S is in P

- S is in P if S can be solved in polynomial time by a deterministic turing macine.

If P ≠ NP

If P = NP

An other P versus NP diagram

1. **P**: The class of problems that can be solved by a deterministic Turing Machine in polynomial time.
2. **NP**: The class of problems that can be solved by a non-deterministic Turing Machine in polynomial time.
3. **NP-complete**: The class of problems that are in NP and can be reducible by any problem in NP.
4. **NP-hard**: The class of problems that can be reducible by any problem in NP

from http: //henry2005.jimdo.com/2011/05/21/p-np-np-hard-and-np-complete/

P=NP?

P=NP

If P=NP then

- If P=NP, all NP-hard problems can also be solved in polynomial time, e.g. Satisfiability, Clique, Graph coloring, Partition numbers
- Assume, all the basic arithmetic operations (addition, subtraction, multiplication, division, and comparison) can be done in polynomial time.

Satisfiability problem (SAT) is NPC

Given a boolean formula, determine if there exists an assignment to the variables, that satisfies the formula?

- NP: Checking a given solution is easy, If given a solution to SAT, just plug in the values and evaluate to True or False, this can be done quickly, even by hand.
- NP Complete: Finding a solution is hard. Brute force algorithm: try all possible T/F values for the n Boolean variables, $O(2^n)$, exponential time.

SAT is NPC

- No known efficient way to find a solution.
- There still may be an efficient way to solve it, though! We don’t know.
- Cook’s theorem (1971) states that SAT is NP-complete. Cook showed that a turing machine computation can be written as a boolean formula.

P = NP?

- P=NP is an Unsolved problem.
- If you could solve any NP complete problem in polynomial time, you would be showing that P = NP

NP Complete problem set

Hardness is hard to prove: nobody has found a fast algorithm for any NPC problem.

"I CAN'T SOLVE IT - BUT NEITHER CAN ALL THESE FAMOUS PEOPLE!"

But what if P=NP, and the SAT algorithm is $O(n^{10000000})$?

FIGURE 3 A Display of the Growth of Functions Commonly Used in Big-*O* Estimates.

Coping with NP-Hardness

- Approximation Algorithms
  - Run quickly (polynomial-time).
  - Obtain solutions that are guaranteed to be close to optimal

Parallel Computers

- Maybe there are physical system that solves an NP-complete problem just by reaching its lowest energy state?
- **DNA computers: Just highly parallel ordinary computers**
- **Quantum computers?**

Physical: soap solver

Dip two glass plates with pegs between them into soapy water

Let the soap bubbles form a minimum Steiner tree connecting the pegs

NP, NP-Hard and NPC Reductions

- NP: to show B is in **NP**, it has non-deterministic polynomial-time guess-and-verify algorithm.
- NP Hard: A decision problem B is said to be **NP**-hard if every decision problem in **NP** reduces to B; that is, for every decision problem S in **NP**, $S \leq_p B$. Thus B is as hard as any problem in **NP**.
- NP Complete: A decision problem B is called **NP-complete** if it is **NP**-hard and in **NP** itself.
- Reductions is used to show that a problem B is at least as hard as another problem S. If S is NPC, then B is also NPC.

Reduction $S \leq_p B$

- S is our known hard problem, and we want to show that B is at least as hard, by showing that S reduces to B in polynomial time.

Sample Reductions

1) PARTITION = { $n_1, n_2, \ldots n_k$ | we can split the integers into two sets which sum to half the total }.
2) SUBSET-SUM = { $\langle n_1, n_2, \ldots n_k, m \rangle$ | there exists a subset which sums to m }

1) If I can solve SUBSET-SUM, how can I use that to solve an instance of PARTITION?
2) If I can solve PARTITION, how can I use that to solve an instance of SUBSET-SUM?

Reduce: Partition $<_p$ Subset-Sum

Given $T = \{ n_1, n_2, \ldots n_k \}$ to partition into two

Let $S = n_1 + n_2 + \ldots + n_k$

We want two paritions of sum=S/2 each.

call SUBSET-SUM($\{ n_1, n_2, \ldots n_k \}$, S/2) will give us one set equal to S/2, and remaining also will sum to S/2.

Reduce: Subset-Sum $<_p$ Partition

Given $T=\{n_1,n_2,\ldots n_k\}$ with sum $S= n_1+ n_2+\ldots +n_k$
To find a subsets of T with sum m (and S-m remaining).

Let w=S-2m, add w to T, and
Call Partition($\{n_1,n_2,\ldots n_k,w\}$)
To get two equal partitions: {m + w, S–m}
Remove w from one partition to get m.

Example.
To solve subset-sum: S=100, m=30,
Let S-m=70, w=40,
call Partition(100+40) to get 40+30, and 70.

Polynomial Reductions

1) Partition REDUCES to Subset-Sum
   - Partition $<_p$ Subset-Sum
2) Subset-Sum REDUCES to Partition
   - Subset-Sum $<_p$ Partition

- Therefore they are equivalently hard

NP-Completeness Proof Method

To show that Q is NP-Complete:

1) Show that Q is in NP

2) Pick an instance, S of your favorite NP-Complete problem (ex: Φ in 3-SAT)

3) Show a polynomial algorithm to transform S into an instance of Q

Starting with SAT

- Cook and Levin independently showed that SAT is **NP-complete**.
- Since the reducibility relation $\leq_p$ is transitive, one can show that another problem B is **NP**-hard by reducing SAT, or any of the other known **NP-complete** problems, to B.

Showing NP

- To show that B is in **NP**, you should give a nondeterministic polynomial-time guess-and-verify algorithm for it.
- Give a (small) solution, and
- Give a deterministic polynomial-time algorithm to check the solution.

Showing NPC

To show that B is **NP-complete**,

1. Show that B is in **NP-**hard by giving a polynomial-time **reduction** from one of the known **NP-complete** problems (SAT, CNFSAT, CLIQUE, etc.) to B.
2. Show that B is in **NP** by giving a nondeterministic polynomial-time guess-and-verify algorithm for it.

Showing NP Hardness:

- If you are only asked to show **NP**-hardness, you only need to do 1. But if you are asked to show **NP**-completeness, you need to do both 1 and 2

Example: Clique is NPC

- CLIQUE = { <G,k> | G is a graph with a clique of size k }
- A clique is a subset of vertices that are all connected
- Why is CLIQUE in NP?

Reduce 3SAT to Clique

- Given a 3SAT problem Φ, with k clauses
- Make a vertex for each literal.
- Connect each vertex to the literals in other clauses that are not its negations.
- Any k-clique in this graph corresponds to a satisfying assignment (see example in next slide)

3SAT as a clique problem

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

Example: Independent Set is NPC

- Defintion: An k-IS, is a subset of k vertices of G with no edges between them.
- Decision problem: Does G have a k-IS?
- To Show k-IS is NPC
- Method: Reduce k-clique to k-IS

Reduce Clique to IS

- Independent set is the *dual problem* of Clique.
- Convert G to G^c (complement graph).
- G has a k-clique iff G^C has k-IS

Theorem: HAMILTONIAN-PATH is NP-complete

Proof:

1. HAMILTONIAN-PATH is in NP
   Can be easily proven

2. We will reduce in polynomial time
   3CNF-SAT to HAMILTONIAN-PATH
   (NP-complete)

3SAT to H-Path: Reduction explained

- For each variable we will build a small graph called a "gadget".
- We will stack the gadgets for each variable
- The h-path will pass through each gadget,
  - Starting at the top,
  - Passing through the center (left to right OR right to left), and
  - Ending at the bottom.

$$(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2)$$

clause 1 clause 2

Gadget for variable x_1

There are only 2 possible h-paths in gadget for x1, depending on whether x1 is true or false

if the variable is negated, the arrows are reversed (right to left).

3SAT to H-Path: Reduction explained

- The h-path can only go in one direction:
  - Left to Right (meaning X=1), visiting clauses where x is used
  - Right to Left (meaning X=0), visiting clauses where !x is used
- h-path goes thru a clause iff the clause is true.
- h-path can go thru all clauses iff
  - 3SAT is true.
  - Path is Hamiltonian.

$$(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2)$$

Gadget for variable x_2

$$(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2)$$

$$(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2) = 1$$

$$(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2) = 1$$

Vertex Cover is NPC

Reductions

If: Language A is NP-complete

Language B is in NP

A is polynomial time reducible to B

Then: B is NP-complete

Vertex Cover

VC of a graph is a subset S of nodes such that every edge in the graph touches one node in S

Example:

S = red nodes

Size of vertex-cover is the number of nodes in the cover

Example: $|S|=4$

Corresponding language:

VERTEX-COVER = { $\langle G, k \rangle$:
graph G contains a vertex cover
of size k }

Example:

$$\langle G', 4 \rangle \in \text{VERTEX-COVER}$$

Theorem: VERTEX-COVER is NP-complete

Proof:

1. VERTEX-COVER is in NP

   Can be easily proven

2. We will reduce in polynomial time

   3CNF-SAT to VERTEX-COVER

   (NP-complete)

Let P be a 3CNF formula
with n variables and c clauses

Example:

$$\varphi = \underbrace{(x_1 \vee x_2 \vee x_3)}_{\text{Clause 1}} \wedge \underbrace{(\overline{x_1} \vee \overline{x_2} \vee \overline{x_4})}_{\text{Clause 2}} \wedge \underbrace{(\overline{x_1} \vee x_3 \vee x_4)}_{\text{Clause 3}}$$

n=4, c=3

Formula P can be converted
to a graph G such that:

P is satisfiable if and only if
G Contains a vertex cover of size $k=n+2c$

3SAT to VC reduction explained

- Given a 3CNF formula P.
- We will build gadgets (small-graph) for each literal (variables x and !x).
- We will build gadgets for each clause.
- We connect the literals to the clause they appear in.

$$\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4)$$

Clause 1 Clause 2 Clause 3

Variable Gadgets, with 2n nodes

Clause Gadgets, with 3c nodes

Clause 1 Clause 2 Clause 3

$$\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4)$$

Clause 1 Clause 2 Clause 3

First direction in proof:

If P is satisfiable,
then G contains a vertex cover of size
$k = n+2c$

Example:

$$P = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4)$$

Satisfying assignment:

$x_1 = 1 \qquad x_2 = 0 \qquad x_3 = 0 \qquad x_4 = 1$

We will show that G contains a VC of size $k=n+2c = 4+2x3 = 10$

$$P = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4)$$

$x_1 = 1$ $\quad$ $x_2 = 0$ $\quad$ $x_3 = 0$ $\quad$ $x_4 = 1$

Put every satisfying literal in the cover

$$P = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4)$$

$x_1 = 1$ $\quad$ $x_2 = 0$ $\quad$ $x_3 = 0$ $\quad$ $x_4 = 1$

Select one satisfying literal in each clause gadget and include the remaining literals in the cover

This is a vertex cover since every edge is adjacent to a chosen node

Explanation for general case:

Edges in variable gadgets
are incident to at least one node in cover

Edges in clause gadgets
are incident to at least one node in cover,
since two nodes are chosen in a clause gadget

Every edge connecting variable gadgets and clause gadgets is one of three types:

All adjacent to nodes in cover

Second direction of proof:

If graph G contains a vertex-cover of size $n+2c$ then formula P is satisfiable

Example:

x_1
$\overline{x_1}$
x_2
$\overline{x_2}$
x_3
$\overline{x_3}$
x_4
$\overline{x_4}$
x_1
$\overline{x_1}$
$\overline{x_1}$
x_2
x_3
$\overline{x_2}$
$\overline{x_4}$
$\overline{x_3}$
x_4

To include "internal" edges to gadgets, and satisfy $k = n + 2c$

Exactly one literal in each variable gadget is chosen

n chosen out of 2n

Exactly two nodes in each clause gadget is chosen

2c chosen out of 3c

For the variable assignment choose the literals in the cover from variable gadgets

$x_1 = 1$ $\quad$ $x_2 = 0$ $\quad$ $x_3 = 0$ $\quad$ $x_4 = 1$

$$P = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4)$$

$$P = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4)$$

is satisfied with

$x_1 = 1$ $x_2 = 0$ $x_3 = 0$ $x_4 = 1$

since the respective literals satisfy the clauses

Vertex cover is NPC, part 2.

VC Problem: Given a graph G=(V,E), find the *smallest* number of vertexes that cover *each edge*

Decision problem: Does a graph G have a vertex cover of size k?

Proof of NPC, Reduce: 3SAT to VC.

Reduction: 3SAT to Vertex cover

- Given 3CNF formula P, create a gadget graph G.
- G has Vertex Cover iff P is SAT.
- P has n variables and c clauses, VC has size n + 2c.

Example:

$$P = \overbrace{(a \vee b \vee c)}^{\text{clause}} \wedge (b \vee \bar{c} \vee d) \wedge (\bar{a} \vee c \vee \bar{d})$$

literal

G=

3SAT solution to VC solution

Given a solution for P, create a VC for G:

1. Put the TRUE literal in each variable-gadget into VC
2. No need to cover this literal again in clause-gadgets, as it is already covered above.
3. Put the remaining two unsatisfied literals in each clause-gadget in VC.
4. So we have a VC of size n+2c.

VC solution to 3SAT solution

Given a VC solution, create a 3SAT solution for P:

1. G has a VC of size n+2C, covering
   A. One variable in each variable-gadget: Make this TRUE.
   B. Two literals in each clause-gadget: Ignore them.
2. In each clause-gadget, the 3rd literal covered by variable-gadget is TRUE, hence all clauses are TRUE.
3. Hence P is True.

CLIQUE is NPC

3SAT to Clique polynomial-time reduction

We will reduce the 3SAT (3CNF-satisfiability) problem to the CLIQUE problem

Given a 3CNF formula:

literal

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge \underbrace{(x_3 \vee \overline{x_5} \vee x_6)}_{\text{clause}} \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6)$$

Each clause has three literals

3CNF-SAT is set of { w: w is a satisfiable 3CNF formula}

A 5-clique in graph G

CLIQUE = $\{< G, k >:$ graph G
contains a k-clique$\}$

Theorem: 3SAT is polynomial time reducible to CLIQUE

Proof: Give a polynomial time reduction of one problem to the other.
Transform formula to graph

3SAT to Clique reduction explained

- Given a 3CNF formula
- We will build a gadget (small graph) for each clause.
- Connect each literal in each clause to every other literal in other clauses
- Don't connect literal to its negation: x to !x
- All the true literals will form a clique.
- There is clique of size equal number of variables iff the formula is Satisfiable.

Transform formula to graph.

Example:

$$(x_1 \vee x_2 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4})$$

Create Nodes:

$$(x_1 \vee x_2 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4})$$

Add link from a literal ξ to a literal in every other clause, except the complement $\overline{\xi}$

$$(x_1 \vee x_2 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4})$$

Resulting Graph

$$(x_1 \vee x_2 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4}) = 1$$

$x_1 = 1$

$x_2 = 0$

$x_3 = 0$

$x_4 = 1$

The formula is satisfied if and only if the Graph has a 4-clique

End of Proof

Corollary: CLIQUE is NP-complete

Proof:

a. CLIQUE is in NP

b. 3SAT is NP-complete

c. 3SAT is polynomial reducible to CLIQUE

Clique and Independent Set are NPC

Example: Clique is NPC

- CLIQUE = { <G,k> | G is a graph with a clique of size k }
- A clique is a subset of vertices that are all connected
- Why is CLIQUE in NP?

Reduce 3SAT to Clique

- Given a 3SAT problem Φ, with k clauses
- Make a vertex for each literal.
- Connect each vertex to the literals in other clauses that are not its negations.
- Any k-clique in this graph corresponds to a satisfying assignment (see example in next slide)

3SAT as a clique problem

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge \quad \wedge$$

Example: Independent Set is NPC

- Defintion: An k-IS, is a k subset of k vertices of G with no edges between them.
- Decision problem: Does G have a k-IS?
- To Show k-IS is NPC
- Method: Reduce k-clique to k-IS

Reduce Clique to IS

- Independent set is the *dual problem* of Clique.
- Convert G to G^c (complement graph).
- G has a k-clique iff G^C has k-IS

G has four 3-clique, e.g see oval

G^c has four 3-IS, e.g. see oval

3SAT is NPC

SAT Reduces to 3SAT

Recall: 3CNF-SAT is a formula where each clause has 3 distinct literals.

Claim. CNF-SAT $\leq_P$ 3CNF-SAT.

Case 1: clause C_j contains only 1 term,
add 4 new terms, and replace C_j with 4 clauses:

$$C_j = \overline{x_3} \quad \Rightarrow \quad \begin{aligned} C'_{j1} &= \overline{x_3} \vee y_1 \vee y_2 \\ C'_{j2} &= \overline{x_3} \vee y_1 \vee \overline{y_2} \\ C'_{j3} &= \overline{x_3} \vee \overline{y_1} \vee y_2 \\ C'_{j4} &= \overline{x_3} \vee \overline{y_1} \vee \overline{y_2} \end{aligned}$$

SAT Reduces to 3SAT

- Case 2: clause C_j contains exactly 3 literals,
  - nothing to do.
- Case 3: clause C_j contains 2 literals:
  - add 1 new literal y, and replace C_j with 2 clauses as follows:

$$C_j = \overline{x_3} \vee x_7 \quad \Rightarrow \quad C'_{j1} = \overline{x_3} \vee x_7 \vee y$$
$$C'_{j2} = \overline{x_3} \vee x_7 \vee \overline{y}$$

Case 4 ?

SAT Reduces to 3SAT

Case 4: clause C_j contains $\ell \geq 4$ terms.

- introduce $\ell - 1$ extra Boolean variables
- replace C_j with ℓ clauses as follows:

$$C_j = x_1 \vee \overline{x_3} \vee \overline{x_4} \vee x_5 \vee x_6 \vee \overline{x_9} \Rightarrow \begin{array}{lcl} C'_{j1} & = & x_1 \vee x_1 \vee y_1 \\ C'_{j2} & = & \overline{y_1} \vee \overline{x_3} \vee y_2 \\ C'_{j3} & = & \overline{y_2} \vee \overline{x_4} \vee y_3 \\ C'_{j4} & = & \overline{y_3} \vee x_5 \vee y_4 \\ C'_{j5} & = & \overline{y_4} \vee x_6 \vee y_5 \\ C'_{j6} & = & \overline{y_5} \vee \overline{x_9} \vee \overline{x_9} \end{array}$$

SAT Reduces to 3SAT

Case 4: clause C_j contains $\ell \geq 4$ terms.
Suppose in LHS, C_j is SAT (true) because of t_{j4}, is true, then in RHS, each of $C'_{j1}..C'_{jl}$ can be made true as follows:

set False

$$C_j = t_{j1} \vee t_{j2} \vee t_{j3} \vee \cdots \vee t_{j\ell} \Rightarrow$$

$$\begin{array}{lclclcl}
C'_{j1} & = & t_{j1} & \vee & t_{j1} & \vee & y_1 \\
C'_{j2} & = & \overline{y_1} & \vee & t_{j2} & \vee & y_2 \\
C'_{j3} & = & \overline{y_2} & \vee & t_{j3} & \vee & y_3 \\
C'_{j4} & = & \overline{y_3} & \vee & \blacksquare & \vee & y_4 \\
C'_{j5} & = & \overline{y_4} & \vee & t_{j5} & \vee & y_5 \\
\vdots & & \vdots & & \vdots & & \vdots \\
C'_{j\ell} & = & \overline{y_{\ell-1}} & \vee & t_{j\ell} & \vee & t_{j\ell}
\end{array}$$

k = 4

set TRUE

SAT Reduces to 3SAT

Proof of case 4 $\Rightarrow$ Suppose SAT instance is satisfiable.

If SAT assignment sets $t_{jk} = 1$, then

3-SAT assignment sets:

- $t_{jk} = 1$
- $y_m = 1$ for all $m < k$;
- $y_m = 0$ for all $m \geq k$

$$C_j = t_{j1} \vee t_{j2} \vee t_{j3} \vee \cdots \vee t_{j\ell} \Rightarrow \begin{array}{lcl} C'_{j1} & = & \overline{t_{j1}} \vee t_{j1} \vee y_1 \\ C'_{j2} & = & \overline{y_1} \vee t_{j2} \vee y_2 \\ C'_{j3} & = & \overline{y_2} \vee t_{j3} \vee y_3 \\ C'_{j4} & = & \overline{y_3} \vee t_{j4} \vee y_4 \\ C'_{j5} & = & \overline{y_4} \vee t_{j5} \vee y_5 \\ \vdots & & \vdots \\ C'_{j\ell} & = & \overline{y_{\ell-1}} \vee t_{j\ell} \vee t_{j\ell} \end{array}$$

SAT Reduces to 3SAT

Case 4 Proof. $\Leftarrow$ Suppose 3-SAT instance is satisfiable, show SAT is true.

If 3SAT sets $t_{jk} = 1$ then

we claim $t_{jk} = 1$ for some k in C_j, because:

- each of ℓ - 1 new Boolean variables y_j can make ℓ-2 of new clauses true.
- ONE remaining clause must be satisfied by an original term t_{jk}

$$
\begin{array}{lcl}
C_j = t_{j1} \vee t_{j2} \vee t_{j3} \vee \cdots \vee t_{j\ell} & \Rightarrow & C'_{j1} = t_{j1} \vee t_{j1} \vee y_1 \\
 & & C'_{j2} = \overline{y_1} \vee t_{j2} \vee y_2 \\
 & & C'_{j3} = \overline{y_2} \vee t_{j3} \vee y_3 \\
 & & C'_{j4} = \overline{y_3} \vee t_{j4} \vee y_4 \\
 & & C'_{j5} = \overline{y_4} \vee t_{j5} \vee y_5 \\
 & & \vdots \qquad \vdots \qquad \vdots \qquad \vdots \\
 & & C'_{j\ell} = \overline{y_{\ell-1}} \vee t_{j\ell} \vee t_{j\ell}
\end{array}
$$

3Coloring is NPC

- 3Coloring a graph is NPC (hard).
- 2Coloring a graph is in P (easy).

Reduce 3SAT to 3Color

- Given a 3SAT formula S, we build a graph G, such at S is SAT iff G is 3-colorable.
- We use the colors { True , False, Blue } to color the graph

3-Coloring a triangle

If the top is colored blue, then there are only two possibilities for the lower nodes:

Case 1. x=True, !x=False

Case 2. x=False, !x=True

Reduce 3SAT to 3Color

- For the variables build this gadget. Color the center node Blue, connected to True and False above it, and the variables connected below it.
- 3 coloring this ensures that each variable x will be colored True or False only (and !x will be opposite of x).

3SAT to 3Color

- For each clause, build this gadget, with output connected to color True.
- The only ways to 3-**color** this is to use the **color** True for at least one of the vertices at the top (so each clause is true).
- Conversely, as long as at least one of top vertex is colored True, the rest of the graph can be 3-colored.

gadget for clause: (x1 or !x2 or !x3).

Example: $(\neg x1 \vee x2 \vee x4)$ v $(x1 \vee x2 \vee x4)$

Connect all the clauses' input to the literals in it,

and join its output to **True**

NP Complete Homework

ILP is Integer Linear Programming

- Give a set of variables {x1..xn} with values in integers.
- A set of m linear constraints of the form:
  - a11 * x1 + … a1n * xn cmp b1
  - am1 * x1 + … amn * xn cmp bm
  - Where a[1..m,1..n], b[1..m] are integers,
  - cmp is { <, >, >=, <=, =}
- Given an ILP problem, does it have an integer solution?

Q1. Reduce 3SAT to ILP

- Given 3CNF SAT with {b1..bn} boolean variables, and 3CNF formulas (b1 or b2 or b3)&….
- Write these as ILP constraints:
  1. b1 is 0 or 1 (true or false).
  2. b1 is opposite of !b1.
  3. Each 3CNF (b1 or b2 or b3) is 1 (true).
- Hence ILP is NPC

Q2. Reduce 3Color to ILP

- Encode color of each vertex as an integer {c1, c2, c3}.
- Add ILP constraints
  1. Each vertex is only one color.
  2. For each edge(v1, v2), the color(v1) != color(v2).
- Hence ILP is NPC.

Q3. Reduce 2Partition to 3Partition

- *2Partition*: Given a set of integer weights A={w1 .. wn}, to partition A into two sets of equal sums is NPC.
- *3Partition*: Divide A into 3 equal sets.
- Hint: To solve 2partition, add a dummy weight to A, and call 3Partition. Afterwards remove the dummy weight from the 3 partitions.

Java Example: Multiplying Big Integers

- Setup: Define BigInt class, and include essential parameters, including the prime P, and primitive root of unity, OMEGA.

```
import java.lang.*;
import java.math.*;
import java.util.*;

public class BigInt {
  protected int signum=0;                  // neg = -1, 0 = 0, pos = 1
  protected int[] mag;                     // magnitude in little-endian format
  public final static int MAXN=134217728;  // Maximum value for n
  public final static int ENTRYSIZE= 10;   // Bits per entry in mag
  protected final static long P=2013265921; // The prime 15*2^{27}+1
  protected final static int OMEGA=440564289; // Root of unity 31^{15} mod P
  protected final static int TWOINV=1006632961; // 2^{-1} mod P
```

Java Integer Multiply Method

- Use convolution to multiply two big integers, this and val:

```
public BigInt multiply(BigInt val) {
  int n = makePowerOfTwo(Math.max(mag.length,val.mag.length))*2;
  int signResult = signum * val.signum;
  int[] A = padWithZeros(mag,n);   // copies mag into A padded w/ 0's
  int[] B = padWithZeros(val.mag,n); // copies val.mag into B padded w/ 0's
  int[] root = rootsOfUnity(n);    // creates all n roots of unity
  int[] C = new int[n];            // result array for A*B
  int[] AF = new int[n];           // result array for FFT of A
  int[] BF = new int[n];           // result array for FFT of B
  FFT(A,root,n,0,AF);
  FFT(B,root,n,0,BF);
  for (int i=0; i<n; i++)
    AF[i] = (int)(((long)AF[i]*(long)BF[i]) % P);   // Component multiply
  reverseRoots(root);              // Reverse roots to create inverse roots
  inverseFFT(AF,root,n,0,C);       // Leaves inverse FFT result in C
  propagateCarries(C);             // Convert C to right no. bits per entry
  return new BigInt(signResult,C);
  }
```

Java FFT in Z^*_p

```
public static void FFT(int[] A, int[] root, int n, int base, int[] Y) {
  int prod;
  if (n==1) {
    Y[base] = A[base];
    return;
    }
  inverseShuffle(A,n,base);     // inverse shuffle to separate evens and odds
  FFT(A,root,n/2,base,Y);     // results in Y[base] to Y[base+n/2-1]
  FFT(A,root,n/2,base+n/2,Y); // results in Y[base+n/2] to Y[base+n-1]
  int j = A.length/n;
  for (int i=0; i<n/2; i++) {
    prod = (int)(((long)root[i*j]*Y[base+n/2+i]) % P);
    Y[base+n/2+i] = (int)(((long)Y[base+i] + P - prod) % P);
    Y[base+i] = (int)(((long)Y[base+i] + prod) % P);
    }
}
public static void inverseFFT(int[] A, int[] root, int n, int base, int[] Y) {
  int inverseN = modInverse(n);      // n^{-1}
  FFT(A,root,n,base,Y);
  for (int i=0; i<n; i++)
    Y[i] = (int)(((long)Y[i]*inverseN) % P);
}
```

Java FFT in Z^*_p

```
protected static int modInverse(int n) { // assumes n is power of two
  int result = 1;
  for (long twoPower = 1; twoPower < n; twoPower *= 2)
    result = (int)(((long)result*TWOINV) % P);
  return result;
}
protected static void inverseShuffle(int[] A, int n, int base) {
  int shift;
  int[] sp = new int[n];
  for (int i=0; i<n/2; i++) { // Unshuffle A into the scratch space
    shift = base + 2*i;
    sp[i] = A[shift];          // an even index
    sp[i+n/2] = A[shift+1];  // an odd index
    }
  for (int i=0; i<n; i++)
    A[base+i] = sp[i];         // copy back to A
}
```

```
protected static int[] rootsOfUnity(int n) { //assumes n is power of 2
  int t = MAXN;
  int nthroot = OMEGA;
  for (int t = MAXN; t>n; t /= 2)  // Find prim. nth root of unity
    nthroot = (int)(((long)nthroot*nthroot) % P);
  int[] roots = new int[n];
  int r = 1;          // r will run through all nth roots of unity
  for (int i=0; i<n; i++) {
    roots[i] = r;
    r = (int)(((long)r*nthroot) % P);
    }
  return roots;
}
protected static void propagateCarries(int[] A) {
  int i, carry;
  carry = 0;
  for (i=0; i<A.length; i++) {
    A[i] = A[i] + carry;
    carry = A[i] >>> ENTRYSIZE;
    A[i] = A[i] - (carry << ENTRYSIZE);
    }
  }
```

Experimental Results

- log/log graph shows traditional multiply runs in $O(n^2)$ time, while FFT versions are almost linear

Matrix computation in

Excel

Matrix Commands in Excel

Excel can perform some useful matrix operations:

- Addition & subtraction
- Scalar multiplication & division
- Transpose (TRANSPOSE)
- Matrix multiplication (MMULT)
- Matrix inverse (MINVERSE)
- Determinant of matrix (MDETERM)

As well as combinations of these operations.

Cells

- Most Excel formula require you to name one or more cell ranges e.g. B2:C3.
- You can type these in directly or select them using the mouse.
- However, it is often better to use a named range.

Naming group of Cells

- To assign a name to a range of cells, highlight it using the mouse and choose Insert > Name > Define and enter a name.
- Choose a useful name.
- Remember Excel does not distinguish between the names MATRIX1, Matrix1 and matrix1.

Entering a Matrix

- Choose a location for the matrix (or vector) and enter the elements of the matrix.
- Highlight the cells of the matrix and choose INSERT > NAME > DEFINE.
- Enter a name for the matrix.
- You can now use the name of the matrix in formulae.

Matrix +- *

To add two named 3 x 2 matrices A and B:

Highlight a blank 3 x 2 results area in the spreadsheet. (If the results area is too small, you will get the wrong answer.)

Type =A+B in the formula bar and press the CONTROL-SHIFT-ENTER keys simultaneously, to get the matrix result.

If you click on any cell in the result, the formula {=A+B} will be displayed, the { } brackets indicate a matrix (array) command.

Matrix Transpose

- Suppose B is a 3 x 2 matrix.
- B' = B transpose is a 2 x 3 matrix.
- Select a 2 x 3 results area, type =TRANSPOSE(A) in the formula bar and press CTRL-SHIFT-ENTER.
- Choose A and B so that AB exist. Check that (AB)' = B 'A', using MMULT.

Matrix Multiplication

- Suppose A and B are named 3 x 2 and 2 x 3 matrices.
- Then A*B is 3 x 3 and B*A is 2 x 2. In general, AB != BA.
- Select a blank 3 x 3 area for the result A*B.
- Type =MMULT(A,B) in the formula bar and press CTRL-SHIFT-ENTER to compute AB.

Matrix Inverse and determinant

- Suppose B is a square 2 x2 matrix.
- Select a 2 x 2 area for the inverse of B.
- Type =MINVERSE(B) in the formula bar and press CRTL-SHIFT-ENTER.
- Find |B|=determinant of B using =MDETERM(B)
- If determinant of B = |B|=0, it is not invertible.
- Let A and B be 3x3, and |A| != 0, |B| != 0, Compute and compare $(AB)^{-1} = B^{-1}A^{-1}$.

Linear Algebra

Outline

- Geometric intuition for linear algebra
- Matrices as linear transformations or as sets of constraints
- Linear systems & vector spaces
- Solving linear systems
- Eigenvalues & eigenvectors

Basic concepts

- *Vector* in R^n is an ordered set of n real numbers.
  - e.g. v = (1,6,3,4) is in R^4
  - "(1,6,3,4)" is a column vector:
  - as opposed to a row vector:
- m-by-n *matrix* is an object with m rows and n columns, each entry fill with a real number:

$$\begin{pmatrix} 1 \\ 6 \\ 3 \\ 4 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 6 & 3 & 4 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 8 \\ 4 & 78 & 6 \\ 9 & 3 & 2 \end{pmatrix}$$

Basic concepts

- Transpose: reflect vector/matrix on line:

$$\begin{pmatrix} a \\ b \end{pmatrix}^T = \begin{pmatrix} a & b \end{pmatrix} \qquad \begin{pmatrix} a & b \\ c & d \end{pmatrix}^T = \begin{pmatrix} a & c \\ b & d \end{pmatrix}$$

  – Note: $(Ax)^T = x^T A^T$ (We'll define multiplication soon…)
- Vector norms:
  – L_p norm of $v = (v_1,\ldots,v_k)$ is $(\Sigma_i |v_i|^p)^{1/p}$
  – Common norms: L_1, L_2
  – $L_{infinity} = \max_i |v_i|$
- Length of a vector v is $L_2(v)$

Basic concepts

- Vector dot product: $u \bullet v = \begin{pmatrix} u_1 & u_2 \end{pmatrix} \bullet \begin{pmatrix} v_1 & v_2 \end{pmatrix} = u_1 v_1 + u_2 v_2$
  - Note dot product of u with itself is the square of the length of u.
- Matrix product:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$AB = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

Vector products

- Dot product: $u \bullet v = u^T v = \begin{pmatrix} u_1 & u_2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = u_1 v_1 + u_2 v_2$

- Outer product:

$$uv^T = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \begin{pmatrix} v_1 & v_2 \end{pmatrix} = \begin{pmatrix} u_1 v_1 & u_1 v_2 \\ u_2 v_1 & u_2 v_2 \end{pmatrix}$$

Matrices as linear transformations

$$\begin{pmatrix} 5 & 0 \\ 0 & 5 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 5 \\ 5 \end{pmatrix}$$

(stretching)

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

(rotation)

Matrices as linear transformations

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

(reflection)

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

(projection)

$$\begin{pmatrix} 1 & c \\ 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x + cy \\ y \end{pmatrix}$$

(shearing)

Matrices as sets of constraints

$$x + y + z = 1$$

$$2x - y + z = 2$$

$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & -1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Special matrices

diagonal

$$\begin{pmatrix} & 0 & 0 \\ 0 & & 0 \\ 0 & 0 & \end{pmatrix}$$

$$\begin{pmatrix} a & b & c \\ 0 & d & e \\ 0 & 0 & f \end{pmatrix}$$

upper-triangular

tri-diagonal

$$\begin{pmatrix} a & 0 & 0 \\ b & c & 0 \\ d & e & f \end{pmatrix}$$

lower-triangular

$$\begin{pmatrix} & 0 & 0 \\ 0 & & 0 \\ 0 & 0 & \end{pmatrix}$$

`I` (identity matrix)

Vector spaces

- Formally, a *vector space* is a set of vectors which is closed under addition and multiplication by real numbers.
- A *subspace* is a subset of a vector space which is a vector space itself, e.g. the plane z=0 is a subspace of R^3 (It is essentially R^2.).
- We'll be looking at R^n and subspaces of R^n

Our notion of planes in R^3 may be extended to *hyperplanes* in R^n (of dimension n-1)

Note: subspaces must include the origin (zero vector).

Linear system and subspaces

$$\begin{pmatrix} 1 & 0 \\ 2 & 3 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

$$u \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} + v \begin{pmatrix} 0 \\ 3 \\ 3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

- Linear systems define certain subspaces
- Ax = b is solvable iff b may be written as a linear combination of the columns of A
- The set of possible vectors b forms a subspace called the *column space* of A

Linear system & subspaces

The set of solutions to "$Ax = 0$" forms a subspace called the *null space* of A.

$$\begin{pmatrix} 1 & 0 \\ 2 & 3 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

→ Null space: {(0,0)}

$$\begin{pmatrix} 1 & 0 & 1 \\ 2 & 3 & 5 \\ 1 & 3 & 4 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

→ Null space: {(c,c,-c)}

Linear independence and basis

- Vectors $v_1,\ldots,v_k$ are linearly independent if $(c_1v_1+\ldots+c_kv_k = 0)$ implies $(c_1=\ldots=c_k=0)$

i.e. the nullspace is the origin

$$\begin{pmatrix} | & | & | \\ v_1 & v_2 & v_3 \\ | & | & | \end{pmatrix}\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 \\ 2 & 3 \\ 1 & 3 \end{pmatrix}\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Recall nullspace contained only (u,v)=(0,0).
i.e. the columns are linearly independent.

Linear independence and basis

- If all vectors in a vector space may be expressed as linear combinations of $v_1,\ldots,v_k$, then $v_1,\ldots,v_k$ *span* the space.

$$\begin{pmatrix}2\\2\\2\end{pmatrix} = 2\begin{pmatrix}1\\0\\0\end{pmatrix} + 2\begin{pmatrix}0\\1\\0\end{pmatrix} + 2\begin{pmatrix}0\\0\\1\end{pmatrix}$$

$$\begin{pmatrix}2\\2\\2\end{pmatrix} = 1.57\begin{pmatrix}.9\\.2\\0\end{pmatrix} + 1.29\begin{pmatrix}.3\\1\\0\end{pmatrix} + 2\begin{pmatrix}.1\\.2\\1\end{pmatrix}$$

Linear independence and basis

- A *basis* is a set of linearly independent vectors which span the space.
- The *dimension* of a space is the # of “degrees of freedom” of the space; it is the number of vectors in any basis for the space.
- A basis is a maximal set of linearly independent vectors and a minimal set of spanning vectors.

$$\begin{pmatrix}2\\2\\2\end{pmatrix} = 2\begin{pmatrix}1\\0\\0\end{pmatrix} + 2\begin{pmatrix}0\\1\\0\end{pmatrix} + 2\begin{pmatrix}0\\0\\1\end{pmatrix}$$

$$\begin{pmatrix}2\\2\\2\end{pmatrix} = 1.57\begin{pmatrix}.9\\.2\\0\end{pmatrix} + 1.29\begin{pmatrix}.3\\1\\0\end{pmatrix} + 2\begin{pmatrix}.1\\.2\\1\end{pmatrix}$$

Linear independence and basis

- Two vectors are *orthogonal* if their dot product is 0.
- An *orthogonal basis* consists of orthogonal vectors.
- An *ortho-normal basis* consists of orthogonal vectors of unit length.

$$\begin{pmatrix}2\\2\\2\end{pmatrix} = 2\begin{pmatrix}1\\0\\0\end{pmatrix} + 2\begin{pmatrix}0\\1\\0\end{pmatrix} + 2\begin{pmatrix}0\\0\\1\end{pmatrix}$$

$$\begin{pmatrix}2\\2\\2\end{pmatrix} = 1.57\begin{pmatrix}.9\\.2\\0\end{pmatrix} + 1.29\begin{pmatrix}.3\\1\\0\end{pmatrix} + 2\begin{pmatrix}.1\\.2\\1\end{pmatrix}$$

About subspaces

- The *rank* of A is the dimension of the column space of A.
- It also equals the dimension of the *row space* of A (the subspace of vectors which may be written as linear combinations of the rows of A).

$$\begin{pmatrix} 1 & 0 \\ 2 & 3 \\ 1 & 3 \end{pmatrix}$$

(1,3) = (2,3) – (1,0)

Only 2 linearly independent rows, so rank = 2.

About subspaces

<u>Fundamental Theorem of Linear Algebra</u>:

If A is (m x n) matrix with rank r,

Column space(A) has dimension r

Nullspace(A) has dimension n-r (= *nullity* of A)

Row space(A) = Column space(A^T) has dimension r

Left nullspace(A) = Nullspace(A^T) has dimension m - r

<u>Rank-Nullity Theorem</u>: rank + nullity = n

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

m = 3
n = 2
r = 2

Null space, column space

- Null space – it is the orthogonal complement of the row space
- Every vector in this space is a solution to the equation, $Ax = 0$
- Rank – nullity theorem
- Column space
- Compliment of rank-nullity

Non-square matrices

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 2 & 3 \end{pmatrix}$$

m = 3
n = 2
r = 2
System Ax=b may not have a solution (x has 2 variables but 3 constraints).

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 2 & 3 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \end{pmatrix}$$

m = 2
n = 3
r = 2
System Ax=b is underdetermined (x has 3 variables and 2 constraints).

$$\begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Basis transformations

- Before talking about basis transformations, we need to recall matrix inversion and projections.

Matrix inversion

- To solve "Ax=b", we can write a closed-form solution if we can find a matrix A^{-1}

  s.t. $AA^{-1} = A^{-1}A = I$ (identity matrix)
- Then Ax=b iff $x = A^{-1}b$:

  $x = Ix = A^{-1}Ax = A^{-1}b$
- A is *non-singular* iff A^{-1} exists iff Ax=b has a unique solution.
- Note: If A^{-1}, B^{-1} exist, then $(AB)^{-1} = B^{-1}A^{-1}$,

  and $(A^T)^{-1} = (A^{-1})^T$

Projections

$$\begin{pmatrix} 2 \\ 2 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 2 \\ 2 \\ 2 \end{pmatrix}$$

$$c = \frac{a^T b}{a^T a} a = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

Basis transformations

We may write v=(2,2,2) in terms of an alternate basis:

$$\begin{pmatrix}2\\2\\2\end{pmatrix}=\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\begin{pmatrix}2\\2\\2\end{pmatrix}$$

$$\begin{pmatrix}2\\2\\2\end{pmatrix}=\begin{pmatrix}.9&.3&.1\\.2&1&.2\\0&0&1\end{pmatrix}\begin{pmatrix}1.57\\1.29\\2\end{pmatrix}$$

Components of (1.57,1.29,2) are projections of v onto new basis vectors, normalized so new v still has same length.

Basis transformations

Given vector v written in standard basis, rewrite as v_Q in terms of basis Q.

If columns of Q are orthonormal, $v_Q = Q^T v$

Otherwise, $v_Q = (Q^T Q) Q^T v$

Special matrices

- Matrix A is *symmetric* if $A = A^T$
- A is *positive definite* if "$x^T A x > 0$" for all non-zero x (*positive semi-definite* if inequality is not strict)
- Examples:

$$\begin{pmatrix} a & b & c \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} a \\ b \\ c \end{pmatrix} = a^2 + b^2 + c^2$$

$$\begin{pmatrix} a & b & c \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} a \\ b \\ c \end{pmatrix} = a^2 - b^2 + c^2$$

Special matrices

Note: Any matrix of form

$A^T A$ is positive semi-definite because,

$x^T(A^T A)x = (x^T A^T)(Ax) = (Ax)^T(Ax) \geq 0$

Determinants

- If det(A) = 0, then A is singular.
- If det(A) ≠ 0, then A is invertible.

$$\det\begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$$

Determinants

- m-by-n matrix A is *rank-deficient* if A has rank $r < m$ ($\leq n$)
- Theorem: $\text{rank}(A) < r$ iff

  $\det(A) = 0$ for all t-by-t submatrices,

  $r \leq t \leq m$

Eigenvalues & eigenvectors

- How can we characterize matrices?
- The solutions to "$Ax = \lambda x$" in the form of eigenpairs (λ,x) = (eigenvalue,eigenvector) where x is non-zero.
- To solve this, $(A - \lambda I)\, x = 0$
- λ is an eigenvalue iff $\det(A - \lambda I) = 0$
- The eigenvectors of a matrix are unit vectors that satisfy: $Ax = \lambda x$

Eigenvalues

(A – λI) x = 0

λ is an eigenvalue iff det(A – λI) = 0

Example:

$$A = \begin{pmatrix} 1 & 4 & 5 \\ 0 & 3/4 & 6 \\ 0 & 0 & 1/2 \end{pmatrix}$$

$$\det(A - \lambda I) = \begin{pmatrix} 1-\lambda & 4 & 5 \\ 0 & 3/4-\lambda & 6 \\ 0 & 0 & 1/2-\lambda \end{pmatrix} = (1-\lambda)(3/4-\lambda)(1/2-\lambda)$$

$$\lambda = 1, \lambda = 3/4, \lambda = 1/2$$

Eigenvector example

$$A = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$$

Eigenvalues λ = 2, 1 with eigenvectors (1,0), (0,1)

Eigenvectors of a linear transformation A are not rotated (but will be scaled by the corresponding eigenvalue) when A is applied.

Solving Ax=b

$$\begin{aligned} x + 2y + z &= 0 \\ y - z &= 2 \\ x \quad\quad +2z &= 1 \end{aligned}$$

$$\begin{pmatrix} 1 & 2 & 1 & 0 \\ 0 & 1 & -1 & 2 \\ 1 & 0 & 2 & 1 \end{pmatrix}$$

Write system of equations in matrix form.

$$\begin{aligned} x + 2y + z &= 0 \\ y - z &= 2 \\ -2y + z &= 1 \end{aligned}$$

$$\begin{pmatrix} 1 & 2 & 1 & 0 \\ 0 & 1 & -1 & 2 \\ 0 & -2 & 1 & 1 \end{pmatrix}$$

Subtract first row from last row.

$$\begin{aligned} x + 2y + z &= 0 \\ y - z &= 2 \\ - z &= 5 \end{aligned}$$

$$\begin{pmatrix} 1 & 2 & 1 & 0 \\ 0 & 1 & -1 & 2 \\ 0 & 0 & -1 & 5 \end{pmatrix}$$

Add 2 copies of second row to last row.

Now solve by back-substitution: z = -5, y = 2-z = -3, x = -2y-z = 11

Solving Ax=b & condition numbers

- Matlab: linsolve(A,b)
- How stable is the solution?
- If A or b are changed slightly, how much does it effect x?
- The *condition number* c of A measures this:

  $$c = \lambda_{max} / \lambda_{min}$$

- Values of c near 1 are good.

Linear Algebra in

Excel

Excel Grid

- Excel file have extension .xls or .xlsx (2007).
- Work book is a 2D matrix/grid of CELLs containing data
- Cells are then named by Column+Row, e.g. A1, B5, I2
- Draggable corner to copy cells

Columns are: A,B,C,D

Rows are: 1,2,3,4,…

Matrix: B5:B6;C5:C6

Sheets: 1,2,…

Solving Ax=b

$$\begin{aligned} x + 2y + z &= 0 \\ y - z &= 2 \\ x \quad\quad +2z &= 1 \end{aligned}$$

$$\begin{pmatrix} 1 & 2 & 1 & 0 \\ 0 & 1 & -1 & 2 \\ 1 & 0 & 2 & 1 \end{pmatrix}$$

Write system of equations in matrix form.

$$\begin{aligned} x + 2y + z &= 0 \\ y - z &= 2 \\ -2y + z &= 1 \end{aligned}$$

$$\begin{pmatrix} 1 & 2 & 1 & 0 \\ 0 & 1 & -1 & 2 \\ 0 & -2 & 1 & 1 \end{pmatrix}$$

Subtract first row from last row.

$$\begin{aligned} x + 2y + z &= 0 \\ y - z &= 2 \\ - z &= 5 \end{aligned}$$

$$\begin{pmatrix} 1 & 2 & 1 & 0 \\ 0 & 1 & -1 & 2 \\ 0 & 0 & -1 & 5 \end{pmatrix}$$

Add 2 copies of second row to last row.

Now solve by back-substitution: z = -5, y = 2-z = -3, x = -2y-z = 11

Using Excel

1. Enter the numbers (A, b),
2. Name the matrices (A) and arrays (x,b)
3. Call math and matrix functions on your data to compute x. $x = A^{-1}b$
4. Compute A^{-1}

Linear Algebra in Excel

- Type in your matrix into Excel cells
- Select matrix and give it a name, e.g. 'A', 'b', 'x'.
- Select the cells you want the result in.
  - click on 'fx' to enter a formula for the result,
  - find your function in the list
  - type in the symbolic arguments,
  - press `Control-Shift-Enter`, not 'OK'.

  if the result is a matrix.

Solving Ax=b in Excel

- Select x (E2:E4), click on fx
- Compute x=MINVERSE(A) * b
- press Control-Shift-Enter instead of ok.

LU Decomposition

LUD of a matrix is a method to solve a set of simultaneous linear equations

Which is better:

- Gauss Elimination
- LUD

$$[A]=[L][U]=\begin{bmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{bmatrix}\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

LUD Method

For a non-singular matrix [*A*], we can use Naive Gauss Elimination forward elimination steps and write A as

$$[A] = [L][U]$$

Where

$[L]$ = lower triangular matrix

$[U]$ = upper triangular matrix

$$L = \begin{bmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{bmatrix} \qquad [U] = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

How does LUD work?

If solving a set of linear equations	$[A][X] = [C]$
If $[A] = [L][U]$ then	$[L][U][X] = [C]$
Multiply by	$[L]^{-1}$
Which gives	$[L]^{-1}[L][U][X] = [L]^{-1}[C]$
Remember $[L]^{-1}[L] = [I]$ which leads to	$[I][U][X] = [L]^{-1}[C]$
Now, if $[I][U] = [U]$ then	$[U][X] = [L]^{-1}[C]$
Now, let	$[L]^{-1}[C]=[Z]$
Which ends with	$[L][Z] = [C]$ (1)
and	$[U][X] = [Z]$ (2)

LUD: How can this be used?

Given $[A][X] = [C]$ to solve

1. Decompose $[A]$ into $[L]$ and $[U]$

2. Solve $[L][Z] = [C]$ for $[Z]$

3. Solve $[U][X] = [Z]$ for $[X]$

Is LUD better than Gaussian Elimination?

To solve $[A][X] = [B]$

Table. Time taken by methods

Gaussian Elimination	LU Decomposition
$T\left(\frac{8n^3}{3}+12n^2+\frac{4n}{3}\right)$	$T\left(\frac{8n^3}{3}+12n^2+\frac{4n}{3}\right)$

where T = clock cycle time and n = size of the matrix

So both methods are equally efficient.

To find inverse of [A]

Time taken by Gaussian Elimination

$$= n\left(CT\,|_{FE} + CT\,|_{BS}\right)$$

$$= T\left(\frac{8n^4}{3} + 12n^3 + \frac{4n^2}{3}\right)$$

Time taken by LU Decomposition

$$= CT\,|_{LU} + n \times CT\,|_{FS} + n \times CT\,|_{BS}$$

$$= T\left(\frac{32n^3}{3} + 12n^2 + \frac{20n}{3}\right)$$

Table 1 Comparing computational times of finding inverse of a matrix using LU decomposition and Gaussian elimination.

n	10	100	1000	10000
$CT\|_{\text{inverse GE}} / CT\|_{\text{inverse LU}}$	3.28	25.83	250.8	2501

Method: [A] Decompose to [L] and [U]

$$[A] = [L][U] = \begin{bmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

[*U*] is the same as the coefficient matrix at the end of the forward elimination step.

[*L*] is obtained using the *multipliers* that were used in the forward elimination process

LUD Worked Example

Using the Forward Elimination Procedure of Gauss Elimination

$$A = \begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix}$$

Finding the [*U*] matrix

Using the Forward Elimination Procedure of Gauss Elimination

$$A = \begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix}$$

Step 1: $\frac{64}{25} = 2.56; \quad Row2 - Row1(2.56) = \begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 144 & 12 & 1 \end{bmatrix}$

$$\frac{144}{25} = 5.76; \quad Row3 - Row1(5.76) = \begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & -16.8 & -4.76 \end{bmatrix}$$

Finding the [U] Matrix

Matrix after Step 1: $\begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & -16.8 & -4.76 \end{bmatrix}$

Step 2: $\frac{-16.8}{-4.8} = 3.5; \quad Row3 - Row2(3.5) = \begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & 0 & 0.7 \end{bmatrix}$

$$[U] = \begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & 0 & 0.7 \end{bmatrix}$$

Finding the [*L*] matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{bmatrix}$$

Using the multipliers used during the Forward Elimination Procedure

From the first step of forward elimination

$$\begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix}$$

$$\ell_{21} = \frac{a_{21}}{a_{11}} = \frac{64}{25} = 2.56$$

$$\ell_{31} = \frac{a_{31}}{a_{11}} = \frac{144}{25} = 5.76$$

Finding the [L] Matrix

From the second step of forward elimination

$$\begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & -16.8 & -4.76 \end{bmatrix} \qquad \ell_{32} = \frac{a_{32}}{a_{22}} = \frac{-16.8}{-4.8} = 3.5$$

$$[L] = \begin{bmatrix} 1 & 0 & 0 \\ 2.56 & 1 & 0 \\ 5.76 & 3.5 & 1 \end{bmatrix}$$

Does [L][U] = [A]?

$$[L]*[U]=\begin{bmatrix} 1 & 0 & 0 \\ 2.56 & 1 & 0 \\ 5.76 & 3.5 & 1 \end{bmatrix} * \begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & 0 & 0.7 \end{bmatrix} =$$

Using LU Decomposition

Solve the following set of linear equations using LU Decomposition

$$\begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 106.8 \\ 177.2 \\ 279.2 \end{bmatrix}$$

find the [*L*] and [*U*] matrices

$$[A] = [L][U] = \begin{bmatrix} 1 & 0 & 0 \\ 2.56 & 1 & 0 \\ 5.76 & 3.5 & 1 \end{bmatrix} \begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & 0 & 0.7 \end{bmatrix}$$

Example

Set $[L][Z] = [C]$

$$\begin{bmatrix} 1 & 0 & 0 \\ 2.56 & 1 & 0 \\ 5.76 & 3.5 & 1 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 106.8 \\ 177.2 \\ 279.2 \end{bmatrix}$$

Solve for $[Z]$

$$z_1 = 10$$

$$2.56z_1 + z_2 = 177.2$$

$$5.76z_1 + 3.5z_2 + z_3 = 279.2$$

Example

Complete the forward substitution to solve for $[Z]$

$$
\begin{aligned}
z_1 &= 106.8 \\
z_2 &= 177.2 - 2.56 z_1 \\
&= 177.2 - 2.56(106.8) \\
&= -96.2 \\
z_3 &= 279.2 - 5.76 z_1 - 3.5 z_2 \\
&= 279.2 - 5.76(106.8) - 3.5(-96.21) \\
&= 0.735
\end{aligned}
$$

$$
[Z] = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 106.8 \\ -96.21 \\ 0.735 \end{bmatrix}
$$

Example

Set $[U][X] = [Z]$

$$\begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & 0 & 0.7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 106.8 \\ -96.21 \\ 0.735 \end{bmatrix}$$

Solve for $[X]$

The 3 equations become

$$25a_1 + 5a_2 + a_3 = 106.8$$

$$-4.8a_2 - 1.56a_3 = -96.21$$

$$0.7a_3 = 0.735$$

Now use backward substitution

From the 3rd equation

$$0.7a_3 = 0.735$$

$$a_3 = \frac{0.735}{0.7}$$

$$a_3 = 1.050$$

Substituting in a_3 and using the second equation

$$-4.8a_2 - 1.56a_3 = -96.21$$

$$a_2 = \frac{-96.21 + 1.56a_3}{-4.8}$$

$$a_2 = \frac{-96.21 + 1.56(1.050)}{-4.8}$$

$$a_2 = 19.70$$

To get the solution

Substituting in a_3 and a_2 using the first equation

$$25a_1 + 5a_2 + a_3 = 106.8$$

$$a_1 = \frac{106.8 - 5a_2 - a_3}{25}$$

$$= \frac{106.8 - 5(19.70) - 1.050}{25}$$

$$= 0.2900$$

Hence the Solution Vector is:

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 0.2900 \\ 19.70 \\ 1.050 \end{bmatrix}$$

Matrix Inverse

$$\begin{bmatrix} \cos 90^\circ & \sin 90^\circ \\ -\sin 90^\circ & \cos 90^\circ \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 \end{bmatrix}$$

Finding the inverse of a square matrix

The inverse [B] of a square matrix [*A*] is defined as

$$[A][B] = [I] = [B][A]$$

Finding the inverse of a square matrix

How can LU Decomposition be used to find the inverse?

Assume the first column of [*B*] to be $[b_{11} \;\; b_{21} \;\; \ldots \; b_{n1}]^T$

Using this and the definition of matrix multiplication

First column of [*B*]

$$[A]\begin{bmatrix} b_{11} \\ b_{21} \\ \vdots \\ b_{n1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Second column of [*B*] is

$$[A]\begin{bmatrix} b_{12} \\ b_{22} \\ \vdots \\ b_{n2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

The remaining columns in [*B*] can be found similarly

Example: Inverse of a Matrix

Find the inverse of a square matrix [*A*]

$$[A] = \begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix}$$

Using the decomposition procedure, the [*L*] and [*U*] matrices are found to be

$$[A] = [L][U] = \begin{bmatrix} 1 & 0 & 0 \\ 2.56 & 1 & 0 \\ 5.76 & 3.5 & 1 \end{bmatrix} \begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & 0 & 0.7 \end{bmatrix}$$

Example: Inverse of a Matrix

Solving for the each column of [*B*] requires two steps

- Solve [*L*] [*Z*] = [*C*] for [*Z*]
- Solve [*U*] [*X*] = [*Z*] for [*X*]

Step 1: $$[L][Z]=[C]\rightarrow\begin{bmatrix}1 & 0 & 0\\ 2.56 & 1 & 0\\ 5.76 & 3.5 & 1\end{bmatrix}\begin{bmatrix}z_1\\ z_2\\ z_3\end{bmatrix}=\begin{bmatrix}1\\ 0\\ 0\end{bmatrix}$$

This generates the equations:

$$z_1 = 1$$

$$2.56z_1 + z_2 = 0$$

$$5.76z_1 + 3.5z_2 + z_3 = 0$$

Example: Inverse of a Matrix

Solving for [*Z*]

$$z_1 = 1$$

$$z_2 = 0 - 2.56z_1$$

$$= 0 - 2.56(1)$$

$$= -2.56$$

$$z_3 = 0 - 5.76z_1 - 3.5z_2$$

$$= 0 - 5.76(1) - 3.5(-2.56)$$

$$= 3.2$$

$$[Z] = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -2.56 \\ 3.2 \end{bmatrix}$$

Example: Inverse of a Matrix

Solving [*U*][*X*] = [*Z*] for [*X*]

$$\begin{bmatrix} 25 & 5 & 1 \\ 0 & -4.8 & -1.56 \\ 0 & 0 & 0.7 \end{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix} = \begin{bmatrix} 1 \\ -2.56 \\ 3.2 \end{bmatrix}$$

$$25b_{11} + 5b_{21} + b_{31} = 1$$

$$-4.8b_{21} - 1.56b_{31} = -2.56$$

$$0.7b_{31} = 3.2$$

Example: Inverse of a Matrix

Using Backward Substitution

$$b_{31} = \frac{3.2}{0.7} = 4.571$$

$$b_{21} = \frac{-2.56 + 1.560 b_{31}}{-4.8}$$

$$= \frac{-2.56 + 1.560(4.571)}{-4.8} = -0.9524$$

$$b_{11} = \frac{1 - 5b_{21} - b_{31}}{25}$$

$$= \frac{1 - 5(-0.9524) - 4.571}{25} = 0.04762$$

So the first column of the inverse of [*A*] is:

$$\begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix} = \begin{bmatrix} 0.04762 \\ -0.9524 \\ 4.571 \end{bmatrix}$$

Example: Inverse of a Matrix

Repeating for the second and third columns of the inverse

Second Column

$$\begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix} \begin{bmatrix} b_{12} \\ b_{22} \\ b_{32} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} b_{12} \\ b_{22} \\ b_{32} \end{bmatrix} = \begin{bmatrix} -0.08333 \\ 1.417 \\ -5.000 \end{bmatrix}$$

Third Column

$$\begin{bmatrix} 25 & 5 & 1 \\ 64 & 8 & 1 \\ 144 & 12 & 1 \end{bmatrix} \begin{bmatrix} b_{13} \\ b_{23} \\ b_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} b_{13} \\ b_{23} \\ b_{33} \end{bmatrix} = \begin{bmatrix} 0.03571 \\ -0.4643 \\ 1.429 \end{bmatrix}$$

Example: Inverse of a Matrix

The inverse of [*A*] is

$$[A]^{-1} = \begin{bmatrix} 0.04762 & -0.08333 & 0.03571 \\ -0.9524 & 1.417 & -0.4643 \\ 4.571 & -5.000 & 1.429 \end{bmatrix}$$

To check your work do the following operation

$$[A][A]^{-1} = [I] = [A]^{-1}[A]$$

The Assignment Problem

The Problem

- Three air conditioners need to be installed in the same week by three different companies
- Bids for each job are solicited from each company
- To which company should each job be assigned?

	Bldg 1	Bldg 2	Bldg3
A	53	96	37
B	47	87	41
C	60	92	36

Naive Solution

- There are only 6 possible assignments of companies to jobs
- Check them and compare

Naïve Solution—Guess #1

	Bldg 1	Bldg 2	Bldg3
A	53	96	37
B	47	87	41
C	60	92	36

Total Cost = 53 + 87+ 36 = 176

Naïve Solution—Guess #2

	Bldg 1	Bldg 2	Bldg3
A	53	96	37
B	47	87	41
C	60	92	36

Total Cost = 53 + 92+ 41 = 186

Naïve Solution—Guess #3

	Bldg 1	Bldg 2	Bldg3
A	53	96	37
B	47	87	41
C	60	92	36

Total Cost = 47 + 96 + 36 = 179

Naïve Solution—Guess #4

	Bldg 1	Bldg 2	Bldg3
A	53	96	37
B	47	87	41
C	60	92	36

Total Cost = 47 + 92 + 37 = 176

Naïve Solution—Guess #5

	Bldg 1	Bldg 2	Bldg3
A	53	96	37
B	47	87	41
C	60	92	36

Total Cost = 47 + 96 + 41 = 197

Naïve Solution—Guess #6

	Bldg 1	Bldg 2	Bldg3
A	53	96	37
B	47	87	41
C	60	92	36

Total Cost = 60 + 87 + 37 = 184

Naïve Solution—Completion

	1	2	3
A	53	96	37
B	47	87	41
C	60	92	36

	1	2	3
A	53	96	37
B	47	87	41
C	60	92	36

	1	2	3
A	53	96	37
B	47	87	41
C	60	92	36

	1	2	3
A	53	96	37
B	47	87	41
C	60	92	36

	1	2	3
A	53	96	37
B	47	87	41
C	60	92	36

	1	2	3
A	53	96	37
B	47	87	41
C	60	92	36

Disadvantages of Naïve Solution

- How does the time-to-solution vary with problem size?
- Answer: O(n!)

Rates of Growth

n	log(n)	n	n^2	e^n	n!
1	0.00000	1	1	2.7	1
2	0.30103	2	4	7.4	2
3	0.47712	3	9	20.1	6
4	0.60206	4	16	54.6	24
5	0.69897	5	25	148.4	120
6	0.77815	6	36	403.4	720
7	0.84510	7	49	1096.6	5040
8	0.90309	8	64	2981.0	40320
9	0.95424	9	81	8103.1	362880
10	1.00000	10	100	22026.5	3628800
11	1.04139	11	121	59874.1	39916800
12	1.07918	12	144	162754.8	479001600
13	1.11394	13	169	442413.4	6227020800
14	1.14613	14	196	1202604.3	8.7178E+10
15	1.17609	15	225	3269017.4	1.3077E+12

Mathematical Modeling of the Problem

- Given a *Cost Matrix* C which lists for each "company" *i* the "cost" of doing "job" *j*.
- Solution is a *permutation matrix* X : all zeros except for one 1 in each row and column
- Objective is to minimize the total cost

$$c_{\text{total}} = \sum_{i,j=1}^{n} c_{ij} x_{ij}$$

An Ideal Cost Matrix

- All nonnegative entries
- An possible assignment of zeroes, one in each row and column
- In this case the minimal cost is apparently zero!

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & 10 \\ 8 & 0 & 0 \end{bmatrix}$$

The Hungarian Algorithm

- Find an “ideal” cost matrix that has the same optimal assignment as the given cost matrix
- From there the solution is easy!

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

Critical Observation

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

- Let C be a given cost matrix and consider a new cost matrix C’ that has the same number added to each entry of a single row of C
- For each assignment, the new total cost differs by that constant
- The optimal assignment is the same as before

Critical Observation

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

- Same is true of columns
- So: we can subtract *minimum* entry from each row and column to insure nonnegative entries

On our given matrix

$$\begin{bmatrix} 53 & 96 & 37 \\ 47 & 87 & 41 \\ 60 & 92 & 36 \end{bmatrix} \mapsto \begin{bmatrix} 16 & 59 & 0 \\ 6 & 46 & 0 \\ 24 & 56 & 0 \end{bmatrix} \mapsto \begin{bmatrix} 10 & 13 & 0 \\ 0 & 0 & 0 \\ 18 & 10 & 0 \end{bmatrix}$$

Still Not Done

- No assignment of zeros in this matrix

$$\begin{bmatrix} 10 & 13 & 0 \\ 0 & 0 & 0 \\ 18 & 10 & 0 \end{bmatrix}$$

Still Not Done

- No assignment of zeros in this matrix

$$\begin{bmatrix} 10 & 13 & 0 \\ 0 & 0 & 0 \\ 18 & 10 & 0 \end{bmatrix}$$

Still Not Done

- No assignment of zeros in this matrix

$$\begin{bmatrix} 10 & 13 & 0 \\ 0 & 0 & 0 \\ 18 & 10 & 0 \end{bmatrix}$$

Still Not Done

- No assignment of zeros in this matrix

$$\begin{bmatrix} 10 & 13 & 0 \\ 0 & 0 & 0 \\ 18 & 10 & 0 \end{bmatrix}$$

Still Not Done

- No assignment of zeros in this matrix

$$\begin{bmatrix} 10 & 13 & 0 \\ 0 & 0 & 0 \\ 18 & 10 & 0 \end{bmatrix}$$

?

Still Not Done

- No assignment of zeros in this matrix
- Still, we can create new zeroes by subtracting the smallest entry from some rows

$$\begin{bmatrix} 0 & 3 & -10 \\ 0 & 0 & 0 \\ 8 & 0 & -10 \end{bmatrix}$$

Still Not Done

- No assignment of zeros in this matrix
- Still, we can create new zeroes by subtracting the smallest entry from some rows
- Now we can preserve nonnegativity by adding that entry to columns which have negative entries

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & 10 \\ 8 & 0 & 0 \end{bmatrix}$$

Solutions

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & 10 \\ 8 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & 10 \\ 8 & 0 & 0 \end{bmatrix}$$

Naïve Solution—Completion

	1	2	3
A	53	96	37
B	47	87	41
C	60	92	36

	1	2	3
A	53	96	37
B	47	87	41
C	60	92	36

	1	2	3
A	53	96	37
B	47	87	41
C	60	92	36

	1	2	3
A	53	96	37
B	47	87	41
C	60	92	36

	1	2	3
A	53	96	37
B	47	87	41
C	60	92	36

	1	2	3
A	53	96	37
B	47	87	41
C	60	92	36

The Hungarian Algorithm

1. Find the minimum entry in each row and subtract it from each row
2. Find the minimum entry in each column and subtract it from each column

- Resulting matrix is nonnegative

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

The Hungarian Algorithm

3. Using lines that go all the way across or all the way up-and-down, cross out all zeros in the new cost matrix

- Find a way to do this with a minimum number of lines (≤n)

$$\begin{bmatrix} 10 & 13 & 0 \\ 0 & 0 & 0 \\ 18 & 10 & 0 \end{bmatrix}$$

The Hungarian Algorithm

4. If you can only do this with n lines, an assignment of zeroes is possible.

$$\begin{bmatrix} 10 & 13 & 0 \\ 0 & 0 & 0 \\ 18 & 10 & 0 \end{bmatrix}$$

The Hungarian Algorithm

5. Otherwise, determine the smallest entry not covered by any line.

- Subtract this entry from all uncovered entries
- Add it to all double-covered entries
- Return to Step 3.

$$\begin{bmatrix} 0 & 3 & 1 & 3 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 8 & 3 & 1 & 0 & -1 & 0 \end{bmatrix}$$

The Hungarian Algorithm

3. Using lines that go all the way across or all the way up-and-down, cross out all zeros in the new cost matrix

$$\begin{bmatrix} \cancel{0} & 3 & \cancel{0} \\ \cancel{0} & \cancel{0} & 10 \\ \cancel{8} & \cancel{3} & \cancel{0} \end{bmatrix}$$

The Hungarian Algorithm

3. Using lines that go all the way across or all the way up-and-down, cross out all zeros in the new cost matrix

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & 10 \\ 8 & 3 & 0 \end{bmatrix}$$

The Hungarian Algorithm

3. Using lines that go all the way across or all the way up-and-down, cross out all zeros in the new cost matrix

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & 10 \\ 8 & 3 & 0 \end{bmatrix}$$

The Hungarian Algorithm

4. If you can only do this with n lines, an assignment of zeroes is possible.

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & 10 \\ 8 & 3 & 0 \end{bmatrix}$$

Solutions

$$\begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & 10 \\ 8 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & 10 \\ 8 & 0 & 0 \end{bmatrix}$$

Example 2

- A cab company gets four calls from four customers simultaneously
- Four cabs are out in the field at varying distance from each customer
- Which cab should be sent where to minimize total (or average) waiting time?

		Customer			
		1	2	3	4
Cab	A	9	7.5	7.5	8
	B	3.5	8.5	5.5	6.5
	C	12.5	9.5	9.0	10.5
	D	4.5	11.0	9.5	11.5

Integerizing the Matrix

$$\begin{bmatrix} 9 & 7.5 & 7.5 & 8 \\ 3.5 & 8.5 & 5.5 & 6.5 \\ 12.5 & 9.5 & 9 & 10.5 \\ 4.5 & 10 & 9.5 & 11.5 \end{bmatrix} \mapsto \begin{bmatrix} 90 & 75 & 75 & 80 \\ 35 & 85 & 55 & 65 \\ 125 & 95 & 90 & 105 \\ 45 & 10 & 95 & 115 \end{bmatrix}$$

Non-negativizing the Matrix

$$\begin{bmatrix} 90 & 75 & 75 & 80 \\ 35 & 85 & 55 & 65 \\ 125 & 95 & 90 & 105 \\ 45 & 10 & 95 & 115 \end{bmatrix} \mapsto \begin{bmatrix} 15 & 0 & 0 & 5 \\ 0 & 50 & 20 & 30 \\ 35 & 5 & 0 & 15 \\ 0 & 65 & 50 & 70 \end{bmatrix} \mapsto \begin{bmatrix} 15 & 0 & 0 & 0 \\ 0 & 50 & 20 & 25 \\ 25 & 5 & 0 & 10 \\ 0 & 65 & 50 & 65 \end{bmatrix}$$

Covering the Zeroes

$$\begin{bmatrix} 15 & 0 & 0 & 0 \\ 0 & 50 & 20 & 25 \\ 25 & 5 & 0 & 10 \\ 0 & 65 & 50 & 65 \end{bmatrix}$$

Can do it with three!

Find Smallest Uncovered Entry

$$\begin{bmatrix} 15 & 0 & 0 & 0 \\ 0 & 50 & 20 & 25 \\ 25 & 5 & 0 & 10 \\ 0 & 65 & 50 & 65 \end{bmatrix}$$

Subtract and Add

$$\begin{bmatrix} 30 & 0 & 0 & 0 \\ 0 & 30 & 0 & 5 \\ 5 & 5 & 0 & 10 \\ 0 & 45 & 30 & 45 \end{bmatrix}$$

Cover Again

$$\begin{bmatrix} 30 & 0 & 0 & 0 \\ 0 & 30 & 0 & 5 \\ 5 & 5 & 0 & 10 \\ 0 & 45 & 30 & 45 \end{bmatrix}$$

Still three!

Find Smallest Uncovered

$$\begin{bmatrix} 30 & 0 & 0 & 0 \\ 0 & 30 & 0 & 5 \\ 55 & 5 & 0 & 10 \\ 0 & 45 & 30 & 45 \end{bmatrix}$$

Subtract and Add

$$\begin{bmatrix} 40 & 0 & 5 & 0 \\ 0 & 25 & 0 & 0 \\ 55 & 0 & 0 & 5 \\ 0 & 40 & 30 & 40 \end{bmatrix}$$

Cover Again

$$\begin{bmatrix} \cancel{40} & \cancel{0} & \cancel{5} & \cancel{0} \\ \cancel{0} & \cancel{25} & \cancel{0} & \cancel{0} \\ \cancel{55} & \cancel{0} & \cancel{0} & \cancel{5} \\ \cancel{0} & \cancel{40} & \cancel{30} & \cancel{40} \end{bmatrix}$$

Done!

Solutions

$$\begin{bmatrix} 40 & 0 & 5 & 0 \\ 0 & 25 & 0 & 0 \\ 55 & 0 & 0 & 5 \\ 0 & 40 & 30 & 40 \end{bmatrix} \qquad \begin{bmatrix} 40 & 0 & 5 & 0 \\ 0 & 25 & 0 & 0 \\ 55 & 0 & 0 & 5 \\ 0 & 40 & 30 & 40 \end{bmatrix}$$

Done!

Solutions

$$\begin{bmatrix} 9 & 7.5 & 7.5 & 8 \\ 3.5 & 8.5 & 5.5 & 6.5 \\ 12.5 & 9.5 & 9 & 10.5 \\ 4.5 & 10 & 9.5 & 11.5 \end{bmatrix} \quad \begin{bmatrix} 9 & 7.5 & 7.5 & 8 \\ 3.5 & 8.5 & 5.5 & 6.5 \\ 12.5 & 9.5 & 9 & 10.5 \\ 4.5 & 10 & 9.5 & 11.5 \end{bmatrix}$$

Total Cost = 27.5

Other Applications of AP

- Assigning teaching fellows to time slots
- Assigning airplanes to flights
- Assigning project members to tasks
- Determining positions on a team
- Assigning brides to grooms (called the *marriage problem*)

Outline

- The Assignment Problem
- Bipartite Graphs and Matching
- Network Flow
- Hungarian Algorithm
- Example

History

- Two Hungarian mathematicians: Dénes König (1936) and Jenő Egerváry (1931)

- Harold W. Kuhn, "The Hungarian Method for the assignment problem", *Naval Research Logistic Quarterly*, **2**:83-97, 1955.

- J. Munkres, "Algorithms for the Assignment and Transportation Problems", *Journal of the Society of Industrial and Applied Mathematics*, **5**(1):32-38, 1957.

The Assignment Problem

The Simple Assignment Problem

- Four individuals (i=1, 2, 3, 4)
- Four jobs (j=1, 2, 3, 4)

Individual	Qualified job(s)
1	1, 2, and 3
2	3 and 4
3	4
4	4

Qualification Matrix → $Q = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

The Simple Assignment Problem (Cont.)

What is the largest number of jobs that can be assigned to qualified individuals (with not more than one job assigned to each individual)?

What is the largest number of 1's that can be chosen from Q with no two chosen from the same row or column?

$$Q = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Simple Assignment Problem (Cont.)

- Start from an assignment

$$\begin{bmatrix} 1 & 1 & 1^* & 0 \\ 0 & 0 & 1 & 1^* \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Impossible to improve → "Complete"

Transfer 1

Transfer 2

$$\begin{bmatrix} 1^* & 1 & 1 & 0 \\ 0 & 0 & 1^* & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

"Incomplete"

$$\begin{bmatrix} 1^* & 1 & 1 & 0 \\ 0 & 0 & 1^* & 1 \\ 0 & 0 & 0 & 1^* \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

New Assignment

Bipartite Graphs

& The Matching Problem

Bipartite Graph

$$Q = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1^* & 0 \\ 0 & 0 & 0 & 1^* \\ 0 & 0 & 1^* & 1 \\ 0 & 0 & 0 & 1^* \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

*Characterizing Bipartite Graphs

- Theorem. Let G be a graph with at least 2 vertices. The following statements about G are equivalent:
  - 1. G is bipartite.
  - 2. G can be properly 2-colored.
  - 3. G has no odd cycles.

*Applications of Bipartite Graphs

- <u>Personnel Assignment Problem</u>
  - A company has workers $X_1, \ldots, X_m$ and jobs $Y_1, \ldots, Y_n$. Each worker is qualified to do some jobs, but not others.
  - Can every worker be assigned a job?
- <u>Optimal Assignment Problem</u>
  - Same basic setup as above, but now each pair (X_i, Y_j) is given a weighting w_{ij} indicated the 'effectiveness' (e.g. profit to company) of assigning worker X_i to job Y_j.
  - How should jobs be assigned to maximize the total effectiveness of the assignments?
- <u>Marriage Problem</u>
  - There are k men and m women, and each male-female pair has expressed whether or not they are willing to marry.
  - How can we pair them up so that all the men are paired with acceptable mates (or the gender-reversed question)?

*Matchings

- All three problems involve forming a ***matching*** in a bipartite graph:
- Definition: Let G be a graph with $\{V_1, V_2\}$. A matching in G is a set of edges, no two of which share an endpoint.
- Note: ***G does not need to bipartite***, but in applications it often is.

Maximum and Perfect Matchings

- A matching M is ***maximum*** if it has the largest size among all possible matchings.

- A matching M is ***perfect*** if every vertex in G is incident with an edge in the matching.
  - Does maximum imply perfect?
  - Does perfect imply maximum?

*M-alternating path

- Given a matching M, a M-alternating path is a path that alternates between edges in M and edges not in M

*M-augmenting path

- An M-alternating path whose endpoints are unsaturated by M.

- M-augmenting paths can be used to enlarge matchings.

Berge's Theorem

- Berge's Theorem: A matching M is maximum if and only if it has no M-augmenting paths.

Formulating - Simple Assignment

- Decision variable

$$Q = \begin{bmatrix} 1^* & 1 & 1 & 0 \\ 0 & 0 & 1^* & 1 \\ 0 & 0 & 0 & 1^* \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- Let A be the set of allowed assignments

$$x_{ij} = \begin{cases} 1 & \text{if i is assigned to j} \\ 0 & \text{otherwise} \end{cases}$$

$$\boxed{\begin{array}{ll} \text{maximize} & \displaystyle\sum_{(i,j)\text{ in } A} q_{ij} x_{ij} \\ \text{s.t.} & \displaystyle\sum_{j:(i,j)\text{ in } A} x_{ij} \le 1, \quad \text{for all } i \\ & \displaystyle\sum_{i:(i,j)\text{ in } A} x_{ij} \le 1, \quad \text{for all } j \\ & x_{ij} = 0,1 \quad \text{for all } (i,j) \text{ in } A \end{array}}$$

Network Flow

Matching as Network Flow

- Bipartite Graph
- Network Flow
- Augmentation Graph

$$Q = \begin{bmatrix} 1^* & 1 & 1 & 0 \\ 0 & 0 & 1^* & 1 \\ 0 & 0 & 0 & 1^* \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The General Assignment Problem

- n individuals (i=1, 2, …, n)
- n jobs (j=1, 2, …, n)

- cost c_{ij}, cost of individual i to do job j
- How can we assign the jobs to individuals to minimize the total cost?

- Rating r_{ij} indicating the quality of work
- How can we assign the jobs to individuals to maximize the total rating?

Formulating- General Assignment

- Decision variable

$$x_{ij} = \begin{cases} 1 & \text{if i is assigned to j} \\ 0 & \text{otherwise} \end{cases}$$

- Let A be the set of allowed assignments and c_{ij} be the cost of assigning i to j.

$$\begin{aligned} \text{minimize} \quad & \sum_{(i,j)\text{ in } A} c_{ij}.x_{ij} \\ \text{s.t.} \quad & \sum_{j:(i,j)\text{ in } A} x_{ij} \le 1, \quad \text{for all } i \\ & \sum_{i:(i,j)\text{ in } A} x_{ij} \le 1, \quad \text{for all } j \\ & x_{ij} = 0,1 \quad \text{for all } (i,j) \text{ in } A \end{aligned}$$

*Optimization Problem

Four machines are available to process four jobs.

The processing time for each machine/job assignment is as follows:

machine \ job	1	2	3	4
A	4	6	5	5
B	7	4	5	6
C	4	7	6	4
D	5	3	4	7

What is the assignment (one job per machine) which will minimize total processing time?

Min.

$$4X_{11}+6X_{12}+5X_{13}+5X_{14}$$
$$+7X_{21}+4X_{22}+5X_{23}+6X_{24}$$
$$+4X_{31}+7X_{32}+6X_{33}+4X_{34}$$
$$+5X_{41}+3X_{42}+4X_{43}+7X_{44}$$

St.

$$X_{11}+X_{12}+X_{13}+X_{14}=1$$
$$X_{21}+X_{22}+X_{24}+X_{24}=1$$
$$X_{31}+X_{32}+X_{33}+X_{34}=1$$
$$X_{41}+X_{42}+X_{43}+X_{44}=1$$
$$X_{11}+X_{21}+X_{31}+X_{41}=1$$
$$X_{12}+X_{22}+X_{32}+X_{42}=1$$
$$X_{13}+X_{23}+X_{33}+X_{43}=1$$
$$X_{14}+X_{24}+X_{34}+X_{44}=1$$

Network Flow

Knowing the following capacities, what is the maximum flow from source to sink?

	Red 1	**Red 2**	**Red 3**	**Red 4**
Blue 1	2	5	3	4
Blue 2	13	2	4	5
Blue 3	4	3	8	3
Blue 4	13	6	4	3

Augmentation Graphs -General case

Flow Graph

Augmentation Graph

Flow Graph

Augmentation Graph

The Hungarian Algorithm

Why Hungarian?

- Bipartite graph G with V nodes and E edges

- The Hungarian algorithm: $O(V^3)$
- The Network Flow algorithm: $O(V.E^2)$

Example

- We must determine how jobs should be assigned to machines to **minimize** setup times, which are given below:

		Job 1	**Job 2**	**Job 3**	**Job 4**
Machine 1		14	5	8	7
Machine 2		2	12	6	5
Machine 3		7	8	3	9
Machine 4		2	4	6	10

Hungarian Algorithm

- Two Observations
  - Adding a constant to any row or column does not change the solution → Changing C
  - If C is nonnegative and $\sum c_{ij}x_{ij} = 0$ then X is a solution.
- Let 2 zeroes in C be called independent if they appear in different rows and columns.

Hungarian Theorem

- A set of elements of a matrix are said to be ‘independent’ if no two of them lie in the same row or column.

- König Theorem:

  If C is a matrix and m is the number of independent zero elements of C, then there are m lines which contain all the zero elements of C.

Hungarian Algorithm

1. From each line (row or column) subtract its minimum element.
2. Find a maximum set of N′ mutually independent zeroes.
3. *if* N′ = N such zeroes have been found: output their indices and stop
   otherwise: cover all zeroes in W with N′ lines and find the minimum uncovered value; subtract it from all uncovered elements, and add it to all doubly covered elements; go to 2.

Example

- We must determine how jobs should be assigned to machines to **minimize** setup times, which are given below:

		Job 1	**Job 2**	**Job 3**	**Job 4**
Machine 1		14	5	8	7
Machine 2		2	12	6	5
Machine 3		7	8	3	9
Machine 4		2	4	6	10

Hungarian Algorithm

- **<u>Step 1</u>**: (a) Find the minimum element in each row of the cost matrix. Form a new matrix by subtracting this cost from each row. (b) Find the minimum cost in each column of the new matrix, and subtract this from each column. This is the *reduced cost matrix*.

Example: Step 1(a)

		Job 1	Job 2	Job 3	Job 4
Machine 1		14	**5**	8	7
Machine 2		**2**	12	6	5
Machine 3		7	8	**3**	9
Machine 4		**2**	4	6	10

Row Reduction

		Job 1	Job 2	Job 3	Job 4
Machine 1		9	0	3	2
Machine 2		0	10	4	3
Machine 3		4	5	0	6
Machine 4		0	2	4	8

Example: Step 1(b)

		Job 1	**Job 2**	**Job 3**	**Job 4**
Machine 1		9	**0**	3	**2**
Machine 2		**0**	10	4	3
Machine 3		4	5	**0**	6
Machine 4		**0**	2	4	8

Column Reduction

		Job 1	**Job 2**	**Job 3**	**Job 4**
Machine 1		9	0	3	0
Machine 2		0	10	4	1
Machine 3		4	5	0	4
Machine 4		0	2	4	6

Hungarian Algorithm

- **<u>Step 2</u>**: Draw the minimum number of lines that are needed to cover all the zeros in the reduced cost matrix. If m lines are required, then an optimal solution is available among the covered zeros in the matrix. Otherwise, continue to Step 3.

How do we find the minimum number of lines?!

Example: Step 2

		Job 1	**Job 2**	**Job 3**	**Job 4**
Machine 1		9	0	3	0
Machine 2		0	10	4	1
Machine 3		4	5	0	4
Machine 4		0	2	4	6

We need 3<4 lines, so continue to Step 3

Hungarian Algorithm

- **<u>Step 3</u>**: Find the smallest nonzero element (say, k) in the reduced cost matrix that is uncovered by the lines. Subtract k from each uncovered element, and add k to each element that is covered by two lines. Return to Step 2.

Example: Step 3

		Job 1	Job 2	Job 3	Job 4
Machine 1		9	0	3	0
Machine 2		0	10	4	**1**
Machine 3		4	5	0	4
Machine 4		0	2	4	6

		Job 1	Job 2	Job 3	Job 4
Machine 1		**10**	0	3	0
Machine 2		0	**9**	**3**	**0**
Machine 3		**5**	5	0	4
Machine 4		0	**1**	**3**	**5**

Example: Step 2 (again)

Zero Assignment

		Job 1	Job 2	Job 3	Job 4
Machine 1		**10**	0	3	0
Machine 2		0	**9**	**3**	**0**
Machine 3		**5**	5	0	4
Machine 4		0	**1**	**3**	**5**

Need 4 lines, so we have the optimal assignment and we stop

Example: Final Solution

		Job 1	**Job 2**	**Job 3**	**Job 4**
Machine 1		10	**0**	3	**0**
Machine 2		**0**	9	3	**0**
Machine 3		5	5	**0**	4
Machine 4		**0**	1	3	5

Optimal assignment

$$x_{12} = 1, x_{33} = 1, x_{41} = 1, x_{24} = 1$$

How did we know which 0's to choose?!

Munkres Contribution

- Providing a constructive procedure for finding
  1) A minimal set of lines which contain all zeros,
  2) A maximal set of independent zeros

- "Starred zeros" and "Primed zeros"
- Alternating sequence between 0* and 0'

LP: Linear Programming

LP: Linear Programming

- Large proportion of all scientific and financial computations
- What is LP: Finding optimal solution under linear constraints.
- LP *is used to allocate resources,*
  - *plan production,*
  - *schedule workers,*
  - *plan investment portfolios and*
  - *formulate marketing (and military) strategies.*
- The versatility and economic impact of LP in today's industrial world is truly astounding.

Important Examples

- *Simplex method*
- *Ford-Fulkerson algorithm for maximum flow problem*
- *Maximum matching of graph vertices*
- *Gale-Shapley algorithm for the stable marriage problem*

Example

maximize	**$3x + 5y$**
subject to	**$x + y \le 4$**
	$x + 3y \le 6$
$x \ge 0,\ y \ge 0$	

Feasible region **is the set of points defined by the constraints**

Geometric solution

maximize	**$3x + 5y$**
subject to	**$x + y \le 4$**
	$x + 3y \le 6$
$x \ge 0,\ y \ge 0$	

Extreme Point Theorem **Any LP problem with a nonempty bounded feasible region has an optimal solution;**

moreover, an optimal solution can always be found at an *extreme point* of the problem's feasible region.

Possible outcomes in solving an LP problem

- has a finite optimal solution, which may not be unique.
- *unbounded*: the objective function of maximization (minimization) LP problem is unbounded from above (below) on its feasible region.
- *infeasible*: there are no points satisfying all the constraints, i.e. the constraints are contradictory.

Graphing 2-Dimensional LPs

Example 1:

Maximize $x + y$

Subject to: $x + 2y \geq 2$

$x \leq 3$

$y \leq 4$

$x \geq 0 \quad y \geq 0$

Graphing 2-D LP – Unique solution

Example 1:

Maximize $x + y$

Subject to: $x + 2y \geq 2$

$x \leq 3$

$y \leq 4$

$x \geq 0 \quad y \geq 0$

Graphing LP multiple solutions

Example 2:

Minimize ** $x - y$

Subject to: $1/3\,x + y \leq 4$

$-2x + 2y \leq 4$

$x \leq 3$

$x \geq 0 \quad y \geq 0$

Graphing 2-D LP

Example 3:

Minimize $x + 1/3\ y$

Subject to: $x + y \geq 20$

$-2x + 5y \leq 150$

$x \geq 5$

$x \geq 0 \quad y \geq 0$

Graphing 2-D LP, Unbounded

Example 4:

Maximize $x + 1/3\ y$

Subject to: $x + y \geq 20$

$-2x + 5y \leq 150$

$x \geq 5$

$x \geq 0 \quad y \geq 0$

Optimal Solutions unbounded

Optimal lies on a corner

Extend to Higher Dimensions

To solve an LP

- **The constraints of an LP give rise to a polytope.**

- **If we can determine all the corner points of the polytope, then we can calculate the objective value at these points and take the best one as our optimal solution.**

- **The *Simplex Method* intelligently moves from corner to corner until it can prove that it has found the optimal solution.**

Linear Programs in higher dimensions

<u>maximize</u> $z = -4x_1 + x_2 - x_3$

subject to

c1: $-7x_1 + 5x_2 + x_3 <= 8$

c2: $-2x_1 + 4x_2 + 2x_3 <= 10$

$x_1, \quad x_2, \quad x_3 \quad \geq \quad 0$

Linear Programming

- *Linear programming* (LP) problem is to optimize a linear function of several variables subject to linear constraints:

maximize (or minimize) $c_1 x_1 + \ldots + c_n x_n$

subject to $a_{i1}x_1 + \ldots + a_{in} x_n \leq$ (or $\geq$ or $=$) b_i ,

$i = 1,\ldots,m$, $x_1 \geq 0, \ldots , x_n \geq 0$

The function $z = c_1 x_1 + \ldots + c_n x_n$ is called the

objective function;

constraints $x_1 \geq 0, \ldots , x_n \geq 0$ are called

non-negativity constraints

LP Geometry

- n-dimensional polytope
- convex: mid-point of any two feasible solutions is also feasible.

LP Geometry:
Plane in the Polytope

- Feasible region: convex polytope
  - given by inequalities)
- Objective function: A plane intersecting this polytope

LP Geometry

- Extreme point theorem: If there exists an optimal solution to an LP Problem, then there exists one extreme point where the optimum is achieved.

LP Algorithms

- Simplex by Dantzig 1947
- Practical solution method that moves from one extreme point to a neighboring extreme point.
- Exponential in worse case complexity
- Number of vertices = C(n,m)=n!/(m!(n-m)!)
  - n = number of variables
  - m = number of constraints
- Very fast in practice, steps usually O(m), n effect cost per iteration.
- TWO PHASE simplex:
  1. Phase 1. Finding an initial basic feasible solution may pose a problem
  2. Phase 2. Move from vertex to vertex.

❖ Theoretical possibility of cycling

Worst case simplex takes exponential time to complete.

- Imagine N^2 path in 3-Dimension.

Standard form of LP problem

- **Must be a maximization problem**
- **All constraints (except the non-negativity constraints) must be in the form of linear equations**
- **All the variables must be required to be nonnegative**

Thus, the general linear programming problem in standard form with ***m*** constraints and ***n*** unknowns ($n \geq m$) is

maximize $\boldsymbol{c_1 x_1 + \ldots + c_n x_n}$
subject to $\boldsymbol{a_{i1} x_1 + \ldots + a_{in} x_n = b_i}$, , $i = 1,\ldots,m$,
$\boldsymbol{x_1 \geq 0, \ldots, x_n \geq 0}$

Every LP problem can be represented in such form

In Matrix terms

$$\text{Max } c^T x$$

$$\text{subject to } Ax \leq b$$

$$A_{n\text{x}d}, c_{\text{dx1}}, x_{\text{dx1}}$$

LP Example:
Diet problem

The Diet Problem

- Dietician preparing a diet consisting of two foods: Egg and Dal

	Cost	Protein	Fat	**Carbohydrate**	Buy
Egg	0.60 Rs	20g	12g	30g	x1
Dal	0.40 Rs	30g	6g	15g	x2
daily required		60g	24g	30g	
	Minimize				

Looking for *minimum cost diet*

Diet LP

	Cost	Protein	Fat	**Carbohydrate**	Buy
Egg	0.60 Rs	20g	12g	30g	x1
Dal	0.40 Rs	30g	6g	15g	x2
daily required		60g	24g	30g	
	Minimize				

$$\min C = 0.60x_1 + 0.40x_2$$

$$s.t.\ \ 20x_1 + 30x_2 \geq 60$$

$$12x_1 + \ \ 6x_2 \geq 24$$

$$30x_1 + 15x_2 \geq 30$$

$$where$$

$$x_1, x_2 \geq 0$$

Minimize cost of egg + dal

60gm of protein

24gm of fat

30gm of carb

Dual of Diet LP

$$\min C = 0.60x_1 + 0.40x_2$$

$$s.t.\ \ 20x_1 + 30x_2 \geq 60$$

$$12x_1 + \ \ 6x_2 \geq 24$$

$$30x_1 + 15x_2 \geq 30$$

$$where$$

$$x_1, x_2 \geq 0$$

max z = 60 u1 + 24 u2 + 30 u3
s.t.
20 u1 + 12 u2 + 30 u3 <= 0.6
30 u1 + 6 u2 + 15 u3 <= 0.4

u1,u2,u3 >= 0

	Cost	Protein	Fat	**Carbohydrate**	Buy
Egg	0.60 Rs	20g	12g	30g	x1
Dal	0.40 Rs	30g	6g	15g	x2
daily required		60g	24g	30g	
	Minimize				

Another Diet and its dual example

	Chocolate	Sugar	Cream Cheese	Cost
Brownie	3	2	2	50
Cheesecake	0	4	5	80
Requirements	6	10	8	

$$\begin{array}{ll} \min_{x_1,x_2} & 50x_1 + 80x_2 \\ & 3x_1 \geq 6, \\ \text{subject to} & 2x_1 + 4x_2 \geq 10, \\ & 2x_1 + 5x_2 \geq 8, \\ & x_1, x_2 \geq 0, \end{array}$$

$$\begin{array}{ll} \max_{u_1,u_2,u_3} & 6u_1 + 10u_2 + 8u_3 \\ & 3u_1 + 2u_2 + 2u_3 \leq 50, \\ \text{subject to} & 4u_2 + 5u_3 \leq 80, \\ & u_1, u_2, u_3 \geq 0. \end{array}$$

Buyer's perspective: "How can I buy cheapest brownie and cheesecake with so much chocolate, sugar, cream cheese?"

Wholesaler's perspective: "How can I set the prices per ounce of chocolate, sugar, and cream cheese so that the baker will buy from me, and so that I will maximize my revenue?"

Exercise: Linear Programing LP

Q. Write Linear programming equations to describe the following optimization problem:

A furniture manufacturer makes two types of furniture:

chairs and sofas.

The production of the sofas and chairs requires three operations: carpentry, finishing, and upholstery.

Manufacturing a chair requires 3 hours of carpentry, 9 hours of finishing, and 2 hours of upholstery.

Manufacturing a sofa requires 2 hours of carpentry, 4 hours of finishing, and 10 hours of upholstery.

The factory has allocated at most 66 labor hours for carpentry, 180 labor hours for finishing, and 200 labor hours for upholstery.

The profit per chair is Rs 900 and the profit per sofa is Rs 750.

What are the variables?

What are we optimizing?

What are the constraints?

LP Farm Example

LP Farm Example

- Farmer produces (Apples,Oranges)=(x,y)
- Each crop needs { land, fertilizer, time }.

LP example to help a farmer

We have following linear constraints:

- 6 acres of land: $3x + y \leq 6$
- 6 tons of fertilizer: $2x + 3y \leq 6$
- 8 hour work day: $x + 5y \leq 8$
- Apples sell for twice as much as oranges
- We want to maximize profit: $z = 2x + y$
- Production is positive: $x \geq 0$, $y \geq 0$

Maximize profit

Apples sell for twice as much as oranges

We want to maximize profit (z): $2x + y = z$

Traditional Method

x = 1.71
y = .86
z = 4.29

- Graph the inequalities
- Look at the line we're trying to maximize.

Convert to linear program

- $-z + 2x + y = 0$:Objective Equation
- $s_1 + x + 5y = 8$: C1
- $s_2 + 2x + 3y = 6$: C2
- $s_3 + 3x + y = 6$: C3
- Initial feasible solution
- {s1,s2,s3} = slack variables
- z is the objective function to optimize
- {C1,C2,C3} are the linear constraints.

More definitions

- Non-basic: {x, y}
- Basic variables: $\{s_1, s_2, s_3, z\}$
- First solution: non-basic variables (x,y)=0 means z is also zero in $-z + 2x + y = 0$
- Start with zero profit z, and improve on z.

Next step...

Select a non-basic variable with largest coeff, ie. x, increasing this will give most profit (x gives twice the profit of y):

$-z + 2x + 1y = 0$

Select the basic variable with the smallest positive ratio to exit (s3 go from 6 to 0):

$1x + 5y + s1 = 8$ ratio=8

$2x + 3y + s2 = 6$ ratio=3

$3x + y + s3 = 6$ ratio=2

Algorithm

Step 1. Add slack variables, convert <= to =.

Step 2. Write the tableau (combined matrix)

Step 3.Find pivot element

Step 3a. Pivot column is the most negative coefficient of the objective row. If there is column with negative entry, you have the solution, exit.

Step 3b. Compute ratios of last column divided by pivot column. Pick the row with the minimum positive ratio as the pivot row.

Step 4. Make the pivot element 1 (by dividing the row by the pivot element).

Step 5. Make remaining elements in the pivot column ‘0’, using only row operations.

Reading the tableau: Unit columns are basic variables, whose values are given by the last column, in row with ‘1’. Other variables are ‘0’ (non-basic).

Goto step 3.

Simplex using Maple package

Put the equations into a tableau (matrices):

```
        x   y   s1  s2  s3     // x,y = 0 (non basic)
       [1   5   1   0   0]     // s1 = 8
A := [2   3   0   1   0]     // s2 = 6
       [3   1   0   0   1]    // s3 = 6

      [8]
b := [6]
      [6]

c := [-2 -1 0 0 0]       // optimization function, note signs
    change
z := 0                   // optimal value
```

Use row 3, column 1 as pivot

- > ratios(1); // enter x into basic, find who will leave.
- row 1: Upper bound = 8.0000
- row 2: Upper bound = 3.0000
- row 3: Upper bound = 2.0000 // smallest + ratio, so s3 will leave

- > pivot(3,1); // enter x into basic, leave s3.

x	y	s1	s2	s3	b	
0.00	4.66	1.00	0.00	-.33	6.00	
0.00	2.33	0.00	1.00	-.66	2.00	
1.00	.33	0.00	0.00	.33	2.00	
0.00	-.33	0.00	0.00	.66	4.00	= z

- The optimal z = 4, at (x,y)=(2,0), can be further improved
- Because of negative value in last row: -0.33, y can enter.

Use row 2, column 2 as pivot

- > ratios(2); // enter y, find who will leave
- row 1: Upper bound = 1.2857
- row 2: Upper bound = .8571 // smallest +ve, so s2 will leave
- row 3: Upper bound = 6.0000

- > pivot(2,2); // enter y into basic, leave s2.

x	y	s1	s2	s3	b
0.00	0.00	1.00	-2.00	1.00	2.00
0.00	1.00	0.00	.42	-.28	.85
1.00	0.00	0.00	-.14	.42	1.71
0.00	0.00	0.00	.14	.57	4.28 = z

- No more negative values in last row
- So z = 4.28 is optimal, at (x,y)=(1,0.85)

Homework

- Download glpk (gnu linear programming package) and solve the above problem.
- Use Microsoft Excel to solve the problem.

Simplex Algorithm Example

LP Example, convert to std form

maximize $\mathbf{3x + 5y}$	maximize $\mathbf{3x + 5y + 0u + 0v}$
subject to	subject to
$x + y \le 4$ $x + 3y \le 6$	$x + y + u = 4$ $x + 3y + v = 6$
$x \ge 0, \quad y \ge 0$	$x \ge 0, \; y \ge 0, \; u \ge 0, \; v \ge 0$

Variables ***u*** and ***v***, transforming inequality constraints into equality constrains, are called ***slack variables***

Basic feasible solutions

A *basic solution* to a system of m linear equations in n unknowns ($n \geq m$) is obtained by setting $n - m$ variables to 0 and solving the resulting system to get the values of the other m variables.

The variables set to 0 are called *nonbasic*;
the variables obtained by solving the system are called *basic*.

A basic solution is called *feasible* if all its (basic) variables are nonnegative.

Example

$$\begin{aligned} x + y + u &= 4 \\ x + 3y + v &= 6 \end{aligned}$$

(0, 0, 4, 6) is basic feasible solution
(x, y are non-basic; u, v are basic)

Simplex Tableau

maximize

$$z = 3x + 5y + 0u + 0v$$

subject to

$$x + y + u = 4$$
$$x + 3y + v = 6$$

$$x \geq 0,\ y \geq 0,\ u \geq 0,\ v \geq 0$$

		x	y	u	v		
basic variables	u	1	1	1	0	4	basic feasible solution (0, 0, 4, 6)
	v	1	3	0	1	6	
objective row		−3	−5	0	0	0	value of z at (0, 0, 4, 6)

Outline of the Simplex Method

Step 0 [Initialization] Present a given LP problem in standard form and set up initial tableau.

Step 1 [Optimality test] If all entries in the objective row are nonnegative, then stop: the tableau represents an optimal solution.

Step 2 [Find entering variable] Select (the most) negative entry in the objective row.

Mark its column to indicate the entering variable and the pivot column.

Outline of the Simplex Method

Step 3 [Find departing variable]

•For each positive entry in the pivot column, calculate the θ-ratio by dividing that row's entry in the rightmost column by its entry in the pivot column. (If there are no positive entries in the pivot column then stop: the problem is unbounded.)

•Find the row with the smallest positive θ-ratio, mark this row to indicate the departing variable and the pivot row.

Step 4 [Form the next tableau]

• Divide all the entries in the pivot row by its entry in the pivot column.

• Subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the pivot column of the row in question.

• Replace the label of the pivot row by the variable's name of the pivot column and go back to Step 1.

maximize

$$3x + 5y + 0u + 0v = z$$

subject to

$$x + y + u = 4$$
$$x + 3y + v = 6$$

$$x \geq 0,\ y \geq 0,\ u \geq 0,\ v \geq 0$$

Example of Simplex Method

	x	y	u	v		
u	1	1	1	0	4	
← v	1	3	0	1	6	
	−3	−5	0	0	0	z

↑

BFS: (0, 0, 4, 6)

$z = 0$

Pivot column 2

Ratios: 4/1=4, 6/3=2

Pivot row 2

	x	y	u	v		
← u	$\frac{2}{3}$	0	1	$-\frac{1}{3}$	2	
y	$\frac{1}{3}$	1	0	$\frac{1}{3}$	2	
	$-\frac{4}{3}$	0	0	$\frac{5}{3}$	10	z

↑

BFS: (0, 2, 2, 0)

$z = 10$

Pivot column 1

Ratios: 2/(2/3)=3, 2/(1/3)=6

Pivot row: 1

	x	y	u	v		
x	1	0	3/2	-1/2	3	
y	0	1	−1/2	1/2	1	
	0	0	2	1	14	z

BFS: (3, 1, 0, 0)

$z = 14$

Chapter 29: Linear Programming

Motivating Example:

- A company makes two products, X and Y.
- Its labor force can work at most 900 hours per week.
- Its outlay for running costs must be at most $10,000 per week.
- It takes 5 hours to produce one item of X; the cost is $80, the profit is $30.
- It takes 2 hours to produce one item of Y; the cost is $100, the profit is $20.

How should this company maximize its profit?

Express as a **linear program**:

$$\begin{aligned} \text{maximize} \quad & 30x + 20y \\ \text{subject to} \quad & 5x + 2y \le 900 \\ & 80x + 100y \le 10000 \\ & x \ge 0,\ y \ge 0 \end{aligned}$$

(linear) objective function

(linear) constraints

Here, x (y) denotes the number of items of X (Y) produced per week.

General Linear Programming

- A linear function f is defined by
  $f(x_1, x_2, \ldots, x_n) = a_1x_1 + a_2x_2 + \ldots + a_nx_n$,
  where the a_i's are real numbers.
- Want to either **maximize** or **minimize** a linear function subject to some linear constraints.
- Each constraint is of the form
  $$f(x_1, x_2, \ldots, x_n) \ ? \ b,$$
  where ? is $=$, $\leq$, or $\geq$ and b is a real number.
  - Called a **linear equality** if ? is $=$, and a **linear inequality** otherwise.
  - Strict inequalities are not allowed!

General vs. Integer LP

- General linear programming problems are solvable in polynomial time.
  - The algorithm we will cover, the simplex algorithm, can actually require exponential time for some inputs, but is usually fast in practice.
- If the variables (the x_i's) are required to be integers, then we have an **integer linear programming problem**.
- Integer linear programming is NP-hard.

Outline

- Standard and slack forms.
- Formulating problems as linear programs.
- The simplex algorithm (motivated by an extended example).
- Termination.
- Optimality and duality.
- Finding an initial basic feasible solution.

Standard Form

In **standard form**, a linear program is expressed precisely as follows:

$$\boxed{\begin{aligned} &\text{maximize} \quad \sum_{j=1}^{n} c_j x_j \\ &\text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_j \le b_i \quad \text{for } i = 1, 2, \ldots, m \\ &\qquad\qquad\qquad x_j \ge 0 \quad \text{for } j = 1, 2, \ldots, n. \end{aligned}}$$

The c_j, a_{ij}, and b_i terms are all real numbers. In **matrix form**:

$$\boxed{\begin{aligned} &\text{maximize} \quad c^T x \\ &\text{subject to} \quad Ax \le b,\ x \ge 0 \end{aligned}}$$

where $A = (a_{ij})$, $b = (b_i)$, and $c = (c_j)$.

Some Terminology

- An assignment $\bar{x}$ of values to the variables x is a **feasible solution** if all constraints are satisfied, and an **infeasible solution** otherwise.
- We say solution $\bar{x}$ has the **objective value** $c^T\bar{x}$.
- If the objective value of $\bar{x}$ is the maximum over all feasible solutions, then $\bar{x}$ is an **optimal solution** and $c^T\bar{x}$ is the **optimal objective value**.
- A linear program is **feasible** if it has a feasible solution, and **infeasible** otherwise. If it has a feasible solution, but no optimal solution with a finite objective value, then it's **unbounded**.

Example

maximize $x_1 + x_2$

subject to $4x_1 - x_2 \le 8$

$2x_1 + x_2 \le 10$

$-5x_1 + 2x_2 \le 2$

$x_1 \ge 0$

$x_2 \ge 0$

Note: This is a convex region (more on this later).

Converting to Standard Form

- **Problems:**
  - Minimization problem
    - Negate objective function and find maximum instead.
  - Some variable x_j is not subject to $x_j \geq 0$.
    - Replace x_j by $x'_j - x''_j$ and include $x'_j \geq 0$ and $x''_j \geq 0$.
  - Some constraints are equality constraints.
    - Replace $f(x_1, x_2, \ldots, x_n) = b$ by $f(x_1, x_2, \ldots, x_n) \leq b$ <u>and</u> $f(x_1, x_2, \ldots, x_n) \geq b$.
  - Some constraints are "≥" constraints.
    - Replace $\Sigma_j\, a_{ij}x_j \geq b$ by $\Sigma_j\, -a_{ij}x_j \leq -b$.

Slack Form

(Assumed by the Simplex Algorithm)

- In slack form, all constraints, other than the "$x_i \geq 0$" constraints, are equalities.
- Converting from standard from to slack form:
  - Replace
    - $\sum_j a_{ij}x_j \leq b_i$
  - by
    - $s = b_i - \sum_j a_{ij}x_j$
    - $s \geq 0$.
  - s is called a **slack variable**: it measures the "slack" between the LHS and RHS of the original inequality.

Example

In slack form, we often drop "maximize" and "subject to" and use z to denote the objective function.

$$\begin{aligned} \text{maximize} \quad & x_1 + x_2 \\ \text{subject to} \quad & 4x_1 - x_2 \le 8 \\ & 2x_1 + x_2 \le 10 \\ & -5x_1 + 2x_2 \le 10 \\ & x_1, x_2 \ge 0 \end{aligned}$$

standard form

$$\begin{aligned} z &= x_1 + x_2 \\ x_3 &= 8 - 4x_1 + x_2 \\ x_4 &= 10 - 2x_1 - x_2 \\ x_5 &= 10 + 5x_1 - 2x_2 \\ & x_1, x_2, x_3, x_4, x_5 \ge 0 \end{aligned}$$

slack form

These slack variables are called **basic variables**; the other x_i's are **nonbasic**.

General Specification

Slack Form:

$$\boxed{\begin{aligned} z &= v + \sum_{j \in N} c_j x_j \\ x_i &= b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i \in B \end{aligned}}$$

As before, this can be specified in matrix/vector notation.

Formulating Problems as LPs

- **Example 1:** Single-pair shortest-path problem.
- As a linear program:

$$
\begin{array}{ll}
\text{maximize} & d[t] \\
\text{subject to} & d[v] \le d[u] + w(u,v) \quad \text{for each edge } (u,v) \in E \\
& d[s] = 0
\end{array}
$$

1st edition of CLRS says minimize, which is wrong!

- **Exercise:** Generalize to formulate the single-source shortest-path problem.

Another Example

- **Example 2:** Maximum flow.
- As a linear program:

maximize	$\sum_{v \in V} f(s,v)$	
subject to	$f(u,v) \leq c(u,v)$	for each $u,v \in V$
	$f(u,v) = -f(v,u)$	for each $u,v \in V$
	$\sum_{v \in V} f(u,v) = 0$	for each $u \in V - \{s,t\}$

And Another

- **Example 3:** Minimum-cost flow.
  - Each edge (u,v) has a real-valued cost a(u,v). Also, a target flow d is given.
- As a linear program:

$$
\begin{array}{lll}
\text{minimize} & \sum_{(u,v) \in V} a(u,v) \cdot f(u,v) & \\
\text{subject to} & f(u,v) \le c(u,v) & \text{for each } u,v \in V \\
 & f(u,v) = -f(v,u) & \text{for each } u,v \in V \\
 & \sum_{v \in V} f(u,v) = 0 & \text{for each } u \in V - \{s,t\} \\
 & \sum_{v \in V} f(s,v) = d &
\end{array}
$$

Is this a <u>linear</u> expression? (It better be!)

And Yet Another

- **Example 4:** Multicommodity flow. Twists:
  - Have k different commodities $K_1, \ldots, K_k$.
  - K_i is specified by (s_i, t_i, d_i) = (source, sink, desired flow).
  - **Aggregate flow** from u to v is $f(u,v) = \sum_{i=1..k} f_i(u,v)$, where f_i is the actual flow for K_i.
  - **Question:** Does such a flow exist? (Note: nothing to minimize.)
- As a linear program:

$$
\begin{array}{ll}
\text{minimize} & 0 \\
\text{subject to} & \sum_{i=1..k} f_i(u,v) \le c(u,v) \qquad \text{for each } u,v \in V \\
& f_i(u,v) = -f_i(v,u) \qquad \text{for each } i \text{ and } u,v \in V \\
& \sum_{v \in V} f_i(u,v) = 0 \qquad \text{for each } i \text{ and } u \in V - \{s_i,t_i\} \\
& \sum_{v \in V} f_i(s_i,v) = d_i
\end{array}
$$

Note: This is the only known polynomial-time way to solve this problem.

Simplex Algorithm

(First, the Name)

Recall this example:

$$
\begin{aligned}
\text{maximize} \quad & x_1 + x_2 \\
\text{subject to} \quad & 4x_1 - x_2 \le 8 \\
& 2x_1 + x_2 \le 10 \\
& -5x_1 + 2x_2 \le 2 \\
& x_1 \ge 0 \\
& x_2 \ge 0
\end{aligned}
$$

Simplex Algorithm

(First, the Name)

Each constraint defines a **half-space** in 2-d space. The intersection of these half-spaces is called a **simplex**. An optimal solution (if one exists) will correspond to a **vertex** of this simplex.

All of this is true for higher dimensions (i.e., more variables), but it's hard to imagine graphically.

The simplex algorithm starts at a vertex of the simplex. It then iterates. At each iteration, it moves along an edge to a neighboring vertex whose objective value is at least as big (usually bigger).

It stops when it reaches a **local maximum**. This will also be a **global optimum**. (Why?)

Motivating Example

$$
\begin{array}{ll}
\text{maximize} & 3x_1 + x_2 + 2x_3 \\
\text{subject to} & x_1 + x_2 + 3x_3 \leq 30 \\
& 2x_1 + 2x_2 + 5x_3 \leq 24 \\
& 4x_1 + x_2 + 2x_3 \leq 36 \\
& x_1, x_2, x_3 \geq 0
\end{array}
$$

First, we must convert to slack form…

In Slack Form

$$
\begin{aligned}
z &= 3x_1 + x_2 + 2x_3 \\
x_4 &= 30 - x_1 - x_2 - 3x_3 \\
x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\
x_6 &= 36 - 4x_1 - x_2 - 2x_3
\end{aligned}
$$

Note: All the x_i's are subject to a "$x_i \geq 0$" constraint, so we will not bother to show these.

We say that an equality constraint is **tight** for a particular setting of its nonbasic variables if the constraint's basic variable is 0.

Note: The slack variables indicate how far from tight each constraint is, and thus indicate how much the nonbasic variables can be increased.

Feasible Solutions

$$
\begin{aligned}
z &= 3x_1 + x_2 + 2x_3 \\
x_4 &= 30 - x_1 - x_2 - 3x_3 \\
x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\
x_6 &= 36 - 4x_1 - x_2 - 2x_3
\end{aligned}
$$

Each assignment to x_1, x_2, x_3 defines values for x_4, x_5, x_6.

Such an assignment is a **feasible solution** if each of $x_1, \ldots, x_6$ is nonnegative.

Note: There can be an infinite number of feasible solutions.

Basic (Feasible) Solutions

$$
\begin{array}{rcl}
z & = & 3x_1 + x_2 + 2x_3 \\
x_4 & = & 30 - x_1 - x_2 - 3x_3 \\
x_5 & = & 24 - 2x_1 - 2x_2 - 5x_3 \\
x_6 & = & 36 - 4x_1 - x_2 - 2x_3
\end{array}
$$

The **basic solution** is defined by assigning all nonbasic variables the value 0.

For this example, the basic solution is $(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_6) = (0, 0, 0, 30, 24, 36)$ and it has an objective value of $z = (3 \cdot 0) + (1 \cdot 0) + (2 \cdot 0) = 0$.

If a basic solution is also feasible, it is called a **basic feasible solution**.

Outline of Algorithm

- Starting with a basic solution, the algorithm iterates.
- At each iteration, the equations and objective function are rewritten so as to put a different set of variables on the RHS.
- The new problem has a different basic solution.
- **Goal:** Each iteration produces a linear program whose basic solution (usually) has a greater objective value.
  - Select some nonbasic variable x_e (e = "entering") whose coefficient in the objective function is positive.
  - Increase the value of x_e without violating any constraint.
  - Makes x_e basic and some x_ℓ (ℓ = "leaving") nonbasic.

Example (Continued)

$$
\begin{aligned}
z &= 3x_1 + x_2 + 2x_3 \\
x_4 &= 30 - x_1 - x_2 - 3x_3 \\
x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\
x_6 &= 36 - 4x_1 - x_2 - 2x_3
\end{aligned}
$$

Suppose we choose to increase x_1.

How much can we increase it? Increasing x_1 decreases x_4, x_5, and x_6, which must remain nonnegative.

The third constraint is the tightest, so we rewrite this equation to put x_1 on the LHS and then substitute out x_1 on all RHS's.

Example (Continued)

$$\boxed{\begin{aligned} z &= 3x_1 + x_2 + 2x_3 \\ x_4 &= 30 - x_1 - x_2 - 3x_3 \\ x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\ x_6 &= 36 - 4x_1 - x_2 - 2x_3 \end{aligned}}$$

Rewriting the third constraint, we get $x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}$.

Rewriting the first constraint (the others are similar):

$$x_4 = 30 - x_1 - x_2 - 3x_3$$

$$= 30 - \left(9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}\right) - x_2 - 3x_3 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4}.$$

Pivoting

The new problem looks like this:

$$
\begin{aligned}
z &= 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \\
x_1 &= 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \\
x_4 &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \\
x_5 &= 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2}
\end{aligned}
$$

The basic solution is now (9, 0, 0, 21, 6, 0) with objective value $z = 27$.

The old objective value was 0, so we have an increase!

This operation is called **pivoting** (just like Gaussian elimination). $x_e = x_1$ is called the **entering variable**; $x_\ell = x_6$ is called the **leaving variable**.

Next Step

We can next increase either x_2 or x_3. (Why not x_6?) We choose x_3.

$$
\begin{aligned}
z &= 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \\
x_1 &= 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \\
x_4 &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \\
x_5 &= 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2}
\end{aligned}
$$

The third constraint is the tightest (check this). So, we "exchange" x_3 and x_5.

After Pivoting

After "exchanging" x_3 and x_5:

$$
\begin{aligned}
z &= \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \\
x_1 &= \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \\
x_3 &= \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \\
x_4 &= \frac{69}{4} - \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}
\end{aligned}
$$

Can now increase only x_2. (Why?) The second constraint is the tightest, so we "exchange" x_2 and x_3.

Finally!

After "exchanging" x_2 and x_3:

$$
\begin{aligned}
z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\
x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\
x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\
x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2}
\end{aligned}
$$

Nothing can be increased now, so the algorithm terminates. Basic solution yields an objective value of **28**, which is optimal.

Motivating Example Again

The simplex algorithm starts at a vertex of the simplex. It then iterates. At each iteration, it moves along an edge to a neighboring vertex whose objective value is at least as big (usually bigger).

Choosing pivots in the order $x_1 \leftrightarrow x_3$, $x_2 \leftrightarrow x_4$, $x_3 \leftrightarrow x_5$ will cause simplex to visit vertices in this fashion:

Pivot Procedure

set of nonbasic
and basic variables

indices of leaving
and entering variables

$$N, B, A, b, c, v, \ell, e \;\Rightarrow\; \boxed{\text{Pivot}} \;\Rightarrow\; N', B', A', b', c', v'$$

defines the program

from slack
form notation

$$\boxed{\begin{aligned} z &= v + \sum_{j \in N} c_j x_j \\ x_i &= b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i \in B \end{aligned}}$$

Example

$$
\begin{aligned}
z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\
x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\
x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\
x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2}
\end{aligned}
$$

Note: The "A" coefficients get negated.

$B = \{1, 2, 4\}$, $N = \{3, 5, 6\}$,

$$A = \begin{pmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{pmatrix},$$

$$b = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix},$$

$c = (-1/6 \;\; -1/6 \;\; -2/3)^T$, and

$v = 28$.

Code

```
Pivot(N, B, A, b, c, v, ℓ, e)
    /* compute the coefficients of the      */
    /* equation for new basic variable x_e */
    b′_e := b_ℓ/a_ℓe;
    for each j ∈ N – {e} do
        a′_ej := a_ℓj /a_ℓe
    od;
    a′_eℓ := 1/a_ℓe;
    /* compute coefficients of the */
    /* remaining constraints       */
    for each i ∈ B – {ℓ} do
        b′_i := b_i – a_ie b′_e;
        for each j ∈ N – {e} do
            a′_ij := a_ij – a_ie a′_ej
        od;
        a′_iℓ := –a_ie a′_eℓ
    od;
    /* compute the objective function */
    v′ := v + c_e b′_e;
    for each j ∈ N – {e} do
        c′_j := c_j – c_e a′_ej
    od;
    c′_ℓ := –c_e a′_eℓ;
    /* compute new sets of basic */
    /* and nonbasic variables    */
    N′ := N – {e} ∪ {ℓ};
    B′ := B – {ℓ} ∪ {e};
    return (N′, B′, A′, b′, c′, v′)
```

Simplex Procedure

```
Simplex(A, b, c)
    (N, B, A, b, c, v) := Init-Simplex(A, b, c);
    while some index j ∈ N has c_j > 0 do
        choose an index e ∈ N for which c_e > 0;
        for each index i ∈ B do
            if a_ie > 0 then Δ_i := b_i/a_ie else Δ_i := ∞ fi
        od;
        choose an index ℓ ∈ B that minimize Δ_ℓ;
        if Δ_ℓ = ∞ then return "unbounded"
                    else (N, B, A, b, c, v) := Pivot(N, B, A, b, c, v, ℓ, e)
        fi
    od;
    for i := 1 to n do
        if i ∈ B then x̄_i := b_i else x̄_i := 0 fi;
    od;
    return (x̄_1, x̄_2, …, x̄_n)
```

As explained later, this returns an initial basic feasible solution if one exists. If the program is infeasible, it will be detected and reported by this routine.

Termination

- We will skip the proof that Simplex returns a feasible solution (if it terminates).
  - The proof is rather tedious, and the extended example we did pretty much explains why it works.
- We do, however, carefully consider the question of when or whether it terminates.
- It can be shown that pivoting cannot decrease the objective value.
- But, it can leave it unchanged … this is called **degeneracy**.

Degeneracy

The objective value is changed only by the assignment
" $v' := v + c_e b'_e$ " in Pivot.

Simplex calls Pivot only when $c_e > 0$. Thus, if the objective value remains unchanged, then it must be that b'_e is 0.

The value of b'_e is determined by the assignment "$b'_e := b_\ell / a_{\ell e}$" in Pivot.

(Pivot is always called with $a_{\ell e} \neq 0$.)

Question: Can b_ℓ be 0 when Pivot is called? **Yes!**

Example

$$\begin{aligned} z &= x_1 + x_2 + x_3 \\ x_4 &= 8 - x_1 - x_2 \\ x_5 &= x_2 - x_3 \end{aligned}$$

objective value = 0

$x_1 \leftrightarrow x_4$

$$\begin{aligned} z &= 8 + x_3 - x_4 \\ x_1 &= 8 - x_2 - x_4 \\ x_5 &= x_2 - x_3 \end{aligned}$$

objective value = 8

$x_3 \leftrightarrow x_5$

same

$$\begin{aligned} z &= 8 + x_2 - x_4 - x_5 \\ x_1 &= 8 - x_2 - x_4 \\ x_3 &= x_2 - x_5 \end{aligned}$$

objective value = 8

Cycles

In the previous example, one more pivot with $x_2 \leftrightarrow x_1$ would increase the objective function.

However, it is possible for Simplex to **cycle** back to a previous slack form and thus never terminate.

Lemma 29.5: If Simplex fails to terminate in at most $\binom{n+m}{m}$ iterations, then it cycles.

Proof:

There are n + m variables and |B| = m, and therefore there are $\binom{n+m}{m}$ ways to choose B.

Preventing Cycles

Cycles can be prevented by using some tie-breaking scheme when choosing entering and leaving variables:

Lemma 29.6: If, when selecting entering and leaving variables, Simplex always breaks ties by choosing the variable with the smallest index, then Simplex must terminate.

Optimality

- We must also argue that the feasible solution returned by Simplex is also an **optimal** solution.
- This is argued in Section 29.4 by introducing the concept of **duality**.
- Here, we give a cursory overview of the argument.

Duality

$$\text{maximize} \quad \sum_{j=1}^{n} c_j x_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_j \le b_i \quad \text{for } i = 1, 2, \ldots, m$$

$$x_j \ge 0 \quad \text{for } j = 1, 2, \ldots, n.$$

Primal

dual problems

$$\text{minimize} \quad \sum_{i=1}^{m} b_i y_i$$

$$\text{subject to} \quad \sum_{i=1}^{m} a_{ij} y_i \ge c_j \quad \text{for } j = 1, 2, \ldots, n$$

$$y_i \ge 0 \quad \text{for } i = 1, 2, \ldots, m.$$

Dual

Example

$$
\begin{array}{ll}
\text{maximize} & 3x_1 + x_2 + 2x_3 \\
\text{subject to} & x_1 + x_2 + 3x_3 \le 30 \\
& 2x_1 + 2x_2 + 5x_3 \le 24 \\
& 4x_1 + x_2 + 2x_3 \le 36 \\
& x_1, x_2, x_3 \ge 0
\end{array}
$$

$$\Updownarrow$$

$$
\begin{array}{ll}
\text{minimize} & 30y_1 + 24y_2 + 36y_3 \\
\text{subject to} & y_1 + 2y_2 + 4y_3 \ge 3 \\
& y_1 + 2y_2 + y_3 \ge 1 \\
& 3y_1 + 5y_2 + 2y_3 \ge 2 \\
& y_1, y_2, y_3 \ge 0
\end{array}
$$

Duality

- In the linear case, in the primal problem, from each sub-optimal point that satisfies all the constraints, there is a direction or subspace of directions to move that increases the objective function. Moving in any such direction is said to remove **slack** between the candidate solution and one or more constraints. An **infeasible** value of the candidate solution is one that exceeds one or more of the constraints.
- In the dual problem, the dual vector multiplies the constants that determine the positions of the constraints in the primal. Varying the dual vector in the dual problem is equivalent to revising the upper bounds in the primal problem. The lowest upper bound is sought. That is, the dual vector is minimized in order to remove slack between the candidate positions of the constraints and the actual optimum. An infeasible value of the dual vector is one that is too low. It sets the candidate positions of one or more of the constraints in a position that excludes the actual optimum. (from Wikipedia)

Duality and Economics

- The primal problem deals with physical quantities. With all inputs available in limited quantities, and assuming the unit prices of all outputs is known, what quantities of outputs to produce so as to maximize total revenue?
- The dual problem deals with economic values. With floor guarantees on all output unit prices, and assuming the available quantity of all inputs is known, what input unit pricing scheme to set so as to minimize total expenditure? (from Wikipedia)

Duality Example: Farmer

- Suppose that a farmer has a piece of farm land, say A square kilometres large, to be planted with either wheat or barley or some combination of the two. The farmer has a limited permissible amount F of fertilizer and P of insecticide which can be used, each of which is required in different amounts per unit area for wheat (F_1, P_1) and barley (F_2, P_2). Let S_1 be the selling price of wheat, and S_2 the price of barley. If we denote the area planted with wheat and barley by x_1 and x_2 respectively, then the optimal number of square kilometres to plant with wheat vs barley can be expressed as a linear programming problem:
- Maximize: $S_1\mathbf{x}_1 + S_2\mathbf{x}_2$ (maximize the revenue—revenue is the "objective function") subject to: $\mathbf{x}_1 + \mathbf{x}_2 \leq A$ (limit on total area) $F_1\mathbf{x}_1 + F_2\mathbf{x}_2 \leq F$ (limit on fertilizer) $P_1\mathbf{x}_1 + P_2\mathbf{x}_2 \leq P$ (limit on insecticide) $\mathbf{x}_1 \geq 0$, $\mathbf{x}_2 \geq 0$ (cannot plant a negative area).

Duality Example

- **Example (Dual Problem)**
- Revisit the above example of the farmer who may grow wheat and barley with the set provision of some A land, F fertilizer and P insecticide. Assume now that unit prices $\mathbf{y}_A$, $\mathbf{y}_F$, $\mathbf{y}_\mathrm{P}$ for each of these means of production (inputs) are set by a planning board. The planning board's job is to minimize the total cost of procuring the set amounts of inputs while providing the farmer with a floor on the unit price of each of his crops (outputs), S_1 for wheat and S_2 for barley. This corresponds to the following linear programming problem:
- Minimize: $A\mathbf{y}_A + F\mathbf{y}_F + P\mathbf{y}_\mathrm{P}$ (minimize the total cost of the means of production as the "objective function") subject to: $\mathbf{y}_A + F_1\mathbf{y}_F + P_1\mathbf{y}_P \geq S_1$ (the farmer must receive no less than S_1 for his wheat) $\mathbf{y}_A + F_2\,\mathbf{y}_F + P_2\mathbf{y}_P \geq S_2$ (the farmer must receive no less than S_2 for his barley) $\mathbf{y}_A \geq 0$, $\mathbf{y}_F \geq 0$, $\mathbf{y}_P \geq 0$ (prices cannot be negative). (From Wikipedia)
- Intuition: The farmer has a cost per unit area for wheat of $\mathbf{y}_A + F_1\mathbf{y}_F + P_1\mathbf{y}_P$ so he must get at least this much for it to pay his costs. The planning board wants to make this minimum cost at least S_1 .

Duality Motivation

- In the farmer example, suppose it is desired to set the prices $\mathbf{y}_A$, $\mathbf{y}_F$, $\mathbf{y}_\mathrm{P}$ fairly. If the prices are set too low, then the farmer can make more money by selling crops than by selling the raw materials (fertilizer, insecticide, land), so he has no motivation to sell the raw materials. If the prices are set too high, then one pays the farmer too much for these things. One desires to set the prices as low as possible so that the farmer will have a motivation to sell.
- This means $\mathbf{y}_A + F_1\mathbf{y}_F + P_1\mathbf{y}_P \geq S_1$ or else the farmer can profit more from raising wheat than from selling the raw materials, and $\mathbf{y}_A + F_2\,\mathbf{y}_F + P_2\mathbf{y}_P \geq S_2$ or else the farmer can gain more from raising barley.
- It is desired to minimize the cost of purchasing all of the farmer's raw materials, including leasing the land for one growing season. This cost is $A\mathbf{y}_A + F\mathbf{y}_F + P\mathbf{y}_\mathrm{P}$, so it is desired to minimize this quantity.

Duality Bound

Because $\mathbf{x}_1 + \mathbf{x}_2 \leq A$, $\mathbf{x}_1\mathbf{y}_A + \mathbf{x}_2\mathbf{y}_A \leq A\mathbf{y}_A$.

Because $F_1\mathbf{x}_1 + F_2\mathbf{x}_2 \leq F$, $F_1\mathbf{x}_1\mathbf{y}_F + F_2\mathbf{x}_2\mathbf{y}_F \leq F\mathbf{y}_F$

Because $P_1\mathbf{x}_1 + P_2\mathbf{x}_2 \leq P$, $P_1\mathbf{x}_1\mathbf{y}_P + P_2\mathbf{x}_2\mathbf{y}_P \leq P\mathbf{y}_P$

(all assuming $\mathbf{y}_A, \mathbf{y}_F, \mathbf{y}_P \geq 0$).

Adding these three inequalities and factoring,

$\mathbf{x}_1(\mathbf{y}_A + F_1\mathbf{y}_F + P_1\mathbf{y}_P) + \mathbf{x}_2(\mathbf{y}_A + F_2\mathbf{y}_F + P_2\mathbf{y}_P) \leq A\mathbf{y}_A + F\mathbf{y}_F + P\mathbf{y}_P$

Assuming $\mathbf{y}_A + F_1\mathbf{y}_F + P_1\mathbf{y}_P \geq S_1$ and $\mathbf{y}_A + F_2\mathbf{y}_F + P_2\mathbf{y}_P \geq S_2$,

$\mathbf{x}_1S_1 + \mathbf{x}_2S_2 \leq A\mathbf{y}_A + F\mathbf{y}_F + P\mathbf{y}_P$. Thus, assuming $\mathbf{y}_A, \mathbf{y}_F, \mathbf{y}_P$ satisfy these constraints, the farmer cannot make money no matter how he divides his land if the prices of wheat and barley are S_1 and S_2, which puts a floor on what he can charge for wheat and barley. Note that $\mathbf{y}_A, \mathbf{y}_F, \mathbf{y}_P$ do not depend on $\mathbf{x}_1$, $\mathbf{x}_2$.

Lemma 29.8

$$\begin{aligned} &\text{maximize} && \sum_{j=1}^{n} c_j x_j \\ &\text{subject to} && \sum_{j=1}^{n} a_{ij} x_j \le b_i && \text{for } i = 1, 2, \ldots, m \\ & && x_j \ge 0 && \text{for } j = 1, 2, \ldots, n. \end{aligned}$$

$$\Updownarrow$$

$$\begin{aligned} &\text{minimize} && \sum_{i=1}^{m} b_i y_i \\ &\text{subject to} && \sum_{i=1}^{m} a_{ij} y_i \ge c_j && \text{for } j = 1, 2, \ldots, n \\ & && y_i \ge 0 && \text{for } i = 1, 2, \ldots, m. \end{aligned}$$

Lemma 29.8: $\sum_{j=1}^{n} c_j \bar{x}_j \le \sum_{i=1}^{m} b_i \bar{y}_i$.

Proof:

$$\begin{aligned} \sum_{j=1}^{n} c_j \bar{x}_j &\le \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j \\ &= \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} \bar{x}_j \right) \bar{y}_i \\ &\le \sum_{i=1}^{m} b_i \bar{y}_i \end{aligned}$$

Corollary 29.9

Corollary 29.9: Let $\bar{x}$ be a feasible solution to the primal linear program (A, b, c), and let $\bar{y}$ be a feasible solution to the corresponding dual linear program. If

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i$$

then $\bar{x}$ and $\bar{y}$ are optimal solutions to the primal and dual linear programs, respectively.

Proof:

By Lemma 29.8, the objective value of a feasible solution to the primal cannot exceed that of the dual. The primal program is a maximization problem and the dual is a minimization problem. Thus, if the objective values are the same, neither can be improved.

Duality, Revisited

In Theorem 29.10 (not covered in detail here), it is shown that, given the $\bar{x}$ values returned by Simplex (i.e., the feasible solution returned for the primal problem), defining $\bar{y}$ values by

$$\bar{y}_i = \begin{cases} -c'_{n+i} & \text{if } (n+i) \in N, \\ 0 & \text{otherwise} \end{cases}$$

gives a feasible solution to the dual problem. Moreover, we have

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i .$$

By Corollary 29.9, this implies that **Simplex returns an optimal solution** (implicitly, for both problems).

Example

$$
\begin{aligned}
z &= 3x_1 + x_2 + 2x_3 \\
x_4 &= 30 - x_1 - x_2 - 3x_3 \\
x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\
x_6 &= 36 - 4x_1 - x_2 - 2x_3
\end{aligned}
$$

$x_1 \leftrightarrow x_6$

$$
\begin{aligned}
z &= 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \\
x_1 &= 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \\
x_4 &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \\
x_5 &= 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2}
\end{aligned}
$$

In basic solution of 2nd slack form, original variables get the values $(\bar{x}_1, \bar{x}_2, \bar{x}_3) = (9, 0, 0)$. Plugging these values in *original* problem gives an objective value of 27.

$$
\begin{aligned}
z &= -30y_1 - 24y_2 - 36y_3 \\
y_4 &= -3 + y_1 + 2y_2 + 4y_3 \\
y_5 &= -1 + y_1 + 2y_2 + y_3 \\
y_6 &= -2 + 3y_1 + 5y_2 + 2y_3
\end{aligned}
$$

$y_3 \leftrightarrow y_4$

$$
\begin{aligned}
z &= -27 - 9y_4 - 21y_1 - 6y_2 \\
y_5 &= -\frac{1}{4} + \frac{1}{4}y_4 + \frac{3}{4}y_1 + \frac{3}{4}y_2 \\
y_6 &= -\frac{1}{2} + \frac{1}{2}y_4 + \frac{5}{2}y_1 + 4y_2 \\
y_3 &= \frac{3}{4} + \frac{1}{4}y_4 - \frac{1}{4}y_1 - \frac{1}{2}y_2
\end{aligned}
$$

In basic solution of 2nd slack form, original variables get the values $(\bar{y}_1, \bar{y}_2, \bar{y}_3) = (0, 0, 3/4)$. Plugging these values in *original* problem gives an objective value of –27.

Example

Note: This solution is *not* feasible for the dual problem!

Why not?

As Simplex iterates, it will produce *identical* solutions to the two problems (27 here).

We can show that the last solution is feasible for both problems.

By Corollary 29.9, this implies that we have optimal solutions to both problems.

$$
\begin{aligned}
z &= -30y_1 - 24y_2 - 36y_3 \\
y_4 &= -3 + y_1 + 2y_2 + 4y_3 \\
y_5 &= -1 + y_1 + 2y_2 + y_3 \\
y_6 &= -2 + 3y_1 + 5y_2 + 2y_3
\end{aligned}
$$

$y_3 \leftrightarrow y_4$

$$
\begin{aligned}
z &= -27 - 9y_4 - 21y_1 - 6y_2 \\
y_5 &= -\frac{1}{4} + \frac{1}{4}y_4 + \frac{3}{4}y_1 + \frac{3}{4}y_2 \\
y_6 &= -\frac{1}{2} + \frac{1}{2}y_4 + \frac{5}{2}y_1 + 4y_2 \\
y_3 &= \frac{3}{4} + \frac{1}{4}y_4 - \frac{1}{4}y_1 - \frac{1}{2}y_2
\end{aligned}
$$

In basic solution of 2nd slack form, original variables get the values $(\overline{y}_1, \overline{y}_2, \overline{y}_3) = (0, 0, 3/4)$. Plugging these values in *original* problem gives an objective value of –27.

The Initial Basic Feasible Solution

- Init-Simplex must return an initial basic feasible solution.

- **Question:** Is one guaranteed to exist?

- **No!**

Example

$$\begin{array}{lrcrcr}
\text{maximize} & 2x_1 & - & x_2 & & \\
\text{subject to} & 2x_1 & - & x_2 & \leq & 2 \\
 & x_1 & - & 5x_2 & \leq & -4 \\
 & x_1, x_2 & \geq & 0 & &
\end{array}$$

After converting to slack form, the basic solution would be $x_1 = 0$ and $x_2 = 0$. This violates the second constraint and thus is not a feasible solution.

This doesn't necessarily imply this program has <u>no</u> feasible solutions.

To determine this, we can formulate an **<u>auxiliary linear program</u>**.

Auxiliary Linear Program

$$L \left\{ \begin{array}{ll} \text{maximize} & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \le b_i \quad \text{for } i = 1, 2, \ldots, m \\ & x_j \ge 0 \quad \text{for } j = 1, 2, \ldots, n. \end{array} \right.$$

$$L_{aux} \left\{ \begin{array}{ll} \text{maximize} & -x_0 \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j - x_0 \le b_i \quad \text{for } i = 1, 2, \ldots, m \\ & x_j \ge 0 \quad \text{for } j = 0, 1, \ldots, n. \end{array} \right.$$

<u>Lemma 29.11:</u> L is feasible iff the optimal objective value of L_{aux} is 0.

Proof of Lemma 29.11

Suppose L has a feasible solution $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$. This plus $\bar{x}_0 = 0$ is a feasible solution for L_{aux}. Moreover, $\bar{x}_0 = 0$ is clearly an optimal objective value.

Conversely, suppose that the optimal objective value of L_{aux} is 0. Then, $\bar{x}_0 = 0$, and the values of the remaining variables satisfy the constraints of L.

Init-Simplex

```
Init-Simplex(A, b, c)
    let ℓ be the index of the minimum b_i;
    if b_ℓ ≥ 0 then
        /* initial basic solution is feasible */
        return({1, 2, …, n}, {n+1, n+2, …, n+m}, A, b, c, 0})
    fi;
    form L_aux as explained previously;
    let (N, B, A, b, c, v) be the resulting slack form for L_aux;
    /* L_aux has n+1 nonbasic variables and m basic variables */
    (N, B, A, b, c, v) := Pivot(N, B, A, b, c, v, ℓ, 0);
    /* the basic solution is now feasible for L_aux */
    iterate the while loop of Simplex until an optimal solution to L_aux is found;
    if the basic solution sets x̄_0 = 0 then
        return final slack form with x̄_0 removed and original objective function
    else
        return "infeasible"
    fi
```

Example

$$\begin{array}{lrcrcr}
\text{maximize} & 2x_1 & - & x_2 & & \\
\text{subject to} & 2x_1 & - & x_2 & \leq & 2 \\
 & x_1 & - & 5x_2 & \leq & -4 \\
 & \multicolumn{5}{l}{x_1, x_2 \geq 0}
\end{array}$$

In this case, $b_\ell = -4$, so we must construct $\mathbf{L_{aux}}$:

$$\begin{array}{lrcrcrcr}
\text{maximize} & -x_0 & & & & & & \\
\text{subject to} & 2x_1 & - & x_2 & - & x_0 & \leq & 2 \\
 & x_1 & - & 5x_2 & - & x_0 & \leq & -4 \\
 & \multicolumn{7}{l}{x_1, x_2, x_0 \geq 0}
\end{array}$$

Example (Continued)

$$\begin{array}{rcrrrrrrr} z & = & & & & & & - & x_0 \\ x_3 & = & 2 & - & 2x_1 & + & x_2 & + & x_0 \\ x_4 & = & -4 & - & x_1 & + & 5x_2 & + & x_0 \end{array}$$

Basic solution to this would set $x_4 = -4$, but one call to Pivot creates a program with a feasible basic solution. Pivoting with $x_e = x_0$ and $x_\ell = x_4$ yields:

$$\begin{array}{rcrrrrrrr} z & = & -4 & - & x_1 & + & 5x_2 & - & x_4 \\ x_0 & = & 4 & + & x_1 & - & 5x_2 & + & x_4 \\ x_3 & = & 6 & - & x_1 & - & 4x_2 & + & x_4 \end{array}$$

Example (Continued)

$$\boxed{\begin{aligned} z &= -4 - x_1 + 5x_2 - x_4 \\ x_0 &= 4 + x_1 - 5x_2 + x_4 \\ x_3 &= 6 - x_1 - 4x_2 + x_4 \end{aligned}}$$

We now repeatedly call Pivot until we have an optimal solution to L_{aux}. Pivoting with $x_e = x_2$ and $x_\ell = x_0$ yields:

$$\boxed{\begin{aligned} z &= -x_0 \\ x_2 &= \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \end{aligned}}$$

Example (Continued)

$$
\begin{aligned}
z &= -x_0 \\
x_2 &= \frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\
x_3 &= \frac{14}{5} + \frac{4x_0}{5} - \frac{9x_1}{5} + \frac{x_4}{5}
\end{aligned}
$$

This is an optimal solution, and its basic solution sets $x_0 = 0$. We remove x_0 and restore the original objective function:

$$
\begin{aligned}
z &= -\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \\
x_2 &= \frac{4}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\
x_3 &= \frac{14}{5} - \frac{9x_1}{5} + \frac{x_4}{5}
\end{aligned}
$$

error in book here

$$
\begin{aligned}
& 2x_1 - x_2 \\
&= 2x_1 - \left(\frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5}\right) \\
&= -\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \quad \{\text{recall } x_0 = 0\} \\
&\text{(compute objective function)}
\end{aligned}
$$

Fundamental Theorem of LP

Theorem 29.13: Any linear program L, given in standard form, either

- has an optimal solution with a finite objective value,
- is infeasible, or
- is unbounded.

If L is infeasible, Simplex returns "infeasible." If L is unbounded, Simplex returns "unbounded." Otherwise, Simplex returns an optimal solution with a finite objective value.

LP Theory

Theory of Linear Optimization

Terminology:

- Feasible vector
- Feasible basic vector
- Optimal feasible vector

Faster LP Algorithms

- Ellipsoid. (Khachian 1979, 1980)
  - Solvable in polynomial time: $O(n^4 L)$ bit operations.
    - n = # variables
    - L = # bits in input
  - Theoretical tour de force.
  - Not remotely practical.
- Karmarkar's algorithm. (Karmarkar 1984)
  - $O(n^{3.5} L)$.
  - Polynomial and reasonably efficient implementations possible.
- Interior point algorithms.
  - $O(n^3 L)$.
  - Competitive with simplex!
    - Dominates on simplex for large problems.
  - Extends to even more general problems.

Theory

- *Feasible region* is convex and bounded by hyperplanes
- Feasible vectors that satisfy N of the original constraints are termed *feasible basic vectors.*
- Optimal occur at boundary (gradient vector of objective function is always nonzero)
- Combinatorial problem: determining which N constraints (out of the $N+M$ constraints) would be satisfied by the optimal feasible vector

Duality

- Primal problem

maximize $\mathbf{c}^T\mathbf{x}$

subject to $\mathbf{A}\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq 0$

- Dual problem

minimize $\mathbf{b}^T\mathbf{y}$

subject to $\mathbf{A}^T\mathbf{y} \geq \mathbf{c},\ \mathbf{y} \geq 0$

Duality

- If a linear program has a finite optimal solution then so does the dual. Furthermore, *the optimal values of the two programs are the same.*
- If either problem has an unbounded optimal solution, then the other problem has no feasible solutions.

Original	Dual
Maximize $u = 5x + 2y$, s.t.	Minimize $w = \boxed{12a + 9b + 20c}$, s.t.
$1x + 3y \le \boxed{12} \longleftarrow a$	$a + 3b + 7c \ge 5$
$3x - 4y \le \boxed{9} \longleftarrow b$	$3a - 4b + 8c \ge 2$
$7x + 8y \le \boxed{20} \longleftarrow c$	
$x, y \ge 0$	$a, b, c \ge 0$

Original constraints	Dual constraints
$\boxed{1}\,x + \boxed{3}\,y \le 12$	$\boxed{1a + 3b + 7c} \ge 5$
$\boxed{3}\,x - \boxed{4}\,y \le 9$	
$\boxed{7}\,x + \boxed{8}\,y \le 20$	$\boxed{3a - 4b + 8c} \ge 2$

But an Integer Program is different and much harder

1. Feasible region is a set of discrete points.
2. Can't be assured a corner point solution.
3. There are no "efficient" ways to solve an IP.
4. Solving it as an LP provides a relaxation and a bound on the solution.

Rounding ILP solutions can cause infeasibility

Maximize

$Z = x2$,

such that:

$-x1+x2 <= 1/2$

$x1 + x2 <= 3.5$

int $x1$, $x2 >= 0$.

From Hillier and Lieberman

Rounded ILP solutions can be very far from integer solutions

Maximize

$Z = x1 + 5\,x2$,

such that:

$x1 + 10\,x2 <= 20$

$x1 <= 2$

int $x1, x2 >= 0$.

From Hillier and Lieberman

LP Homework

- Download glpk (gnu linear programming package) and
- Solve all the problems discussed in class.

Solve with GLPK

```
C:\mosh> cat cormen29-3.mod
var x1 >= 0;
var x2 >= 0;
var x3 >= 0;
maximize obj: 3 * x1 + 1 * x2 + 2 * x3 ;
s.t. c1:  1 * x1 + 1 * x2 + 3 * x3 <= 30;
s.t. c2:  2 * x1 + 2 * x2 + 5 * x3 <= 24;
s.t. c3:  4 * x1 + 1 * x2 + 2 * x3 <= 36;
solve;
display x1, x2, x3;
end;
```

- C:\> glpsol --math cormen29-3.mod

```
GLPSOL: GLPK LP/MIP Solver, v4.47
Parameter(s) specified in the command line:
 --math cormen29-3.mod
Reading model section from cormen29-3.mod...
18 lines were read
Generating obj...
Generating c1...
Generating c2...
Generating c3...
Model has been successfully generated
GLPK Simplex Optimizer, v4.47
4 rows, 3 columns, 12 non-zeros
Preprocessing...
3 rows, 3 columns, 9 non-zeros
                    Scaling...
 A: min|aij| =  1.000e+00  max|aij| =  5.000e+00  ratio =  5.000e+00
Problem data seem to be well scaled
Constructing initial basis...
Size of triangular part = 3
*     0: obj =   0.000000000e+00  infeas =  0.000e+00 (0)
*     3: obj =   2.800000000e+01  infeas =  0.000e+00 (0)
OPTIMAL SOLUTION FOUND
Time used:   0.0 secs
Memory used: 0.1 Mb (108677 bytes)
Display statement at line 17
x1.val = 8
x2.val = 4
x3.val = 0
1.      Model has been successfully processed
```

Solve with Mathematica

```
In[9] = Maximize[  {
3 x1 +    x2 + 2 x3,
  x1 +    x2 + 3 x3  <= 30 &&
2 x1 + 2 x2 + 5 x3  <= 24 &&
4 x1 +    x2 + 2 x3  <= 36,
x1 >= 0 && x2 >= 0 && x3 >= 0 },
{ x1, x2,x3}  ]

Out[9] = {28, {x1 -> 8, x2 -> 4, x3 -> 0}}
```

Mathematica Interior Point

```
In = LinearProgramming[ {-3, -1, -2},
 {{-1, -1, -3}, {-2, -2, -5}, {-4, -1, -2}},
  {-30, -24, -36},
  Method -> "InteriorPoint"]
```

Out = {8., 4., $9*10^{-7}$}

Solve with Maple V.4 simplex

```
Maple> with(simplex);
maximize(3*x1+x2+2*x3, {
     x1+    x2+3*x3<=30,
  2*x1+2*x2+5*x3<=24,
  4*x1+   x2+2*x3<=36
  }, NONNEGATIVE);
```

{x3 = 0, x1 = 8, x2 = 4}

Using python pysimplex

```
C:\> cat  example.py

from pysimplex import *
if __name__ == '__main__':
    set_printoptions(precision=2)
    t = Tableau([-3,-1,-2])              # max z = 3x + 1y + 2z
    t.add_constraint([1, 1, 3], 30)  # x + y + 3z <=  30
    t.add_constraint([2, 2, 5], 24)  # 2x + 2y + 5z <= 24
    t.add_constraint([4, 1, 2], 36)  # 4x + 1y + 2z <= 36
    t.solve(5)
```

Output of example.py

```
-------------------------
solve:
[[ -3 -1 -2 0 0 0  0]
 [  1  1  3 1 0 0 30]
 [  2  2  5 0 1 0 24]
 [  4  1  2 0 0 1 36]]
ratios: [30, 12, 9]
-------------------------
step=1, pivot column: 2, row: 4
[[ 0 -0.25 -0.5 0 0  0.75   27]
 [ 0  0.75  2.5 1 0  -0.25  21]
 [ 0  1.5   4.  0 1  -0.5    6]
 [ 1  0.25  0.5 0 0  0.25    9]]
ratios: [8.4, 1.5, 18]
-----------------------------
-------------------------
step=2, pivot column: 4, row: 3
[[  0 -0.06 0 0  0.13   0.69  27.75]
 [  0 -0.19 0 1 -0.63   0.06  17.25]
 [  0  0.38 1 0  0.25  -0.13   1.5 ]
 [  1  0.06 0 0 -0.13   0.31   8.25]]
ratios: [1724999982.75, 4, 132]
-----------------------------
step=3, pivot column: 3, row: 3
[[  0 0  0.17 0  0.17   0.67  28 ]
 [  0 0  0.5  1   -0.5    0.   18 ]
 [  0 1  2.67 0  0.67  -0.33   4 ]
 [  1 0 -0.17 0 -0.17   0.33   8 ]]
Solved in  4 steps
-----------------------------
```

Solving LP with MS Excel

LP example to help a farmer

We have following linear constraints:

- 6 acres of land: $3x + y \leq 6$
- 6 tons of fertilizer: $2x + 3y \leq 6$
- 8 hour work day: $x + 5y \leq 8$
- Apples sell for twice as much as oranges
- We want to maximize profit: $z = 2x + y$
- Production is positive: $x \geq 0$, $y \geq 0$

Enter the data into Excel

x : apple
y : orange
Price apple = 2 price orange
maximize profit: $z = 2x + y$

Constraints:
6 acres of land: $3x + y \leq 6$
6 tons of fertilizer: $2x + 3y \leq 6$
8 hour work day: $x + 5y \leq 8$

Production is positive: $x \geq 0$, $y \geq 0$

	A	B	C	D	E	F	G
1	maximize	0					
2	var	x	y				
3	profit	2	1				
4	values	0	0				
5				total		limit	
6	c1	3	1	0	<=	6	land
7	c2	2	3	0	<=	6	fertilizer
8	c3	1	5	0	<=	8	labour
9							

Naming a cell

- Select B1
- Click on "*name box*",
- Type *"profit"*,
- Press *"Control-Shift-Enter"* to name B1 profit.

Input the objective function, (equation for profit)

Select cell B1, Click on ***fx***, Select *sumproduct*, Array1=B3:C3
Array2=B4:C4

So profit is
z = 2 * x + 1 * y
B1 = B3 * B4 + C3 * C4

Enter the formula for constraints

- D6=SUMPRODUCT(B6:C6,$B$4:$C$4)
- D7=SUMPRODUCT(B7:C7,$B$4:$C$4)
- D8=SUMPRODUCT(B8:C8,$B$4:$C$4)

Tools > Solver

	A	B	C	D	E	F	G
1	maximize	0					
2	var	x	y				
3	profit	2	1				
4	values	0	0				
5				total		limit	
6	c1	3	1	0	<=	6	land
7	c2	2	3	0	<=	6	fertilizer
8	c3	1	5	0	<=	8	labour

Solver > Options

- Defaults solver options are good.

Solved

- x=1.714
- y=0.857
- z = 4.28

Answer report

Microsoft Excel - farmer.xls

File Edit View Insert Format Tools Data Window He

B25

Microsoft Excel 11.0 Answer Report
Worksheet: [farmer.xls]Sheet1
Report Created: 04/29/13 7:43:00 PM

Target Cell (Max)

Cell	Name	Original Value	Final Value
$B$1	profit	4.285714286	4.285714286

Adjustable Cells

Cell	Name	Original Value	Final Value
$B$4	values x	1.714285714	1.714285714
$C$4	values y	0.857142857	0.857142857

Constraints

Cell	Name	Cell Value	Formula	Status	Slack
$D$6	constraint1 total	6	$D$6<=$F$6	Binding	0
$D$7	constraint2 total	6	$D$7<=$F$7	Binding	0
$D$8	constraint3 total	6	$D$8<=$F$8	Not Binding	2

Answer Report 1 / Sensitivity Report 1 / Limits Report 1 / Sheet1

Ready

Sensitivity report

Microsoft Excel - farmer.xls

File Edit View Insert Format Tools Data Window Help

C21

Microsoft Excel 11.0 Sensitivity Report
Worksheet: [farmer.xls]Sheet1
Report Created: 04/29/13 7:43:00 PM

Adjustable Cells

Cell	Name	Final Value	Reduced Cost	Objective Coefficient	Allowable Increase	Allowable Decrease
$B$4	values x	1.714285714	0	2	1	1.333333333
$C$4	values y	0.857142857	0	1	2	0.333333333

Constraints

Cell	Name	Final Value	Shadow Price	Constraint R.H. Side	Allowable Increase	Allowable Decrease
$D$6	constraint1 total	6	0.571428571	6	3	2
$D$7	constraint2 total	6	0.142857143	6	1	2
$D$8	constraint3 total	6	0	8	1E+30	2

Answer Report 1 | Sensitivity Report 1 | Limits Report 1 | Sheet1

Ready

Limits Report

Microsoft Excel - farmer.xls

File Edit View Insert Format Tools Data Window Help

D17

Microsoft Excel 11.0 Limits Report
Worksheet: [farmer.xls]Limits Report 1
Report Created: 04/29/13 7:43:00 PM

Cell	Target Name	Value
$B$1	profit	4.285714286

Cell	Adjustable Name	Value	Lower Limit	Target Result	Upper Limit	Target Result
$B$4	values x	1.714285714	0	0.857142857	1.714285714	4.285714286
$C$4	values y	0.857142857	0	3.428571429	0.857142857	4.285714286

Answer Report 1 | Sensitivity Report 1 | Limits Report 1 | Sheet1

Ready

Aproximation Algorithms

Cormen chapter 35

NP-completeness is not the end

Coping With NP-Hardness

Brute-force algorithms.

- Develop clever enumeration strategies.
- Guaranteed to find optimal solution.
- No guarantees on running time.

Heuristics

Develop intuitive algorithms.
Guaranteed to give some correct solution in polynomial time.
No guarantees on quality of solution.

Approximation algorithms (AA)

- Guaranteed to run in polynomial time.
- Guaranteed to find "high quality" solution, say within 1% of optimum.
  How to prove a solution's value is close to optimum,
  without even knowing what optimum value is?

Approaches

Special cases of graph may have faster algorithms

e.g. vertex cover for bipartite graphs.

Fixed parameter problems, may have faster algorithms

e.g. graph coloring, for k=2.

Average case

find an algorithm which works well on average,

e.g. simplex for LP.

Approximation algorithms

find an algorithm which return solutions that are guaranteed to be close to an optimal solution.

Approx ratio $\rho(n)$

An AA is bounded by $\rho(n)$ (Rho) if cost of the AA solution c (for input size n) is within a factor $\rho(n)$ of the optimal c*, that is:

An AA has an **approximation ratio** $\rho(n)$ if for any input of size n

$$\max(C/C^*, C^*/C) \leq \rho(n)$$

where C = cost of solution produced by the AA.

C^* = cost of optimal solution > 0

for both minimization and maximization problems.

Definitions: PTAS, FPTAS

Approximate Scheme: AA that takes $\varepsilon > 0$ as input and is a $(1 + \varepsilon)$-approximation algorithm.
That is, we can get as near to the optimal as we wish.

PTAS (Polynomial-time approximate scheme):
For **fixed** ε, AA scheme runs in polynomial time for input size n. That is, a constant decrease in ε, will result in a constant increase in running time.

Fully PTAS: polynomial time in n and $1/\varepsilon$.
E.g. of FPTAS: $O((1/\varepsilon)^2 * n^3)$

AA examples

- Vertex cover, using greedy
- TSP⇔, using MST.
- TSP, Hamiltionian path, NO AA.
- Set Cover,
- 3CNF, randomized 7/8 algorithm.
- Weighted vertex cover, using LP.
- Subset sum (partition), FPTAS divide and conquer.

Aproximation Algorithm for Vertex Cover using LP

Cormen chapter 35

- **Approximating weighted vertex cover using linear programming.**
- Minimum-weight VC problem:

  Given an undirected graph G = (V, E), where each $v \in$ V has an associated positive weight w(v). For any vertex cover V', w(V') = $\sum_{v \in V'} w(v)$
  The goal is to find a vertex cover of minimum weight.
- Associate a variable $x(v)$ with each $v \in$ V, and let $x(v) \in \{0, 1\}$.

  $x(v)$ = 1 means v is in the VC; 0 0/w.
- For each edge (u, v), at least one of u and v must be in VC.

  Thus $x(u) + x(v) \geq 1$.
- 0-1 linear program:

$$\min \sum_{v \in V} w(v)x(v)$$

Subject to $x(u) + x(v) \geq 1$ for each $(u, v) \in$ E

$x(v) \in \{0, 1\}$ for each v

- Linear-programming relaxation:

$$\min \sum_{v \in V} w(v)x(v)$$

s.t. $x(u) + x(v) \geq 1$ for each $(u, v) \in$ E

$x(v) \leq 1$ for each $v \in$ V

$x(v) \geq 0$ for each $v \in$ V

- **Approximating weighted vertex cover using linear programming.**
- Minimum-weight VC problem:

$$\sum_{v \in V'} w(v)$$

Given an undirected graph G = (V, E), where each $v \in$ V has an associated positive weight w(v). For any vertex cover V', w(V') = The goal is to find a vertex cover of minimum weight.

$$\sum_{v \in V'} w(v)x(v)$$

- Associate a variable $x(v)$ with each $v \in$ V, and let $x(v) \in \{0, 1\}$.

  $x(v)$ = 1 means v is in the VC; 0 0/w.

$$\sum_{v \in V'} w(v)x(v)$$

- For each edge (u, v), at least one of u and v must be in VC.

  Thus $x(u) + x(v) \geq 1$.

- 0-1 linear program:

  min $\sum_{v \in V'} w(v)$

  Subject to $x(u) + x(v) \geq 1$ for each $(u, v) \in E$

  $x(v) \in \{0, 1\}$ for each v

- Linear-programming relaxation:

  min $\sum_{v \in V} w(v)x(v)$

  s.t. $\sum_{v \in V} w(v)x(v)$ $x(u) + x(v) \geq 1$ for each $(u, v) \in E$

  $x(v) \leq 1$ for each $v \in V$

  $x(v) \geq 0$ for each $v \in V$

Aproximation Algorithms for Vertex Cover

Cormen chapter 35

Vertex Cover

Vertex cover: a subset of vertices which "covers" every edge.
An edge is covered if one of its endpoint is chosen.

The Minimum Vertex Cover Problem:
Find a vertex cover with minimum number of vertices.

Vertex Cover AA

```
Approx-VC(G)
    C = { }
    E′ = E[G]
    while E′ ≠ { } do
        pick edge (u,v) in E′
        C = C ∪ {u,v};
        remove from E′ every edge incident on either u or v
    return C // cover.
```

Example: Vertex cover AA

Example

Example

Example

Example

Vertex cover produced by the algorithm
(6 vertices in VC)

Example

Optimal vertex cover
(3 vertices in VC)

VC AA Ratio is 2

VC AA finds upto twice the optimal number of vertices.

Proof:

Let A = set of edges selected by AA.

Notice: That no two edges in A have a common endpoint,

So our solution size is $|C| = 2|A|$.

The optimal cover C* must include one the two endpoints.

hence $|A| \leq |C^*|$.

Combining: $|C| = 2|A| \leq 2|C^*|$.

So our solution is at most twice the optimal. QED

Note: We don't know the optimal value, but we do know lower bound on C*.

Aproximation Algorithm for TSP

Cormen chapter 35

⇔ Triangle Inequality

Given complete, undirected graph.
For all vertices, u,v,w, require:

$$c(u,w) \leq c(u,v) + c(v,w) \ldots \Leftrightarrow \text{inequality}$$

Distances on the euclidean plane obey the triangle inequality.

TSP⇔ (with Triangle Inequality)

TSP⇔ remains NP-complete even with ⇔inequality.

Note that a TSP tour with any one edge removed is a MST of the graph

Approx-TSP is O(V^2), solution is within twice the optimal.

```
Approx- TSP(G,c)
    select r ∈ v[G]
    call MST-Prim(G,c,r) to construct MST with root r
    let  L = vertices on preorder walk of MST
    // This step needs triangle inequality to skip duplicate visits.
    let H = cycle that visits vertices in the order L
    return H
```

Example TSP AA

Example TSP MST prim

Example: Pre-order walk of MST

Example: TSP solution

Computed Tour

Example: TSP optimal tour

The Optimal tour is 23% shorter, than the AA

Claim: Approximation Ratio is 2

Proof:

Let H^* = optimal tour, T = MST.
Deleting an edge of H^* yields a spanning tree.
So, $c(T) \leq c(H^*)$.

Consider **full walk** W of T.
Ex: a,b,c,b,h,b,a,d,e,f,e,g,e,d,a.
Visits each edge twice

$$\Rightarrow c(W) = 2c(T).$$

Hence, $c(W) \leq 2c(H^*)$.

Triangle inequality $\Rightarrow$ can delete any vertex from W and cost doesn't increase.

Deleting all but first appearance of each vertex yields **preorder walk**.
Ex: a,b,c,h,d,e,f,g.

Corresponding cycle H = W with some edges removed.
Ex: a,b,c,h,d,e,f,g,a.

$c(H) \leq c(W)$.

Implies $c(H) \leq 2c(H^*)$.

General TSP has no AA

<u>Theorem 35.3:</u> If $P \neq NP$ and approximation ratio $\rho \geq 1$,
There is no polynomial-time AA with ρ for the general TSP.

<u>Proof:</u>

Suppose $\exists$ polynomial-time a.a. A with approximation ratio ρ.

WLOG, assume ρ is an integer.

We use A to solve the Hamiltonian Circuit (HC) Problem
in polynomial time.

Proof: solving HC using TSP AA.

Given G =(V,E) to find a HCP,
Create a new complete graph G' of G
$G' = (V,E')$ be the complete graph on V.
For each $(u,v) \in E'$, the cost is:

$$c(u,v) = \begin{cases} 1 & \text{if } (u,v) \in E \\ \rho|V|+1 & \text{otherwise (very large)} \end{cases}$$

We run TSP AA on instance (G',c).

Proof Continued

If G has a H.C., then G′ has a tour of cost |V|.

If G does not have a H.C., then any tour in G′ uses an edge not in E, and costs at least $(\rho|V|+1)+(|V|-1) > \rho|V|$.

Now, consider running AA on (G′,c), to find
a tour of cost at most ρ times cost of optimal.

Thus, if G has a HC, A must return it.
If G has no HC, then A returns a tour of cost $> \rho|V|$.

Therefore, can use A to solve HC in polynomial time.
But HC is NPC, [Contradiction] so this is a impossible,
so TSP AA is also impossible.

Randomization: MAX-3CNF

A randomized algorithm has an approximation ratio of $\rho(n)$ as before, but C is interpreted as an expected cost.

Example: MAX-3CNF Satisfiability. Given an expression in 3CNF, want an assignment that makes as many clauses as possible true.

Assume: Each clause consists of exactly three distinct literals, and no clause contains both a variable and its negation.

Approximation Algorithm: Randomly (with equal probability of 0 and 1) assign values to variables.

Claim: Approximation Ratio is 8/7

Proof:

First, note that

P[clause i satisfied]

= 1 – P[all 3 literals are 0]

= 1 – $(1/2)^3$ = 7/8.

Define the indicator random variable Y_i = I{clause i is satisfied}.

Then, $E[Y_i]$ = P[clause i satisfied]

= 7/8.

Let $Y = \Sigma_i Y_i$.

Expected number of satisfied clauses is:

$$E[Y] = E\left[\sum_{i=1}^{m} Y_i\right]$$

$$= \sum_{i=1}^{m} E[Y_i]$$

$$= \sum_{i=1}^{m} 7/8$$

$$= 7m/8$$

Optimal solution is upper bounded by m. So, approx. ratio is at most m/(7m/8) = 8/7.

Approx Algorithm for Set covering

SC Problem and example

Given a set X and a F family of subset of X.
F = {f1,f2,...}, Union(F) = X
Find C, the minimal subset of F : Union(C) = X.
This problem is **NP-Complete**.

Example: Given: X = {1,2,3,4}
F = { {1,2},{1,2,3},{2,3},{2,4}, {1,4} }
One solution: C = { {1,2,3}, {2,4} }
Size(C)=2, Size(F) = 4.
Note: Union(F) = Union(C) = X.
F is also a solution but not the smallest.

SC Example

Input: X, F

Output: C set cover of X

Set Cover: Greedy AA algorithm

Family of Sets.
Each set consists of elements from X.

```
Greedy-SC( X, F)
    U = X
    C = { }
    while U ≠ { } do
        select S ∈F that maximizes |S ∩ U|;
        U = U – S;
        C = C ∪ {S}
    do;
    return C
```

Example

Example

Example

Example

Example

Computed cover consists of 4 sets.

Example

Optimal cover consists of **3 sets**.

Claim: Approx. Ratio is Logarithmic

Let $H(d)$ be d^{th} harmonic number

$$\sum_{i=1,\ldots,d} 1/i = 1 + 1/2 + 1/3 + \ldots + 1/d = O(\ln(d))$$

Claim: Greedy-SC has an approx. ratio of $H(\max\{|S| : S \in F\})$.
Note: approx. ratio is $\ln|X|+1$

Proof: Let
C = set cover returned by the algorithm.
C^* = optimal S.C.
S_i = i^{th} set selected by the algorithm.

Claim: Approx. Ratio is Logarithmic

Let cost of C be |C|, i.e., **charge** 1 for each S_i.

Distribute this cost to elements of X.

If $x \in X$ is covered for the first time by S_i, then

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})\}|}$$

cost of x

no. of elements covered for the 1st time by S_i

no of elements covered before step i.

Example: S1 has cost 1/6 per 6 elements covered:

Computed cover consists of 4 sets.

Example: S3 has cost 1/3 per 3 new elements covered

Computed cover consists of 4 sets.

Example S2 has cost 1/2 for 2 new elements covered

Computed cover consists of 4 sets.

Example: S4 has cost 1 for the last element covered

Computed cover consists of 4 sets = {S1,S3,S2,S4}

Proof continued

We have the following bound on |C|:

$$|C| = \sum_{x \in X} c_x$$

$$\leq \sum_{S \in C^*} \sum_{x \in S} c_x$$

This follows because $x \in X$ may appear in multiple sets in C*.
And each such x appears in at least one such set.(See example.)

Proof Continued

Claim: For any S in F, total cost

$$\sum_{x \in S} c_x \leq H(|S|)$$

By claim:

$$|C| \leq \sum_{S \in C^*} H(|S|)$$

$$\leq |C^*| \cdot H(\max\{|s|: s \in F\})$$

So, approx. ratio is as claimed.

Proof of Claim

Consider any $S \in F$.

For $i = 1,2, \ldots , |C|$, let

$u_i = | S - (S_1 \cup S_2 \cup \ldots \cup S_i)|$
$=$ no. of uncovered elements in S after $S_1,\ldots,S_i$ have been selected.

Let $u_0 = |S|$.

Let k be s.t. S completely covered after S_k.

$u_{i-1} - u_i =$ no. of elements of S covered for the first time by S_i.

Proof of Claim

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup \cdots \cup S_{i-1})|}$$

Greedy Choice $\Rightarrow$

$$|S_i - (S_1 \cup \cdots \cup S_{i-1})| \geq |S - (S_1 \cup \cdots \cup S_{i-1})|$$
$$= u_{i\text{-}1}$$

Thus,

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i\text{-}1}}$$

For $a < b$,

$$H(b) - H(a) = \sum_{i=a+1}^{b} \frac{1}{i}$$
$$\geq (b-a)\frac{1}{b}$$

(For $a = b > 0$, $H(b) - H(a) = (b-a)(1/b)$.)

Thus,

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i))$$
$$= H(u_0) - H(u_k)$$
$$= H(u_0) - H(0)$$
$$= H(u_0)$$
$$= H(|S|)$$

Aproximation Algorithm for Subset sum

Cormen chapter 35

Recall: Bin packing problem is NPC

Can k bins of capacity t each, store a finite set of weights S = {x1,..,xn}?

Example:
We have k=4 bins with capacity t=10 each.
and weights S = {1,9,3,7,6,4,1,2,2,5}.
How to pack S into k bins?

Subset-Sum Problem

Problem: Pack as many of S into box of size t, without overflow.

ie. Find one subset of $S=\{x_1, \ldots, x_n\}$ (positive integers) whose sum is as large as possible but not larger than the capacity t.

Subset-Sum Brute force Exp Algorithm

<u>Notation:</u>

If L = [y1,…,yk], then L + x means [y1 + x, … ,yk + x].

If S is a set, then S + x means { s + x: s in S}.

```
Exact-SubsetSum(S, t)   // Exp because Length(L) can be 2^i.
    n = |S|
    L_0 = [0]
    for i = 1 to n do
        L_i = Merge (L_{i-1}, L_{i-1} + x_i );   /* like Merge Sort */
        Remove from L_i every element  > t
    return largest element in L_n.
```

Example

L_i = Merge (L_{i-1}, L_{i-1} + x_i); /* like Merge Sort */

E.g. if L1 = [0, 1, 2] and x2 = 10 then

L2 = Merge(L1, L1 + 10)

L2 = Merge([0,1,2], [10,11,12])

L2 = [0, 1, 2, 10, 11, 12]

So the list size can double in each step

A Fully-PT-Approximation-Scheme

Goal: Modify exponential algorithm to make it a fully polynomial-time approximation scheme.

Idea: "Trim" elements from L_i that are near other elements.
Given: trim parameter δ, $0 < \delta < 1$. (will depend on ε.)
Remove y from list if there is z nearby it: $\exists\, z : y/(1+\delta) \le z \le y$.

Example: If $\delta = 0.1$ and L = [10,11,12,15,20,21,22,23,24,29],
trim(L) = L′ = [10,12,15,20,23,29].

Intuition: Deleted values 21 & 22 are approximated by 20.

Trim the list L of nearby numbers

```
Trim (L, δ)          // L = [y_1, y_2, …, y_m]
    m =|L|;
    L′  = [y_1];
    last = y_1;
    for i = 2 to m do
        if  y_i > last·(1 + δ) then
            append y_i onto end of L′;
            last = y_i
        fi
    od
    return L′
```

Approx Subset Sum

Example:

S = {104,102,201,101}, $\varepsilon = 0.40$, $\delta = 0.05$
t = 308

line 2: L_0 = <0>

line 4: L_1 = <0, 104>
line 5: L_1 = <0, 104>
line 6: L_1 = <0, 104>

line 4: L_2 = <0, 102, 104, 206>
line 5: L_2 = <0, 102, 206>
line 6: L_2 = <0, 102, 206>

line 4: L_3 = <0, 102, 201, 206, 303, 407>
line 5: L_3 = <0, 102, 201, 303, 407>
line 6: L_3 = <0, 102, 201, 303>

line 4: L_4 = <0, 101, 102, 201, 203, 302, 303, 404>
line 5: L_4 = <0, 101, 201, 302, 404>
line 6: L_4 = <0, 101, 201, 302>
Algorithm returns 302.
Optimal answer is 307 = 104 + 102 + 101.

```
Approx-Subset-Sum(S, t, ε)
1   n = |S|;
2   L_0 = [0];
3   for I =1 to n do
4       L_i = Merge (L_{i-1}, L_{i-1}+x_i);
5       L_i = Trim (L_i, ε/2n);
6       remove from L_i every element > t
    od;
7   return largest element in L_n
```

Theorem: Approx-SS is a fully-PTAS.

Proof: We consider the solution space Pi at step i.
Let P_i = set of all values that can be obtained by selecting a (possibly empty) subset of $\{x_1, x_2, \ldots, x_i\}$ and summing its members.

In Exact-SS, $L_i = P_i$ (with items > t removed).

In Approx-SS, ✂ $y \in P_i$ $(y \bullet t)$,
$\exists\, z \in L_i : \; y/(1+\varepsilon/2n)^i \leq z \leq y.$

Proof: By induction on i.

Proof continued

Let $y^* \in P_n$ denote the optimal solution, and let z^* denote the solution returned by SSAA. Then, $z^* \leq y^*$.
TPT. $\mathbf{y^*/z^* \leq 1 + \varepsilon}$.

By previous observation, $\exists\, z \in L_n$ s.t.
$y^*/(1+ \varepsilon/2n)^n \leq z \leq y^*$, and hence, $y^*/z \leq (1+ \varepsilon/2n)^n$.
Because z^* is the largest value in L_n, we have $y^*/z^* \leq (1+ \varepsilon/2n)^n$.

$d/dn\ (1+ \varepsilon/2n)^n > 0$, so $(1+ \varepsilon/2n)^n$ increases with n. Its limit is $e^{\varepsilon/2}$.
Thus,

$$
\begin{aligned}
(1+ \varepsilon/2n)^n &\leq e^{\varepsilon/2} \\
&\leq 1 + \varepsilon/2 + (\varepsilon/2)^2 && \{e^x \leq 1 + x + x^2 \text{ — see Ch. 3}\} \\
&\leq 1 + \varepsilon && \{0 < \varepsilon < 1\}
\end{aligned}
$$

Proof: Time complexity

Show time complexity is polynomial in length[I] and $1/\varepsilon$.
Want to bound the length of L_i.
After trimming, successive z and z′ satisfy $z'/z > 1 + \varepsilon/2n$.
Thus, each list contains [0 to $\lfloor \log_{1+\varepsilon/2n} t \rfloor$] other values.
So, time complexity is polynomial in length(I) and $1/\varepsilon$.
Because:

$$\log_{1+\varepsilon/2n} t = \frac{\ln t}{\ln(1+\varepsilon/2n)}$$

$$\leq \frac{2n(1+\varepsilon/2n)\ln t}{\varepsilon} \qquad \{x/(1+x) \leq \ln(1+x) \text{ -- see Ch. 3}\}$$

$$\leq \frac{4n \ln t}{\varepsilon} \qquad \{0 < \varepsilon < 1\}$$

TSP: Travelling salesman problem

TSP

Given:

Set of cities $\{c_1, c_2, \ldots, c_N\}$.

For each pair of cities $\{c_i, c_j\}$, a distance $d(c_i, c_j)$.

Find: Permutation Pi of {1..n}
that minimizes the total

$$\sum_{i=1}^{N-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(N)}, c_{\pi(1)})$$

Example of TSP instances

n=1000

n=100

n=10000

Other Instances of TSP

- X-ray crystallography
  - Cities: orientations of a crystal
  - Distances: time for motors to rotate the crystal from one orientation to the other
- High-definition video compression
  - Cities: binary vectors of length 64 identifying the summands for a particular function
  - Distances: Hamming distance (the number of terms that need to be added/subtracted to get the next sum)

No-Wait Floor shop Scheduling

- Cities: Length-4 vectors $\langle c_1,c_2,c_3,c_4 \rangle$ of integer task lengths for a given job that consists of tasks that require 4 processors that must be used in order, where the task on processor i+1 must start as soon as the task on processor i is done).
- Distances: $d(c,c')$ = Increase in the finish time of the 4th processor if c' is run immediately after c.
- Note: Not necessarily symmetric: may have $d(c,c') \neq d(c',c)$.

How Hard is TSP?

- NP-Hard for all the above applications and many more
  - [Karp, 1972]
  - [Papadimitriou & Steiglitz, 1976]
  - [Garey, Graham, & Johnson, 1976]
  - [Papadimitriou & Kanellakis, 1978]

How Hard?

Number of possible tours:

$N! = 1 \times 2 \times 3 \times \cdots \times (N-1) \times N = \Theta(2^{N \log N})$

$10! = 3{,}628{,}200$

$20! \sim 10^{18}$ (quadrillion)

Dynamic Programming Solution:

$O(N^2 * 2^N) = o(2\text{^}(N*\log N))$

Dynamic Programming Algorithm

- For each subset C' of the cities containing c_1, and each city $c \in C'$,
- let $f(C',c)$ = Length of shortest path from c1 to c that is a permutation of C'. So $f(\{c_1\}, c_1) = 0$
- For $x \notin C'$, $f(C' \cup \{x\},x) = \mathrm{Min}_{c \in C'} f(C',c) + d(c,x)$.
- Optimal tour length $= \mathrm{Min}_{c \in C} f(C,c) + d(c, c_1)$.
- Running time: $\sim(N-1)2^{N-1}$ items to be computed, at time N for each $= O(N^2 2^N)$

DP is how Hard?

Number of possible tours:

$N! = 1\times2\times3\times\cdots\times(N-1)\times N = \Theta(2^{N\log N})$

$10! = 3{,}628{,}200$

$20! \sim 2.43\times10^{18}$ (2.43 quadrillion)

Dynamic Programming Solution:

$O(N^2 2^N)$

$10^2 2^{10} = 102{,}400$

$20^2 2^{20} = 419{,}430{,}400$

Planar Euclidean Application #1

- Cities: Holes to be drilled in printed circuit boards

N = 2392

Planar Euclidean Application #2

Cities: Wires to be cut in a "Laser Logic" programmable circuit

N = 7397

N = 33,810

N = 85,900

Standard Approach to Coping with NP-Hardness:

- Approximation Algorithms
  - Run quickly (polynomial-time for theory, low-order polynomial time for practice)
  - Obtain solutions that are guaranteed to be close to optimal
  - Euclidean TSP, the triangle inequality holds: $d(a,c) \leq d(a,b) + d(b,c)$

Travelling Salesman Problem (TSP)

A Salesman wishes to travel around a given set of cities, and return to the beginning, covering the smallest total distance

Easy to State

Difficult to Solve

If there is no condition to return to the beginning. It can still be regarded as a TSP.
Suppose we wish to go from A to B visiting all cities.

If there is no condition to return to the beginning. It can still be regarded as a TSP.

Connect A and B to a ‘dummy’ city at zero distance
(If no stipulation of start and finish cities connect all to dummy at zero distance)

If there is no condition to return to the beginning. It can still be regarded as a TSP.

Create a TSP Tour around all cities

A route returning to the beginning is known as a
Hamiltonian Circuit

A route not returning to the beginning is known as a
Hamiltonian Path

Essentially the same class of problem

Applications of the TSP

Routing around Cities

Computer Wiring - connecting together computer components using minimum wire length

Archaeological Seriation - ordering sites in time

Genome Sequencing - arranging DNA fragments in sequence

Job Sequencing - sequencing jobs in order to minimise total set-up time between jobs

Wallpapering to Minimise Waste

Note: First three applications are *symmetric*, Last three *asymmetric*

Major Practical Extension of the TSP

Vehicle Routing - Meet customers demands within given *time windows* using lorries of limited capacity

Much more difficult than TSP

History of TSP

1800's	**Irish Mathematician, Sir William Rowan Hamilton**
1930's	**Studied by Mathematicians Menger, Whitney, Flood etc.**
1954	**Dantzig, Fulkerson, Johnson, 49 cities (capitals of USA states) problem solved**
1971	**64 Cities**
1975	**100 Cities**
1977	**120 Cities**
1980	**318 Cities**
1987	**666 Cities**
1987	**2392 Cities (Electronic Wiring Example)**
1994	**7397 Cities**
1998	**13509 Cities (all towns in the USA with population > 500)**
2001	**15112 Cities (towns in Germany)**
2004	**24978 Cities (places in Sweden)**

But many smaller instances not yet solved (to proven optimality)

Recent TSP Problems and Optimal Solutions

from

Web Page of William Cook, Georgia Tech, USA

with Thanks

Printed Circuit Board 2392 cities **1987 Padberg and Rinaldi**

2004

n=24978

USA Towns of 500 or more population 13509 cities 1998 Applegate, Bixby, Chvátal and Cook

Towns in Germany 15112 Cities

2001Applegate, Bixby, Chvátal and Cook

Sweden 24978 Cities 2004 Applegate, Bixby, Chvátal, Cook and Helsgaun

Solution Methods

I.	**Try every possibility**	(n-1)! possibilities – grows faster than exponentially

If it took 1 microsecond to calculate each possibility
takes 10^{140} centuries to calculate all possibilities when n = 100

II.	**Optimising Methods**	obtain **guaranteed** optimal solution, but can take a very, very, long time
III.	**Heuristic Methods**	obtain 'good' solutions 'quickly' by intuitive methods. No guarantee of optimality

(Place problem in newspaper with cash prize)

The Nearest Neighbour Method (Heuristic) – A ‘Greedy’ Method

1. Start Anywhere
2. Go to Nearest Unvisited City
3. Continue until all Cities visited
4. Return to Beginning

A 42-City Problem The Nearest Neighbour Method (Starting at City 1)

The Nearest Neighbour Method (Starting at City 1)

The Nearest Neighbour Method (Starting at City 1)

Length 1498

Remove Crossovers

Remove Crossovers

Remove Crossovers Length 1453

Christofides Method (Heuristic)

1. Create Minimum Cost Spanning Tree (Greedy Algorithm)

2. ‘Match’ Odd Degree Nodes

3. Create an Eulerian Tour - Short circuit cities revisited

Christofides Method 42 – City Problem

Minimum Cost Spanning Tree

Length 1124

Minimum Cost Spanning Tree by Greedy Algorithm

Match Odd Degree Nodes

Match Odd Degree Nodes in Cheapest Way – Matching Problem

1. Create Minimum Cost Spanning Tree (Greedy Algorithm)

2. ‘Match’ Odd Degree Nodes

3. Create an Eulerian Tour - Short circuit cities revisited

Create a Eulerian Tour – Short Circuiting Cities revisited

Length 1436

Optimising Method

1.Make sure every city visited once and left once – in cheapest way (Easy)

-The Assignment Problem **- Results in subtours** Length 1098

Put in extra constraints to remove subtours (More Difficult)

Results in new subtours Length 1154

Remove new subtours

Results in further subtours Length 1179

Further subtours Length 1189

Further subtours Length 1192

Further subtours Length 1193

Optimal Solution Length 1194

Command Line shells

- Cmd on Windows
- Bash on Linux (and cygwin on Windows)

CMD Command line

Windows Disk consist of drives:

C:\ D:\ E:

And each disk has directories, e.g. C:\windows **and** C:\Documents and Settings\a\Desktop

And each directory contains files.

Files have a name and extension, e.g. C:\WINDOWS\system32\system.ini

Create shortcut for cmd

- Right click -> new -> shortcut -> cmd
- Right click -> properties -> quick-edit

Cmd (console) window
Only text, no graphics

- Start -> Run -> cmd

cd and dir commands

Cmd icon and name

Buttons to
- Minimize
[] Maximize
X Close window

Title bar

Prompt
Shows the current dir

Cursor to type next command

Command and argument

Scroll bar

Folders and files

Root folder
Linux /
Windows C:/

Directory
tree

Sub folders,
subdirectories

Folder

Files

Folder cabinets (File System)

C: ← Primary Disk volume

C:\ ← Root directory

C:\windows\ ← Windows Folder

C:\Document and Settings\

D: Secondary Disk

E: DVD drive

F: USB disk

Filesystems: Fat32, NTFS, ext2, etc

Disk volumes

/dev/hda (linux disk #1)

- Partition #1
  - C: boot partition (primary dos/windows)
- Partition #2 (ext dos)
  - D: logical drive, FAT32
  - E: logical drive, NTFS
  - F: logical drive, CDFS
- Partition #3, Non dos
  - Bootable, Linux
- Partition #4, non dos
  - OS/2 or BSD

Hard disk partitions

Filenames

- Files have a name and extension
- Avoid non-ascii chars in filenames
- Extension indicate type of file

  eg. readme.txt, photo.jpg, pic.gif, song.mp3, movie.avi

  Extensions can be wrong, e.g. photo.mp3

File attributes

Permissions:

- Readable, writable, readonly, execute
- Which users or groups can access it?

Timestamps:

- Time of creation, modified, access date

Size: in bytes

Type:

- text, binary, symbolic link
- stream, seekable, pipe, socket, lock

Paths

- Filesystems contain folders and files
- Folders and files have names
- Charset: Ascii, German, Unicode
- Paths are locations of Folders and files.
- Backslash is the DOS path separator
- Avoid non-ascii chars in paths, so it is easier to type in commands.
- Eg. C:\home\john is better than "C:\document and settings\john will"

Paths

Paths can be

- Absolute, example:
  C:\home\john\cc
- Relative to current folder
  . dot refers to the current folder
  .. dot-dot refers to the parent folder
  ..\.. Refers to the grand-parent folder
  Example: ..\readme.txt

cmd keys

- Arrow-keys: command history editing
- F7 .. F8: command history
- TAB .. complete directory/filename
- Control-C (C-c) .. kill current command
- Control-Z (C-z) .. EOF (end of file).

Folder (directory) commands

C:\windows> cd ← Shows current dir

"C:\windows"

C:\windows> cd \ ← Change-dir to root

C:\> mkdir tmp ← Make dir C:\tmp

C:\> rmdir tmp ← Remove dir C:\tmp

Getting help

1. Google search
2. C:\> cd /?

Displays the name of or changes the current directory.

CHDIR [/D] [drive:][path]
CHDIR [..]
CD [/D] [drive:][path]
CD [..] .. Specifies that you want to change to the parent directory.

Type CD drive: to display the current directory in the specified drive.
Type CD without parameters to display the current drive and directory.

Use the /D switch to change current drive in addition to changing current directory for a drive.
...

Command syntax

prompt> command **[/switches]**
[arguments]

Command completion, press [TAB]
repeatedly till the right word appears

C:\> cd C:\do<TAB><TAB>

C:\> cd c:\document and settings\<TAB>

C:\> cd c:\document and settings\john

File operations

C:\> Copy oldfile newfile

1 file(s) copied.

C:\> Copy oldfile directory

C:\> Rename oldfile newfile

C:\> Move oldfile folder

C:\> Delete oldname

Find dirs (folders)

C:\Documents and Settings> dir /s /b /ad goo*

C:\Documents and Settings\b\Application Data\Google
C:\Documents and Settings\Default User\Application Data\Google
C:\Documents and Settings\Default User\Local Settings\Application Data\Google

Options to commands:

Dir command takes following options (switches):

/s .. search Sub-directories also
/b .. just print Bare names
/ad .. result Attribute must be Directory (we don't want files)

Arguments to commands:

After switches, we type arguments to the dir command,
here we want folder names starting with goo..

Find files

Find files name ‘Hosts’ in C:\windows

```
C:\windows> dir /s/b hosts
    C:\WINDOWS\I386\HOSTS
    C:\WINDOWS\system32\drivers\etc\hosts
```

Search with wildcards:

```
C:\WINDOWS>  dir/s/b *socket*.*
   C:\WINDOWS\I386\MFSOCKET.IN_
   C:\WINDOWS\inf\mfsocket.inf
   C:\WINDOWS\inf\mfsocket.PNF
```

Saving output of commands

Search all doc files in c: drive and save the result to list-docs.txt file

```
C:\> dir /s/b *.doc > list-docs.txt
```

Save standard error output of gcc to a file

```
C:\> gcc bigfact.c 2> error.txt
```

Wildcards

Copy all files in . with .c extension to C:\tmp

C:\> copy *.c c:\tmp\

List all algo c file files:

C:\> dir algo*.c

List all algo files in any folder:

C:\> dir /s/b algo*.*

cmd wildcards to match filenames

- * matches any chars (zero or more)
- ? matches exactly one character

Examples:

- a*.? matches algo.c, algo.h, aah.c aaaa.c a.c
- a*b.d matches ab.d, axb.d, aabb.d,…
- *.* matches anything
- ?.? matches a.b, x.y, etc.

Search files, Regular expressions

Using GNU grep to find all c files containing the regular expression ‘Random…Numbers’:

C:\> grep –Pins random.*numbers *.c

- -P regular expression is perl syntax
- -i Ignore case,e.g. RANDOMNumbeR
- -n Print line number of match
- -s Ignore errors while searching

Regular (regexp) expressions

- Used to match strings in PERL, Python, C, Java, Vim, Emacs (text editor).
- Regexp syntax:

case sensitive

.	any one char
^	beginning of line
$	end of line
\n	newline
tom | jerry	tom OR jerry
“a(b | c)d”	grouping with parenthesis: abd or acd.
[a-zA-Z@#]	any char: a to z, A to Z, @, #.
[^a-z]	any char except a-z

Search output of a command

- C:\> dir | grep –Pi “(notes|exam)”
- Search the output of dir command for Notes or Exam, ignore case (perl regular expression)

- C:\> dir /s/b . | grep –Pi “(algo.*notes|number.*the.*oo)”
- Matches files names like “Algorithm Notes” and “Number-theory is good”.

Cmd Environment

Every process has an environment, it is a

Env is a list of variable=value, string pairs

C:\> set

```
SystemDrive=C:
SystemRoot=C:\WINDOWS
TEMP=C:\temp
USER=john
```

C:\> echo %TEMP%

```
TEMP=C:\temp
```

Beware of hidden spaces in env variable, e.g.

c:\> set tmp=c:\tmp<space>

c:\> rm –rf %tmp%/* .. this will delete everything: rm –rf c:\tmp<space>/*

cmd aliases (shortcuts)

Make 'ls' mean the same as 'dir'

C:\> doskey ls=dir $*

C:\> ls desktop.ini …. dir desktop.ini

IO Streams

Every process is connected to IO-streams:

0. stdin, standard input (keyboard).
1. stdout, standard output (monitor).
2. stderr, standard error (monitor).

PATH

- How does cmd find the commands you type?

C:\> set PATH … Show the PATH

Path=C:\windows;C:\windows\system32;….

To Change the PATH

C:\> set path=C:\cygwin\bin;%PATH%

cmd batch files

```
C:\> more my.bat
@echo off
@rem this is a comment
echo Hello %USER%

C:\> my.bat
Hello john

C:\>
```

Bash shell

Bash shell

Default terminal shell (command interpreter) on Linux is /bin/bash

Windows users can download c:/cygwin/bin/bash with cygwin.

History:

sh (ATT unix) → ksh → bash (current)

Obselete: zsh, csh, tcsh (bad design)

Start bash in a terminal

- start > terminal

ls – list directory and files

bash$ ls –l will .. –l option for long details

cat (concatenate, print file)

cat filename .. prints the filename to stdout, which is the terminal screen, also called /dev/tty.

```
File  Edit  View  Bookmarks  Settings  Help
[root@localhost ~]# cat /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
/bin/dash
[root@localhost ~]#
```

Environment

echo $HOME

ps -p $$ .. process info

$$ .. Is the pid (process id) of this shell.

TTY .. is the controlling terminal of this bash

Unix filenames

- / is the root
- /dev/ .. are the devices, like keyboard, tty, harddisks.
- /bin .. are the programs
- /home/john .. user directory
- /etc .. system configuration (like registry)
- /usr .. user applications
- Symlinks, one link can point to another

Unix file system

Common terminal keys

- Control-Z .. suspend command
  - fg .. restart command in foreground
  - bg .. send command into background
- Control-C .. interrupt current command
- Control-\ .. Kill current command
- Control-D .. EOF to logout
- Control-S .. Stop screen output
- Control-Q .. continue screen output.

Readline (Command line editing) in bash

- C-a .. beginning of line
- C-e .. end of line
- C-r .. search history
- Up-arrow .. previous history command
- Down-arrow .. next history commands
- C-k .. delete to end of line

See google, same as Emacs editor keys, can remap keys in ~/.inputrc

Common Unix commands

- ls files .. list file or directory
- cat files .. print files to stdout
- man xyz .. show manual help page for xyz
- cp source target .. copy source to target
- mv source target .. move
- rm source .. remove source
- cd /usr/local .. change directory to
- pwd .. show present working dir
- grep regexp file .. search regexp in files
- more files .. show files page by page.

More Unix commands

- ps … show processes
- who … show who is logged on
- date .. show date
- cal .. show calendar
- stty .. terminal settings
- chmod .. change dir/file permissions
- vim files .. vi improved editor
- emacs files .. emacs editor

Network commands

- ping host .. check network connection to host
- tracert host .. trace route to host
- nslookup .. DNS name lookup
- mail .. read or send email
- ftp .. file transfer
- wget urls .. download urls
- telnet host .. login to host
- ssh host .. secure shell login to host
- finger user@host .. find out about user on host

Process and its IO

Saving output to a file

Count number of lines in /etc/shells and save it to x

```
$ wc –l /etc/shells  > x
$ cat x
  16 /etc/shells      .. number of lines in file
```

Save errors to a file (stderror is fd2):

```
$ gcc –Wall bigfact.c 2> errors.txt
$ more  errors.txt  ... show the file page by page
```

Reading input from a file

```
$ wc –l <  /etc/shells
  16 lines
```

Redirect input and output

```
$ wc –l < /etc/shells > x
```

Saving outputs

Save output, redirect stdout to a file.

```
$ wc /etc/shells > /tmp/y
$ cat /tmp/y
  16  16 186 /etc/shells
```

(means 16 lines,16 words, 186 chars in /etc/shells)

Save output and error messages of gcc, send stdout to file x, and also redirect stderr/2 to stdout/1.

```
$ gcc -Wall bigfac.c  >  x     2>&1
```

pipe, and io redirection

$ command1 | command2

Piping '|'

Pipe output of first cmd to next cmd, example

```
$ cat /etc/shells | wc
  16 16 186
```

Pipe example with 3 commands

Example: Count number of lines in file containing the string 'sh' or 'SH' ..

```
$ cat /etc/shells |
  grep -i sh |
  wc -l
2
```

Running cmd in background

```
$ wc /etc/shells > /tmp/x  &
  [1] 2804
```

... process number of background job.

```
$ ls
  [1]+ Done
```

... later background job is done

Quoting arguments

```
$ echo *          .. '*' is globbed into matches
home etc usr
$ echo "*"        .. prevent globbing of *.
*
$ echo \*         .. backslash quotes next char
*
```

Quoting Variables

```
$ echo $HOME
  /home/john
$ echo 22${HOME}99
  22/home/john99

$ echo ‘$HOME’
  $HOME
$ echo \$HOME
  $HOME
```

Bash aliases

Make 'dir' same as 'ls –al' command

```
$ alias dir='ls –al'
$ alias   date-is='date +%Y-%m-%d'
$ date-is
2013-04-13
```

Bash functions

```
$ function dates(){
    echo DATE_SECONDS=$(perl -e "print time")
    echo DATE_YESTERDAY=$(date --date="1 days ago" +%Y-%m-%d)
    echo DATE_TODAY=$(date --date="0 days ago" +%Y-%m-%d)
    echo DATE_TOMORROW=$(date --date="1 days" +%Y-%m-%d)
}
$ dates
DATE_SECONDS=1365864924
DATE_YESTERDAY=2013-04-12
DATE_TODAY=2013-04-13
DATE_TOMORROW=2013-04-14
```

bash scripting

```
$ cat script
  #!/bin/bash
  # my first comment in this file.
  echo “My first script, hello $USER”
$ chmod +x script
$ ./script
  My first script, hello john
$ bash –x –v script      .. To debug verbose
  My first script, hello john
```

Bash script commands, if then

```
$ cat myscript1.sh     # comment.
if [[ file1 –nt file2 ]] ;then
    echo “file1 is newer”
;elsif [[ 20 –gt 5 ]] ;then
    echo “20 is greater than 5”
;else
    true; # dummy stmt.
; fi
```

case stmt

```
$ cat myscript2.sh
case $# in
0) echo You typed no arguments ;;
1) echo You typed $1 ;;
2) echo You typed $1 and $2 ;;
*) echo You typed $* ;;
esac
```

for loop

```
$ for user in a b c ;do
      ls –l /home/$user
; done > list.txt
```

while loop

```
$ file=/tmp/x.log
$ while [[ ! –s $file ]] ; do
  echo waiting for $file to fill up
  sleep 1
  done
```

Perl power user

Fix spelling of 'thier' to 'their' in all c files

```
$ perl -p -i.bak -e 's/\bthier\b/their/g'  *.c
```

---

s/// is Substitute/search-regexp/replacement/

Options:

-p print .. print each line after substitute

-i.bak .. save original as file.bak

-e expr .. to execute perl expression on each line

Windows / Unix differences

	Dos/Windows	Unix/Linux/Bsd
File separator	****	**/**
Root	C:\	**/**
Line ending (Fix dos2unix, unix2dos)	\r\n	\n
Shell	cmd	bash
File case	dir == DIR	ls != LS
Syntax	Inconsistent	Perfect
variables	%USER%	$USER

Broken windows commands with same names as Unix commands:

- echo
- find
- date
- time
- for
- if
- mkdir
- link

Editing

Emacs, Vim, Notepad++

Debugging with

GDB

Compiling for gdb

On MS-Windows, install codeblocks or cygwin:

C:\> set path=c:\codeblocks\mingw\bin;%path%

Linux should have gdb pre-installed.

Must compile with gcc '-g' debug flag

and -Wall to see all warnings:

```
$ gcc –g   -Wall test1.c     -o test1.exe
$ g++ –g  -Wall test2.cpp -o test2.exe
```

Using gdb

```
$ gdb test1.exe
(gdb) help  # to see all the commands
(gdb) run [args to test1.exe]
Hello world
(gdb) kill    # if program is not done
(gdb) quit
$
```

Stack

- (gdb) bt # backtrace
  - #0 func2 (x=30) at test.c:5
  - #1 0xe6 in func1 (a=30) at test.c:10
  - #2 0x80 in main (argc=1, argv=0xbff) at test.c:19
  - #3 0xf23 in __libc_start_main () from /lib/libc.so.6

- (gdb) frame 2
  - #2 0x814 in main (argc=1, argv=0xf4) at test.c:19 ..

What’s on the stack

- (gdb) info frame
  - Stack level 2, frame at 0xbffffa8c:
  - eip = 0x8048414 in main (test.c:19);
  - saved eip 0x40037f5c
    - called by frame at 0xbffffac8,
    - caller of frame at 0xbffffa5c source language c.
  - Arglist at 0xbffffa8c, args: argc=1, argv=0xbffffaf4
  - Locals at 0xbffffa8c,
  - Previous frame's sp is 0x0
  - Saved registers: ebp at 0xbffffa8c, eip at 0xbffffa90
- (gdb) info locals
  - x = 30
  - s = 0x8048484 "Hello World!\n"
- (gdb) info args
  - argc = 1
  - argv = (char **) 0xbffffaf4

Breakpoints

- (gdb) break test.c:19 # break filename:linenumber
  - Breakpoint 2 at 0x80483f8: file test.c, line 19
- (gdb) break func1
  - Breakpoint 3 at 0x80483ca: file test.c, line 10
- (gdb) break TestClass::testFunc(int)
  - Breakpoint 1 at 0x80485b2: file cpptest.cpp, line 16.
- (gdb) tbreak main
  - Will stop once in main
- (gdb) info breakpoints
- (gdb) disable 2
- (gdb) enable 2
- (gdb) ignore 2 5
  - Will ignore next 5 crossings of breakpoint 2.

Running

- (gdb) run
- You Press <Control-C>
  - Program received signal SIGINT, Interrupt.
- (gdb) bt # show call stack
- (gdb) list # show near source code.
- (gdb) print x # show variable value

Stepping

- (gdb) call your_c_function(3)
- (gdb) next # go line by line of source code.
  - (gdb) **step** # go into function also.
  - (gdb) **finish** # step out of function
- (gdb) continue

Viewing data, x/format

- (gdb) x/s ptr # print var as a string.
- (gdb) x/4c ptr # as 4 chars.
- (gdb) x/t ptr # in binary bits
- (gdb) x/3x ptr # as 3 hex bytes
- (gdb) set var ptr = 0 # change var
- (gdb) info registers

Watching data

(gdb) watch x

Hardware watchpoint 4: x, will print message whenever x is changed.

(gdb) rwatch y # read

(gdb) awatch z # access.

Viewing asm code

```
(gdb) dis main      # disassemble exe

C:\> gcc –S file.c   # Generate asm file
C:\> cat file.S      # view it
 …
```

Other options

- C:\> gdb –tui .. START gdb with GUI
- Codeblocks debugger has a gdb command line.
- Emacs: M-x gdb (For linux and emacs)

Also see

- gdb – most systems.
- MS Windows:
  - Visual studio
  - kernel – windbg, softice
- Linux kernel - kgdb, kdb
- GUI – codeblock, ddd
- Valgrind, purify – Runtime memory errors
- Lint - Static bad style warnings

Using GPG

- Making your public key
- Signing
- SSL

Making your own public key

GPG : GNU Privacy Guard

MS-Windows cygwin has pgp

Linux should have gpg installed.

If not, download and install it.

Online help:

https://help.ubuntu.com/community/GnuPrivacyGuardHowto

http://www.gnupg.org/gph/en/manual.html

Using GnuPG to generate your keys

- Open a terminal and enter:
  $ gpg --gen-key
- Please select what kind of key you want:
- (1) RSA and RSA (default)
- (2) DSA and Elgamal
- (3) DSA (sign only)
- (4) RSA (sign only)
- Your selection? 1
  - RSA keys may be between 1024 and 4096 bits long.
- What keysize do you want? (2048)
  - Requested keysize is 2048 bits
- Please specify how long the key should be valid.
- 0 = key does not expire
- <n> = key expires in n days
- <n>w = key expires in n weeks
- <n>m = key expires in n months
- <n>y = key expires in n years
- Key is valid for? (0)
  - Key does not expire at all
- Is this correct? (y/N) y

  - You need a user ID to identify your key; the software constructs the user ID
- Real name: Mohsin Ahmed
- Email address: moshahmed@gmail.com
- Comment: NITK demo
- You selected this USER-ID:
  - "Mohsin Ahmed (NITK demo) <moshahmed@gmail.com>"
- Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
  - You need a Passphrase to protect your secret key.
- Passphrase: **** (e.g. a short sentence)
- Repeat Passphrase: ****
  - Forgetting your passphrase will result in your key being useless. Carefully memorize your passphrase.

Key pair is created

- ..+++++
- .....+++++
- gpg: /cygdrive/c/mosh/.gnupg/trustdb.gpg: trustdb created
- gpg: key B30E2394 marked as ultimately trusted
- public and secret key created and signed.

- gpg: checking the trustdb
- gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
- gpg: depth: 0 valid: 1 signed: 0 trust: 0-, 0q, 0n, 0m, 0f, 1u
- pub 2048R/B30E2394 2013-02-09 (your public key)
- Key fingerprint = FB38 60F1 DAD5 0F64 A8E4 465D 249B A4E8 B30E 2394
- uid Mohsin Ahmed (NITK demo) <moshahmed@gmail.com>
- sub 2048R/3B0695DE 2013-02-09

Upload your public key

```
$ gpg --keyserver pgp.mit.edu
        --send-key B30E2394
```

Print your key

$ gpg -v --fingerprint B30E2394

- pub 2048R/B30E2394 2013-02-09
- Key fingerprint = FB38 60F1 DAD5 0F64 A8E4 465D 249B A4E8 B30E 2394
- uid Mohsin Ahmed (NITK demo) moshahmed@gmail.com

- Print the fingerprint, and give the printout to your friends (who know you), ask them to sign it and certify it as your key.
- You will do the same for people you know well.

Signing someone’s key

- Download their key (e.g. 00AA11BB) or from their email (verify their fingerprint).

$ gpg --keyserver pgp.mit.edu --recv-keys 00AA11BB

Check their fingerprint:

$ gpg --fingerprint 00AA11BB

Sign it:

$ gpg --sign-key 00AA11BB

Send them their signed certificate

```
$ gpg --armor --output cert1.txt --export
  00AA11BB
$ mail friend < cert1.txt
```

Update your signed keys

When you receive your cert2.txt signed from a friend:

```
$ gpg --import cert2.txt
```

Update your key in the server:

```
$ gpg --keyserver pgp.mit.edu --send-key B30E2394
```

Send someone a locked file

```
$ gpg --output locked.txt --encrypt
  --recipient friend@gmail.com poem.txt
```

Now mail the file ‘locked.txt’ to friend.

Friend can read the attached file with the command:

```
$ gpg --output poem.txt --decrypt locked.txt
```

Sign a cheque

Create cheque.txt and then sign it:

$ gpg --output check.sig **--clearsign** check.txt

Mail the signed check.sig to your friend.

Friend can verify your signature:

$ gpg --verify check.sig

Using Code::Blocks IDE

(Integrated Development Environment)

Install Codeblocks

- Codeblocks is the free IDE for C/C++ programming, it comes with the gcc and g++ compilers and libraries.
- It works well on Windows. Windows gcc is called mingw gcc
- Download and Install Codeblocks (with gcc compiler) on your computer, e.g. in **`c:/tools/codeblocks`** (no spaces in path).

Run Codeblock

- Run Codeblocks, then click on the menu:
- File > New > Project > Console Application > Select C or C++ application >
- Pick a new folder to create your application, give the directory a good name (without spaces in path), so you can find it again.
- Click next to finish the wizard

Create project

Console application

Type of program

Directory to save files

Compiler to use

Workspace

Now you can start programming, click on the left pane on

- Workspace >
- project name >
- Sources >
- main.cpp

Type the *hello world* program, main.c

```
1. #include <iostream>
2. using namespace std;
3. int main( ) {
4.   cout << "Hello world!" << endl;
5.   return 0;
6. }
```

Settings>Compiler: enable –g for debug symbols

Settings>Compiler: enable –Wall to see all warnings during compiling.

Settings > Debugger

Build and run

- Now we can run the hello-world console application:

Compiler output

- You will see the compiler output in the build log pane on the bottom right window.
- If there are no errors, the program will run and a *console* window will pop up, with the output “Hello world!”.

Run the program

Debugging
with
Codeblocks

Why debug?

- Programming without a debugger is like driving a car without knowing how to use brakes.
- Whenever there is a problem, you just break (brake) and see what is wrong in the source code, along with the variable values. It is much easier than guessing what is wrong by staring at the code.

Before you debug

1. Compile your code with gcc –Wall and fix all warnings.
2. Add assertions to the code to check your assumptions. E.g.

```
#include <assert.h>
…
g = gcd(3,7);
assert(g == 1);
…
```

3. Use the latest Codeblocks (with gcc).
4. Don't use obsolete Turbo-C or Dev-Cpp.

Example C++ program

- Type the following program into codeblock main.cpp

```
1. #include <iostream>
2. using namespace std;
3. int main() {
4.     for(int i=0;i<5;i++)
5.       cout << i*i;
6.     return 0;
7. }
```

- Click next to the number ‘5’ to create a breakpoint (red dot in the image).
- Then from the Build menu, select build to compile the program and ensure there are no errors.
- If there is an error, you will see a red mark and the error message will be shown in the build-log.

Build main.cpp

Watch variables

- Click on ‘debug > edit watches’

Watch 'i'

- In ‘Edit debugger watches’,
- click on ‘Add’ and
- in the ‘add watch window’
- type ‘i’ to see the
- variable i’s value
- during debugging:

Start debugging:

Stopping at a breakpoint

- Now the program has stopped at the breakpoint, in the image, the red-dot has a yellow-arrow, showing that it is the current statement:

Go line by line

- To run to ‘next line’ you can press F7 or click on the icon shown on the image:

Step until condition

- Press Control-F7, few times till ‘i=4’ appears in the watch window.
- Then press F7 to till ‘i=5’ appears in the watch window.
- Now press Alt-F7 to run the next instruction.
- You will see the ‘Disassembly’ window.

Stop debugging

- Also look at the console window of the running program. To finish, click on debug > stop debugging.

Advanced breapoints

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  void g(int i, int * f){
4.      while(i++ <= 10)
5.          f[-i*2+1]++;
6.  }
7.  int main(){
8.      int i=2, f[21];
9.      g(i, f);
10.     return 0;
11. }
```

Debug > Run > Crash

Conditional breakpoint

Add expression to break when array index out of bounds.
Run program until breakpoint is triggered.

select expression inside f[…] >
right click > watch > view index

Data breakpoint

Select f with mouse
- right click
- add data-breakpoint
- Edit data bp to ‘break on write’ of f[21]

GDB commands in codeblock

If you enable gdb log window, you can see the gdb command generated by your GUI actions:

```
[debug]>  break "C:/src/codeblock/bt/main.c:5"
[debug]Breakpoint 3 at 0x401339: file C:\src\codeblock\bt\main.c, line
    5.
[debug]>>>>>> cb_gdb:
[debug]> condition 3 -i*2+1 < 0 || -i*2+1 >= 21
[debug]>>>>>> cb_gdb:
[debug]> output &f[21]
[debug](int *) 0x22ff4c>>>>>>cb_gdb:
[debug]>  awatch *0x22ff4c
[debug]Hardware access (read/write) watchpoint 4: *0x22ff4c
```

Debugging tips

1. If your program crashes on some input, make a testcase: **hardcode** the data causing the problem, so you can debug it quickly many times (without having to type in the inputs during each debug sessions).
2. Use **binary search** to locate the cause of problems.
3. **Step over** hairy code, and watch the variables are as you expect them to be.
4. Make a **testcase**, learn how to write unittests. Run the unittest everytime you make changes to the code.
5. Use **CVS** or **GIT** to keep older versions of your program, so that you can revert to an older version of the code, if the new version doesn't work. Then use diff to find out what changes are causing the failure.
6. For really hard inconsistent problems, use **valgrind, purify**, and a different compiler.
7. Make notes, add comments explaining the problems.
8. Convert comments to code or assertions.
9. Search **Google** to see if others had the same problem and how they fixed it.

Tips: Assertions and comments

```
int  g(int x) {
  int t;
  // 1. t is total sum
  // 2. assume x > 0.
 ...
 // 3.
 if (t > 0)
  return 1; // true
else
  return 0; // false


}
```

```
int g(int x) {
  int total_sum;
  assert(x>0);
  ...
  return total_sum > 0;



}
```

1. Use descriptive variables names
2. Use assert to check assumptions
3. Keep it short

Using *Microsoft* MSDev VC6 IDE

(Integrated Developer Environment)

Using VC6

- Visual Studio from Microsoft is an IDE for C/C++ programming, msdev comes with the Microsoft C and C++ compilers and libraries and works on Windows only.
- *(Other option is to download Codeblocks with GCC (works on Linux and Windows). GCC supports C99 features, while VC++ is more useful for developing complex Windows applications).*
- This assumes you have installed msdev vc 6.0 on your computer, with vc6 sp2 (service-pack 2 is required to compile newer C++).
- Run msdev, then click on the msdev menu:

  File > New > Win32 Console Application (see next slide)
- Pick a new folder (c:\vc6\demo\hello) to create the application, call it hello.
- Click 'OK' to continue the wizard.

Creating a new application

Created an Empty project

- Select “An empty project” and
- click on “Finish”.

Created a new project

- Click ‘OK’

- Now click on: File > New > C++ source file > Type filename: hello.cpp

Building the application

- Type the following program as hello.cpp
- Click on: Build > Build hello.exe or [F7].
- In the build pane at the bottom, you will see “compiling hello.exe”.
- If there are any typos correct them
- Till you get zero errors and warnings.

Build hello.exe

Running the application

- Now we can run the hello-world console application: Build > Execute (C-F5)

Building until no errors

- You will see the compiler output in the build log pane on the bottom right window.
- If there are no errors, the program will run and a console window will pop up, with the output “hello world” (with no extra spaces around it).

Running the application

- Build > Execute "Hello.exe"

Debugging with MS VC6

Debug main.cpp

Type the following program into vc6, as main.cpp

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4    for(int i=0;i<5;i++)
5      cout << i*i;
6    return 0;
7 }
```

MSDev IDE

debug - Microsoft Visual C++ - [debug.cpp]
File Edit View Insert Project Build Tools Window Help
(Globals)
(All global members)
main
Workspace 'debug': 1 project(s)
debug files
Source Files
main.cpp
Header Files
Resource Files
ClassView
FileView
#include <iostream>
using namespace std;
int main() {
for(int i=0;i<5;i++)
cout << i*i;
return 0;
}
Deleting intermediate files and output files for project 'debug - Win32 Debug'.
--------Configuration: debug - Win32 Debug--------
Compiling...
main.cpp
Linking...
debug.exe - 0 error(s), 0 warning(s)

Creating a Breakpoint

- To create a breakpoint on line 4, click on "Edit > Breakpoints…",
- And pick 'line 4' from the breakpoint dialog.
- And click on 'Build > Build main.cpp'
- And make sure it compiles with no errors.

Accessing the breakpoints

Setting a BP

in.cpp]
ild Tools Window Help
main
#include <iostream>
using namespace std;
int main() {
for(int i=0;i<5;i++)
cout << i*i;
return 0;
}
Breakpoints
Location
Data
Messages
OK
Cancel
Break at:
{,c:\vc6\demo\debug\main.cpp,} .4
Line 4
Advanced...
Condition...
Click the Condition button if you want to set conditional parameters for your breakpoint.
Breakpoints:
at '{,main.cpp,} .4'
Remove
Remove All

Go, Start debugging

- Start debug by clicking on:
- Build > Start debug > Go (or press F5).

Step by Step debugging

- It will stop at the breakpoint, click on: “Debug > Step over” to go to next line:

Watching variables

- Below you see the value of i is ‘0’ in the ‘Auto’ pane (local variables).
- You can add any variable, e.g. ‘i’ to the watch pane, to view it’s value during debugging.
- You can also click on the ‘context’ dropdown menu to change the stackframe:

Changing the Context (Stack frame, activation record)

The debug windows

To view all the debug information, click on:

View > Debug windows > call-stack or disassembly:

Dis-assembly view of source code during debugging

Stack during recursion

Now type the factorial program to see the call stack during recursion.

```
1.  #include <iostream>
2.  using namespace std;
3.  int factorial(int i){
4.    if (i<2) return 1;
5.    else return i * factorial(i-1);
6.  }
7.  int main() {
8.    cout << factorial(4);
9.  }
```

Recursion

- Build and put a breakpoint on line 4 as shown in the image.
- Then start debugging.
- Display of ‘call stack’, as you step through the program,
- you will see the recursion go deeper into the factorial function until i=1 is seen in the call stack.

Recursive call stack

Computing 4!

- Now step ahead couple of times to see the call stack unwind, and the main function will print ‘24’ (4! is 1*2*3*4).

Computed 4! = 24.

Hardware breakpoints

Visual C++ only supports write-only data breakpoints,
example use to find out "*Who is corrupting our data structure?*"
Consider this program, why is it printing k=202, i=301 (not k=201)?

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. //
4. void update(int mr[]) {
5.     int i;
6.     for(i=0;i<=3;i++)
7.         mr[i]++;
8. }
9. int main( ) {
10.    int k=201, ar[3] = {10,20,30}, i=301;
11.    update(ar);
12.    printf("k=%d, i=%d",k,i);
13.    return 0;
14.}
```

Debug registers

- The Intel x86 CPUs have some special registers which are intended for debugging use only. By storing special values into these registers, a program can ask the CPU to execute an INT 1 (interrupt 1) instruction immediately whenever a specified memory location is read from or written to.
- **`INT 1`** also happens to be the interrupt that's executed by the CPU after a debugger asks the CPU to single-step one assembly line of the program.

Watching memory for change

- Determine what address you want to watch
- Open the new **breakpoint** dialog, and switch to the data tab
- Enter the address that you want to watch in the variable column
- The context is ignored. You can clear this if you want.
- Set the item count. If you are using an address, this should be the number of bytes (example: 4 for a integer type)

Stop after k is allocated, find its address,
and put a Data change breakpoint on 4 bytes of k

Breakpoint hit in update(), when k is changed

At the bp i=3 and mr[3]++=k++

hwbp - Microsoft Visual C++ [break] - [hw-bp.c]

File Edit View Insert Project Debug Tools Window Help

(Globals) | (All global members) | update

```
#include <stdio.h>
#include <stdlib.h>

void update(int mr[]) {
  int i;
  for(i=0;i<=3;i++)
    mr[i]++;
}
int main( ) {
  int k=201, ar[3] = {10,20,30}, i=301;
  update(ar);
  printf("k=%d, i=%d",k,i);
  return 0;
}
```

Context: update(int *)

Name	Value
i	3
mr[i]	202

Auto | Locals | this

Name	Value
k	CXX0017: Error: symbol "k"
&k	CXX0017: Error: symbol "k"

Watch1 | Watch2 | Watch3 | Watch4

Ln 7, Col 1 REC COL OVR READ

View the disassembly and call stack

Right click to see the source

Conditional Breakpoint

- We want to stop in the loop, only when i=3

Hardcoded Breakpoint: __asm int 3

VC Command line setup

After Installing VC6 in c:\vc6,

Start > Run > Cmd

```
C:\> c:\vc6\VC98\bin\VCVARS32.BAT
set VSCommonDir=C:\vc6\common
set MSDevDir=C:\vc6\common\msdev98
set MSVCDir=C:\vc6\VC98
set PATH=%MSDevDir%\BIN;%MSVCDir%\BIN;
                 %VSCommonDir%\TOOLS\WINNT;
                 %VSCommonDir%\TOOLS;
                 %PATH%
set INCLUDE=%MSVCDir%\ATL\INCLUDE;
                 %MSVCDir%\INCLUDE;
                 %MSVCDir%\MFC\INCLUDE;
                 %INCLUDE%
set LIB=%MSVCDir%\LIB;
               %MSVCDir%\MFC\LIB;
                %LIB%
```

VC Command line compiler

```
C:\> cl /W4 /ZI /GZ  hello.c    .. To compile debug hello.exe
C:\> cl /W4 hello.c | gvim –q - .. Quick-Fix errors with vim

----------------------------------------------
C:\> cl /?                      .. help with c compiler
Useful Flags:
       /GX  .. enable C++ Exception Handler
       /W4  .. all warnings
       /ZI  .. incremental debugger
       /GZ  .. runtime debug checks

C:\> ntsd    .. command line debugger (if gcc doesn't work)
```

C Program Execution

- Stack
- Heap memory
- Operation system

Compiler

- User edits hello.c
- Preprocessing
- Compiling hello.c to hello.s
- Assembling hello.s to hello.o
- Linking hello.o + libs to hello.exe on disk
- Loader hello.exe + dll or so into RAM
- Running on CPU

Compiler

hello.c → hello.exe

- Preprocessing
- Compiling
- Assembling
- Linking
- Loader
- Running

GCC commands

- gcc -E file.c # run only preprocessor
- gcc –S file.c # creates asm file.s
- gcc –c file.c # compiles to file.o
- readelf –a file.o # (linux)
- dumpbin /all file.o # (windows)

Memory contents

- Stack (local variables, return address)
- Heap (globals, malloc)
- Data (static)
- Text (exe,dll,so)
- Kernel

Memory

- stack
- dll libraries
- global data
- functions

Stack frames

- Function `call` creates a new *frame* on the stack.
- `return` deletes the current frame, and we return to previous frame.

What's in a frame on the stack?

- Local variables
- Parameters
- Return address

push 'abcd' on the stack

- Push 'ABCD'
- Register word size is 4 bytes

Function call example

```
#include <stdio.h>

int func_A(int x, int y )
{
  int s, t; // automatic variables
  return 3;
}

int main(int argc,
    char *argv[] )
{
  int i, k; // automatic variables
  func_A( 1, 2);
  return 0;
}
```

Function call

Stack during nested function calls

```
#include <stdio.h>
int c( ){ return 'c'; }
int b( ){ return 'b'; }
int a( ){ b( ); c( ); return 'a'; }
int main( ){ a( ); return 0; }
```

Calling a cdecl function

- TestFunc(1, 2, 'A'); // call

```
0x08048388 <main+28>:   movb   $0x41, 0xffffffff(%ebp)        ; prepare the byte of 'A'
0x0804838c <main+32>:   movsbl 0xffffffff(%ebp), %eax         ; put into eax
0x08048390 <main+36>:   push   %eax                           ; push the third parameter, 'A' prepared
                                                              ; in eax onto the stack, [ebp+16]
0x08048391 <main+37>:   push   $0x2                           ; push the second parameter, 2 onto
                                                              ; the stack, [ebp+12]
0x08048393 <main+39>:   push   $0x1                           ; push the first parameter, 1 onto
                                                              ; the stack, [ebp+8]
```

Stack before the push

Params on the stack in reverse order

Calling TestFunc(…)

```
0x08048395 <main+41> :          ; assembler comment
     call 0x8048334 <TestFunc>  ; function call.
    ; Push the return address [0x0804839a]
    ; onto the stack, [ebp+4]
```

Before

Ready to call

From http://www.tenouk.com/Bufferoverflowc/Bufferoverflow3.html

Entering TestFunc(…)

```
0x08048334 <TestFunc+0>:    push   %ebp        ;push the previous stack frame
                            ;pointer onto the stack, [ebp+0]
0x08048335 <TestFunc+1>:    mov    %esp, %ebp   ;copy the ebp into esp, now the ebp and esp
                            ;are pointing at the same address,
                            ;creating new stack frame [ebp+0]
0x08048337 <TestFunc+3>:    push   %edi        ;save/push edi register, [ebp-4]
0x08048338 <TestFunc+4>:    push   %esi        ;save/push esi register, [ebp-8]
0x08048339 <TestFunc+5>:    sub    $0x20, %esp  ;subtract esp by 32 bytes for local
                            ;variable and buffer if any, go to [ebp-40]
```

Pushing the EBP onto the stack, saving the previous stack frame.

Exercises

1. Debug this file abc.c in CodeBlocks:

```
#include <stdio.h>
int c( ){ return 'c'; }
int b( ){ return 'b'; }
int a( ){ b( ); c( ); return 'a'; }
int main( ){ a( ); return 0; }
```

2. Put breakpoints on all the functions
3. Run it step by step, and watch the stack.
4. Generate assembly using gcc
5. Optionally use ms-vc6 to do the same.

Solution: gcc assembly file

```
$ gcc –S abc.c
$ cat abc.s
```

```
_a:
  pushl %ebp
  movl  %esp, %ebp
  call  _b
  call  _c
  movl  $97, %eax  ; return 'a'
  popl  %ebp
  ret
```

```
. _main:
 pushl %ebp
 movl  %esp, %ebp
 andl  $-16, %esp
 call  ___main
 call  _a
 movl  $0, %eax
 movl  %ebp, %esp
 popl  %ebp
 ret
```

Solution: vc6 assembly file

```
c:\tmp> cl /FAs abc.c
c:\tmp> cat abc.asm
; line 4 : int a( ){ b( ); c( ); return 'a'; }
        push    ebp
        mov     ebp, esp
        call    _b
        call    _c
        mov     eax, 97; 00000061H
        pop     ebp
        ret     0
-----------------------------------------------------------------
c:\tmp> cl /Fc abc.c
c:\tmp> cat abc.cod
; line 4    : int a( ){ b( ); c( ); return 'a'; }
  00014 55              push    ebp
  00015 8b ec           mov     ebp, esp
  00017 e8 00 00 00 00  call    _b
  0001c e8 00 00 00 00  call    _c
  00021 b8 61 00 00 00  mov     eax, 97; 00000061H
  00026 5d              pop     ebp
  00027 c3              ret     0
```

Stack of abc.c in Codeblocks

MAPLE mathematical software

Maple

Symbolic and Numerical Mathematics

C:\> start C:\tools\MAPLEV4\BIN.WIN\WMAPLE32.exe

CRT: Chinese Remainder Theorem

Solve for x, given these modular equations:

- $x \equiv 2 \pmod 3$
- $x \equiv 3 \pmod 5$
- $x \equiv 2 \pmod 7$

```
C:\> c:\tools\maplev4\bin.win\cmaple.exe
>> ??                                    # General help
>> ??chinese                             # Search for help on CRT
  Calling Sequence:   chrem(u, m)
  Parameters: u - the list of evaluations [u0, u1,..., un]
              m - the list of moduli      [m0,m1,...,mn]
> chrem( [2,3,2], [3,5,7] );
  23
> quit;
C:\>
```

Powermod (used in RSA)

```
# Ordinary arithmetic power and then mod
> 7**123456 mod 101;
      16
# Powermod is faster and handles larger input
> 7 &^ 1234567891 mod 101;
      98
```

Graphing

```
> f := x -> 10*x^5 - 30*x +10;
> plot(f,-3..3);
```

. Stmts end in semi-colon

. Assignment is "colon equal"

Numerical Computation

```
> Pi*3^2
  9 Pi        # Maple keeps it as symbols.
> evalf(Pi*3^2,30);
28.274333882308139146163790449 5
```

```
> (3+2*I)*(2-I)/(3-I);   # Complex Numbers
                    23   11
                    -- + -- I
                    10   10
```

```
> (3+2*I)*(2-I)/(3-I);
```

$$\frac{23}{10}+\frac{11}{10}I$$

Quadratic Equations

> **solve(3*x^2+2*x +1,x);**

-1/3+1/3*I*2^(1/2), -1/3-1/3*I*2^(1/2)

```
> solve(3*x^2+2*x +1 ,x);
```

$$-\frac{1}{3}+\frac{1}{3}I\sqrt{2}, -\frac{1}{3}-\frac{1}{3}I\sqrt{2}$$

Java

Java programming

- Install java in easy-to-type toplevel directory (name without spaces), e.g.

```
C:\java\jdk16
C:\java\jre16
```

To compile and run a java file:

C:\> javac BinarySearch.java

C:\> java –cp . BinarySearch < input.txt

command	arguments	purpose
javac	.java file name	compile Java program
java	.class file name (no extension) and command-line arguments	run Java program

Java programming reference

- Textbook "CoreJava", Vol 1 and 2, by Horstmann and Cornell, http://www.horstmann.com/corejava.html

Anatomy of a command

Anatomy of a Java program and its invocation from the command line

BinarySearch.java
Java Class Example

```
import java.util.Arrays;

public class BinarySearch
{
   public static int rank(int key, int[] a)
   {
      int lo = 0;
      int hi = a.length - 1;
      while (lo <= hi)
      {
         int mid = lo + (hi - lo) / 2;
         if      (key < a[mid]) hi = mid - 1;
         else if (key > a[mid]) lo = mid + 1;
         else                   return mid;
      }
      return -1;
   }

   public static void main(String[] args)
   {
      int[] whitelist = In.readInts(args[0]);

      Arrays.sort(whitelist);

      while (!StdIn.isEmpty())
      {
         int key = StdIn.readInt();
         if (rank(key, whitelist) == -1)
            StdOut.println(key);
      }
   }

}
```

import a Java library (see page 27)

code must be in file `BinarySearch.java` *(see page 26)*

parameter variables

static method (see page 22)

return type

parameter type

initializing declaration statement (see page 16)

expression (see page 11)

loop statement (see page 15)

return statement

system calls `main()`

unit test client (see page 26)

no return value; just side effects (see page 24)

call a method in a Java library (see page 27)

call a method in our standard library; need to download code (see page 27)

conditional statement (see page 15)

call a local method (see page 27)

system passes argument value `"whitelist.txt"` *to* `main()`

Java functions syntax

Anatomy of a static method

LaTeX Typesetting Software

LaTeX

Latex

- **LaTeX** is a document preparation system for high-quality typesetting.
- It is most often used for medium-to-large technical or scientific documents but it can be used for almost any form of publishing.
- **LaTeX** is *not* a word processor!
- **LaTeX** encourages authors *not* to worry too much about the appearance of their documents but to concentrate on getting the right content.

Installation

- Already installed on Linux
- Windows: install Miktex in **`c:\miktex`**

For LaTeX commands:

- *search google "latex tex tutorial"*

Or start here:

- *http://www.tug.org/begin.html*

Sample Document

```
c:\> cat test.tex

    \documentclass{article}
    \title{Algorithm assignment}
    \author{Mosh Ahmed}
    \date{29 February 2013}
    \begin{document}
      \maketitle
      Hello world!
    \end{document}
```

Compiling file.tex to file.dvi

```
C:\> latex test.tex
...
Output written on test.dvi (1 page, 424 bytes).
Transcript written on test.log.
```

View the dvi file

C:\> yap test.dvi .. View the output

Printing dvi file

```
C:\> dvipdfm test.dvi          # dvi to pdf
  test.dvi -> test.pdf
```

Greek letters and Symbols

- Search google for Latex manual in pdf.

α	\alpha	θ	\theta	o	o	τ	\tau
β	\beta	ϑ	\vartheta	π	\pi	υ	\upsilon
γ	\gamma	ι	\iota	ϖ	\varpi	ϕ	\phi
δ	\delta	κ	\kappa	ρ	\rho	φ	\varphi
ϵ	\epsilon	λ	\lambda	ϱ	\varrho	χ	\chi
ε	\varepsilon	μ	\mu	σ	\sigma	ψ	\psi
ζ	\zeta	ν	\nu	ς	\varsigma	ω	\omega
η	\eta	ξ	\xi				
Γ	\Gamma	Λ	\Lambda	Σ	\Sigma	Ψ	\Psi
Δ	\Delta	Ξ	\Xi	Υ	\Upsilon	Ω	\Omega
Θ	\Theta	Π	\Pi	Φ	\Phi		

Arrows

$\geq$	\geq	$\gg$	\gg	$\leq$	\leq	$\ll$	\ll	$\neq$	\neq

$\Downarrow$	\Downarrow	$\longleftarrow$	\longleftarrow	$\nwarrow$	\nwarrow
$\downarrow$	\downarrow	$\Longleftarrow$	\Longleftarrow	$\Rightarrow$	\Rightarrow
$\hookleftarrow$	\hookleftarrow	$\longleftrightarrow$	\longleftrightarrow	$\rightarrow$	\rightarrow
$\hookrightarrow$	\hookrightarrow	$\Longleftrightarrow$	\Longleftrightarrow	$\searrow$	\searrow
$\rightsquigarrow$	\leadsto*	$\longmapsto$	\longmapsto	$\swarrow$	\swarrow
$\leftarrow$	\leftarrow	$\Longrightarrow$	\Longrightarrow	$\uparrow$	\uparrow
$\Leftarrow$	\Leftarrow	$\longrightarrow$	\longrightarrow	$\Uparrow$	\Uparrow
$\Leftrightarrow$	\Leftrightarrow	$\mapsto$	\mapsto	$\updownarrow$	\updownarrow
$\leftrightarrow$	\leftrightarrow	$\nearrow$	\nearrow†	$\Updownarrow$	\Updownarrow

\arccos	\cos	\csc	\exp	\ker	\limsup	\min	\sinh
\arcsin	\cosh	\deg	\gcd	\lg	\ln	\Pr	\sup
\arctan	\cot	\det	\hom	\lim	\log	\sec	\tan
\arg	\coth	\dim	\inf	\liminf	\max	\sin	\tanh

π	\pi	ρ	\rho
ϖ	\varpi	ϱ	\varrho
ϖ	\varvarpi	ϱ	\varvarrho

Vectors

$\widetilde{abc}$	\widetilde{abc}*	$\widehat{abc}$	\widehat{abc}*
$\overleftarrow{abc}$	\overleftarrow{abc}†	$\overrightarrow{abc}$	\overrightarrow{abc}†
$\overline{abc}$	\overline{abc}	$\underline{abc}$	\underline{abc}
$\overbrace{abc}$	\overbrace{abc}	$\underbrace{abc}$	\underbrace{abc}
$\sqrt{abc}$	\sqrt{abc}‡		

$\aleph$	\aleph	$\diamond$	\Diamond*	∞	\infty	$'$	\prime
$\angle$	\angle	$\diamondsuit$	\diamondsuit	$\mho$	\mho*	$\sharp$	\sharp
$\backslash$	\backslash	$\emptyset$	\emptyset‡	∇	\nabla	$\spadesuit$	\spadesuit
$\square$	\Box*,†	$\flat$	\flat	$\natural$	\natural	$\surd$	\surd
$\clubsuit$	\clubsuit	$\heartsuit$	\heartsuit	$\neg$	\neg	$\triangle$	\triangle

Math

TABLE 67: Binary Relations

$\approx$	\approx	$\equiv$	\equiv	$\perp$	\perp	$\smile$	\smile
$\asymp$	\asymp	$\frown$	\frown	$\prec$	\prec	$\succ$	\succ
$\bowtie$	\bowtie	$\bowtie$	\Join*	$\preceq$	\preceq	$\succeq$	\succeq
$\cong$	\cong	$\mid$	\mid†	$\propto$	\propto	$\vdash$	\vdash
$\dashv$	\dashv	$\models$	\models	$\sim$	\sim		
$\doteq$	\doteq	$\parallel$	\parallel	$\simeq$	\simeq		

Latex Math Examples

```
 % Example 1
\begin{equation*}R = \frac{
\displaystyle{\sum_{i=1}^n (x_i-
   \bar{x})(y_i-
\bar{y})}}{\displaystyle{\left[
\sum_{i=1}^n(x_i-\bar{x})^2
\sum_{i=1}^n(y_i-
   \bar{y})^2\right]^{1/2}}}
\end{equation*}
% Example 2
$$
c = \sqrt{ a^2 + b^2 }
\int_{-\infty}^{\infty} \frac{1}{x}
   \, dx f(x) = \sum_{n =
   0}^{\infty} \alpha_n x^n
x_{1,2}=\frac{b\pm\sqrt{\color{Red}
   b^2-4ac}}{2a}
\hat a  \bar b  \vec c  x'  \dot{x}
   \ddot{x}
$$
```

$$R = \frac{\displaystyle{\sum_{i=1}^n (x_i-\bar{x})(y_i-\bar{y})}}{\displaystyle{\left[\sum_{i=1}^n(x_i-\bar{x})^2 \sum_{i=1}^n(y_i-\bar{y})^2\right]^{1/2}}}$$

$$c = \sqrt{ a^2 + b^2 }$$

$$\int_{-\infty}^{\infty} \frac{1}{x} \, dx$$

$$f(x) = \sum_{n=0}^{\infty} \alpha_n x^n$$

$$x_{1,2}=\frac{b\pm\sqrt{\color{Red} b^2-4ac}}{2a}$$

$$\hat a \ \bar b \ \vec c \ x' \ \dot{x} \ \ddot{x}$$

Python

- Python is a scripting language with lots of libraries
- Classes and objects
- Indent to arrange code

Python

- Useful for small calculations, scripts
- Install python33 / python26 / numpy superpack
- Usage:

```
C:\> C:\python33\python.exe
>>> 2+2
4
>>> ^Z                    # (^D on unix) EOF to exit
C:\>                      # back to DOS prompt
```

Python calculator

```
C:\> c:\python33\python.exe
>>> import math
>>> dir(math)              # see what's in math
>>> math.pi                # math object has attribute pi with value=...
3.141592653589793
>>> math.log2(math.pi)     # math object has member function log2
1.6514961294723187
------
>>> from math import *     # Make all math functions are global
>>> sin(pi/4)
0.70710678118654746
>>> log2(pi)
1.6514961294723187
```

Functions, quicksort

```
>>> def quicksort(arr):
…       “”””quicksort in python3”””
...       if len(arr) <= 1: return arr
...       pivot = arr[0]
...       return quicksort([x for x in arr[1:] if x < pivot]) + [pivot]
     + \
...                 quicksort([x for x in arr[1:] if x >= pivot])
...
>>> print(quicksort([4,5,1,3,2]))
[1, 2, 3, 4, 5]
```

- '**...**' means python is prompting for more input

Don’t use tabs,
indent with 4 spaces

Linear Algebra in Python

- Install Python
- Install NumPy superpack python library

Python Linear Algebra calculator

```
C:\> python
>>> import numpy as np
>>> x = np.array( ((2,3), (3, 5)) )
>>> y = np.matrix( ((1,2), (5, -1)) )
>>> np.dot(x,y)
matrix([[17,  1],  [28,  1]])
```

Python solver and graphics

Solve the system of equations

```
3 * x + 1 * y = 9
1 * x + 2 * y = 8
```

C:\> python

- >>> import numpy as np
- >>> a = np.array([[3,1], [1,2]])
- >>> b = np.array([9,8])
- >>> x = np.linalg.solve(a, b)
- >>> x
- array([2., 3.])
- >>> (np.dot(a, x) == b).all()
- True
- import matplotlib.pyplot as plt
- >>> >>> plt.plot(x, 5*x + 2, 'r')
- >>> plt.show()

Matrix inverse

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.mat('[1 3 5; 2 5 1; 2 3 8]')
>>> linalg.inv(A)
array([[-1.48,  0.36,  0.88],
       [ 0.56,  0.08, -0.36],
       [ 0.16, -0.12,  0.04]])
>>> linalg.det(A)
-25.
```

When Matrix A = $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$

$$A^{-1} = \frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Exercise

Compute inverse of A using python

```
>>> A = [ [ 3, 4 ],
          [2, 3] ]
>>>
```

Solution

```
>>> from numpy import *
>>> A = [[ 3, 4 ], [2, 3]]
>>> B = linalg.inv(A)
array([[ 3., -4.],[-2., 3.]])
>>> dot(A,B)                    # Not A*B
array([[ 1., 0.], [ 0., 1.]])
```

Python Turtle Graphics

```
C:\> c:\python33\python.exe
>>> import turtle      # load a library
>>> dir(turtle)        # lists members of turtle
>>> turtle.circle(100)     # draws a circle
```

Turtle Graphics in Python3

```
C:\> cat random-walk.py
#!python3
import turtle, random
wn = turtle.Screen()
turtle.colormode(255)
turtle.speed(0)
for i in range(1000):
    turtle.forward(5)
    turtle.left(random.random() * 180 - 90)
    turtle.color(i%255,(i+100)%255,(i+200)%25
wn.mainloop()

C:\>  c:\python33\python.exe random-walk.py
```

Hilbert space-filling curve (animation)

C:\> c:\python33\python.exe hilbert.py

Sierpinski triangles

- Originally constructed as a curve, this is one of the basic examples of self-similar sets, i.e. it is a mathematically generated pattern that can be reproducible at any magnification or reduction.
- The Sierpinsky Triangle is a fractal created by taking a triangle, decreasing the height and width by 1/2, creating 3 copies of the resulting triangle, and place them such each triangle touches the other two on a corner. This process is repeated over and over again with the resulting triangles to produce the **Sierpinski** triangle:

Coordinate Transformations

Given a point p = vector(x1,y1,1), we can do:

Translation

```
| x2 |   |1 0 tx| |x1|
| y2 | = |0 1 ty|*|y1|
| 1  |   |0 0  1| |1 |
```

Scaling

```
| x2 |   |sx 0 0| |x1|
| y2 | = |0 sy 0|*|y1|
| 1  |   |0  0 1| |1 |
```

Rotation

```
| x2 |   |cos(v) -sin(v) 0| |x1|
| y2 | = |sin(v) cos(v)  0|*|y1|
| 1  |   |0        0     1| |1 |
```

We can combine several matrix operations into a single matrix, because matrix multiplication is associative.

Wallpaper patterns, only 17 different symmetric tilings are possible

The 17 Wallpaper groups

Penrose non-periodic tilings

Bézier curve

drawing.bezier 20,120, 20,20, 320,20, 320,120

drawing.bezier 25,125, 100,25, 400,25, 325,125

drawing.bezier 100,150, 25,50, 475, 50, 400,150

Drawing a bezier curve

$$P(t) = (1-t)^3P_0 + 3(1-t)^2tP_1 + 3(1-t)t^2P_2 + t^3P_3$$

with t=[0..1].

```
#!python3
import turtle
def turtle_cubic_bezier(x0, y0, x1, y1, x2, y2, x3, y3, n=20):
    turtle.penup()
    turtle.goto(x0,y0)
    turtle.pendown()
    for i in range(n+1):
        t = i / n
        a = (1. - t)**3
        b = 3. * t * (1. - t)**2
        c = 3.0 * t**2 * (1.0 - t)
        d = t**3
        x = int(a * x0 + b * x1 + c * x2 + d * x3)
        y = int(a * y0 + b * y1 + c * y2 + d * y3)
        turtle.goto(x,y)
    turtle.goto(x3,y3)
turtle_cubic_bezier(0,0, 0,100, 100,0, 100,100)
```

Homework

- Install python33
- Use turtle graphics to draw a figure recursively.
- Example, the H tree:

Python 2.X Reference

(or use google)

Data types

- Numbers -- 3.1415, 1234, 999L, 3+4j
- Strings -- 'spam', "guido's"
- Lists -- [1, [2, 'three'], 4]
- Dict -- {'food':'spam', 'taste':'yum'}
- Tuples -- (1,'spam', 4, 'U')
- Files -- text = open('eggs', 'r').read()

Constants

- 1234, -24, 0 -- integers (C longs)
- 999999999999L -- Long integers (unlimited size)
- 1.23, 3.14e-10, 4E210, 4.0e+210 -- Floats (C doubles)
- 0177, 0x9ff -- Octal and hex
- 3+4j, 3.0+4.0j, 3J -- Complex number

Operators

- Operators -- Description
- lambda args: expression -- anonymous function
- x or y, -- Logical 'or' short-circuit
- x and y -- Logical 'and' short-circuit
- not x -- Logical negation
- in, not in -- sequence membership
- x | y -- Bitwise or
- x ^ y -- Bitwise exclusive or
- x & y -- Bitwise and
- x << y, x >> y -- Shift x left or right by y bits
- x + y, x - y -- Addition/concat, sub
- x * y, x / y, x % y -- Mult/repeat, div, mod/format
- -x, +x, ~x -- Unary negation, ident, bit complement
- x[i], x[i:j], x.y, x(...) -- Indexing, slicing, member, calls
- (...), [...], {...}, `...` -- Tuple, list, dictionary, to-string

String operations

- s1 = '' -- Empty string
- s2 = "spam's" -- Double quotes
- block = """...""" -- Triple-quoted blocks
- s1 + s2, -- Concatenate,
- s2 * 3 -- repeat
- s2[i], -- Index,
- s2[i:j], -- slice,
- len(s2) -- length
- "a %s parrot" % 'dead' -- String formatting
- for x in s2, -- Iteration,
- 'm' in s2 -- membership

Slicing

```
>>> S = 'spam'
>>> S[0], S[-2]   # indexing, front or end
('s', 'a')
>>> S[1:3], S[1:], S[:-1]  # slice
('pa', 'pam', 'spa')
```

String formatting

```
>>> exclamation = "Hi"
>>> "Who said %s!" % exclamation
'Who said Hi!'
>>> "%d %s %d you" % (1, ‘like', 4)
'1 like 4 you'
>>> "%s -- %s -- %s" % (42, 3.14, [1, 2, 3])
'42 -- 3.14 -- [1, 2, 3]'
```

Format specifiers

- % -- String (or any object's print format)
- %X -- Hex integer (uppercase)
- %c -- Character
- %e -- Floating-point format 1[6]
- %d -- Decimal (int)
- %E -- Floating-point format 2
- %i -- Integer
- %f -- Floating-point format 3
- %u -- Unsigned (int)
- %g -- Floating-point format 4
- %o -- Octal integer
- %G -- Floating-point format 5
- %x -- Hex integer
- %% -- Literal %

Escaping chars

- \newline -- Ignored (a continuation)
- \n -- Newline (linefeed)
- \\ -- Backslash (keeps one \)
- \v -- Vertical tab
- \' -- Single quote (keeps ')
- \t -- Horizontal tab
- \" -- Double quote (keeps ")
- \r -- Carriage return
- \a -- Bell
- \f -- Formfeed
- \b -- Backspace
- \0XX -- Octal value XX
- \e -- Escape (usually)
- \xXX -- Hex value XX
- \000 -- Null (doesn't end string)
- \other -- Any other char (retained)

List operations

- L1 = [] -- An empty list
- L2 = [0, 1, 2, 3] -- Four items: indexes 0.3
- L3 = ['abc', ['def', 'ghi']] -- Nested sublists
- L2[i], L3[i][j] -- Index,
- L2[i:j], -- slice,
- len(L2) -- length
- L1 + L2, -- Concatenate,
- L2 * 3 -- repeat
- for x in L2, -- Iteration,
- 3 in L2 -- membership
- L2.append(4), -- Methods: grow,
- L2.sort(), -- sort,
- L2.index(1), -- search,
- L2.reverse() -- reverse, etc.
- del L2[k], -- Shrinking
- L2[i:j] = [] -- slice assignment
- L2[i] = 1, -- Index assignment,
- L2[i:j] = [4,5,6] -- range(4), xrange(0, 4)

Dictionary/Hash table operations

- d1 = {} -- Empty dictionary
- d2 = {'spam': 2, 'eggs': 3} -- Two-item dictionary
- d3 = {'food': {'ham': 1, 'egg': 2}} -- Nesting
- d2['eggs'], d3['food']['ham'] -- Indexing by key
- d2.has_key('eggs'), -- Methods: membership test,
- d2.keys(), -- keys list,
- d2.values() -- values list, etc.
- len(d1) -- Length (number stored entries)
- d2[key] = new, -- Adding/changing,
- del d2[key] -- deleting

Tuple operations

- () -- An empty tuple
- t1 = (0,) -- A one-item tuple (not an expression)
- t2 = (0, 1, 2, 3) -- A four-item tuple
- t2 = 0, 1, 2, 3 -- Another four-item tuple (same as prior line)
- t3 = ('abc', ('def', 'ghi')) -- Nested tuples
- t1[i], t3[i][j] t1[i:j], len(t1) -- Index, slice, length
- t1 + t2 t2 * 3 -- Concatenate, repeat
- for x in t2, -- membership
- 3 in t2 -- Iteration,

File operations

- output = open('/tmp/spam', 'w') -- Create output file ('w' means write)
- input = open('data', 'r') -- Create input file ('r' means read)
- S = input.read() -- Read entire file into a single string
- S = input.read(N) -- Read N bytes (1 or more)
- S = input.readline() -- Read next line (through end-line marker)
- L = input.readlines() -- Read entire file into list of line strings
- output.write(S) -- Write string S onto file
- output.writelines(L) -- Write all line strings in list L onto file
- output.close() -- Manual close (or it's done for you when collected)

Mutable (can change)

- Object type -- Category--Mutable?
- Numbers -- Numeric--No
- Strings -- Sequence--No
- Lists -- Sequence--Yes
- Dictionaries -- Mapping--Yes
- Tuples -- Sequence--No
- Files -- Extension--N/A

Booleans

- Object -- Value
- “text" -- True (any non empty string)
- "" -- False (empty string).
- [] -- False (empty list)
- {} -- False (empty set)
- 1 -- True
- 0.0 -- False (float number)
- None -- False

Statements

- Statement -- Role -- Examples
- Assignment -- references -- curly, moe, larry = 'good', 'bad', 'ugly'
- Calls -- functions -- stdout.write("spam, ham, toast\n")
- Print -- Printing objects -- print 'The Killer', joke
- If/elif/else -- Selecting actions -- if "python" in text: print text
- For/else -- iteration -- for x in mylist: print x
- While/else -- General loops -- while 1: print 'hello'
- Pass -- placeholder -- while 1: pass
- Continue -- Loop jumps -- while 1: if not line: break
- Try/except/finally -- exceptions -- try: action() except: print 'error'
- Raise -- exception -- raise endSearch, location
- Import, From -- Module access -- import sys; from sys import stdin
- Def, Return -- functions -- def f(a, b, c=1, *d): return a+b+c+d[0]
- Class -- Building objects -- class subclass: staticData = []
- Global -- Namespaces -- def function(): global x, y; x = 'new'
- Del -- Deleting things -- del data[k]; del data[i:j]; del obj.attr
- Exec -- Running code strings -- exec "import " + modName in gdict, ldict
- Assert -- Debugging checks -- assert X > Y

keywords

and -- assert -- break -- class -- continue

def -- del -- elif -- else -- except

exec -- finally -- for -- from -- global

if -- import -- in -- is -- lambda

not -- or -- pass -- print -- raise

return -- try -- while

Syntax

- Operation -- meaning
- cook(eggs, cheese) -- Function call
- eggs.cook( ) -- Method calls
- print eggs( ) -- Interactive print
- a < b and c != d -- Compound expression
- a < b < c -- Range tests

Control: for, if, else

Printing prime numbers less than 10

```
>>> \
... for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'factors', x, '*', n/x)
...             break  # not a prime
...     else:
...         print(n, 'is a prime number')
```

Introduction to C

Brief Introduction to the C Programming Language

Fred Kuhns
fredk@cse.wustl.edu
Applied Research Laboratory,
Department of Computer Science and Engineering,
Washington University in St. Louis

Washington
WASHINGTON UNIVERSITY IN ST LOUIS

Introduction

- The C programming language was designed by Dennis Ritchie at Bell Laboratories in the early 1970s
- Influenced by
  - ALGOL 60 (1960),
  - CPL (Cambridge, 1963),
  - BCPL (Martin Richard, 1967),
  - B (Ken Thompson, 1970)
- Traditionally used for systems programming.
- Traditional C:
  - *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, 2nd Edition, Prentice Hall
  - Referred to as *K&R*

Standard C

- Standardized in 1989 by ANSI (American National Standards Institute) known as ANSI C
- International standard (ISO) in 1990 which was adopted by ANSI and is known as *C89*
- As part of the normal evolution process the standard was updated in 1995 (*C95*) and 1999 (*C99*)
- C++ and C
  - C++ extends C to include support for Object Oriented Programming and other features that facilitate large software development projects
  - C is not strictly a subset of C++, but it is possible to write "*Clean C*" that conforms to both the C++ and C standards.

Elements of a C Program

- A C development environment includes
  - *System libraries* and *headers*: a set of standard libraries and their header files. For example see `/usr/include` and `glibc`.
  - *Application Source*: application source and header files
  - *Compiler*: converts source to object code for a specific platform
  - *Linker*: resolves external references and produces the executable module
- User program structure
  - there must be one main function where execution begins when the program is run. This function is called main
    - `int main (void) { ... },`
    - `int main (int argc, char *argv[]) { ... }`
    - UNIX Systems have a 3rd way to define main(), though it is not POSIX.1 compliant
      `int main (int argc, char *argv[], char *envp[])`
  - additional local and external functions and variables

A Simple C Program

- *Create* example file: `try.c`
- *Compile* using gcc:
  `gcc -o try try.c`
- The standard C library *libc* is included automatically
- *Execute* program
  `./try`
- Note, I always specify an absolute path
- Normal termination:
  `void` **`exit`**`(int status);`
  - calls functions registered with `atexit()`
  - flush output streams
  - close all open streams
  - return status value and control to host environment

```
/* you generally want to
 * include stdio.h and
 * stdlib.h
 * */
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
   printf("Hello World\n");
   exit(0);
}
```

Source and Header files

- Just as in C++, place related code within the same module (i.e. file).
- Header files (`*.h`) export interface definitions
  - function prototypes, data types, macros, inline functions and other common declarations
- Do not place source code (i.e. definitions) in the header file with a few exceptions.
  - inline'd code
  - class definitions
  - const definitions
- *C preprocessor* (`cpp`) is used to insert common definitions into source files
- There are other cool things you can do with the preprocessor

Another Example C Program

/usr/include/stdio.h

```
/* comments */
#ifndef _STDIO_H
#define _STDIO_H

... definitions and prototypes

#endif
```

/usr/include/stdlib.h

```
/* prevents including file
 * contents multiple
 * times */
#ifndef _STDLIB_H
#define _STDLIB_H

... definitions and protoypes

#endif
```

`#include` directs the preprocessor to "include" the contents of the file at this point in the source file.
`#define` directs preprocessor to define macros.

example.c

```
/* This is a C-style comment.
 * You generally want to place
 * all file includes at start of file
 */
#include <stdio.h>
#include <stdlib.h>

int
main (int argc, char **argv)
{
  // this is a C++-style comment
  // printf prototype in stdio.h
  printf("Hello, Prog name = %s\n",
            argv[0]);
  exit(0);
}
```

Passing Command Line Arguments

- When you execute a program you can include arguments on the command line.
- The run time environment will create an argument vector.
  - `argv` is the argument vector
  - `argc` is the number of arguments
- Argument vector is an array of pointers to strings.
- a *string* is an array of characters terminated by a binary 0 (NULL or '\0').
- *argv[0]* is always the program name, so *argc* is at least 1.

```
./try -g 2 fred
```

C Standard Header Files you may want to use

- Standard Headers you should know about:
  - `stdio.h` - file and console (also a file) IO: *perror*, *printf*, *open*, *close*, *read*, *write*, *scanf*, etc.
  - `stdlib.h` - common utility functions: *malloc*, *calloc*, *strtol*, *atoi*, etc
  - `string.h` - string and byte manipulation: *strlen*, *strcpy*, *strcat*, *memcpy*, *memset*, etc.
  - `ctype.h` - character types: *isalnum*, *isprint*, *isupport*, *tolower*, etc.
  - `errno.h` - defines *errno* used for reporting system errors
  - `math.h` - math functions: *ceil*, *exp*, *floor*, *sqrt*, etc.
  - `signal.h` - signal handling facility: *raise*, *signal*, etc
  - `stdint.h` - standard integer: *intN_t*, *uintN_t*, etc
  - `time.h` - time related facility: *asctime*, *clock*, *time_t*, etc.

The Preprocessor

- The C preprocessor permits you to define simple macros that are evaluated and expanded prior to compilation.

- Commands begin with a '#'. Abbreviated list:
  - `#define` : defines a macro
  - `#undef` : removes a macro definition
  - `#include` : insert text from file
  - `#if` : conditional based on value of expression
  - `#ifdef` : conditional based on whether macro defined
  - `#ifndef` : conditional based on whether macro is not defined
  - `#else` : alternative
  - `#elif` : conditional alternative
  - `defined()` : preprocessor function: 1 if name defined, else 0

    ```
    #if defined(__NetBSD__)
    ```

Preprocessor: Macros

- Using macros as functions, exercise caution:
  - flawed example: `#define mymult(a,b) a*b`
    - Source: `k = mymult(i-1, j+5);`
    - Post preprocessing: `k = i - 1 * j + 5;`
  - better: `#define mymult(a,b) (a)*(b)`
    - Source: `k = mymult(i-1, j+5);`
    - Post preprocessing: `k = (i - 1)*(j + 5);`
- Be careful of *side effects*, for example what if we did the following
  - Macro: `#define mysq(a) (a)*(a)`
  - flawed usage:
    - Source: `k = mysq(i++)`
    - Post preprocessing: `k = (i++)*(i++)`
- Alternative is to use inline'ed functions
  - `inline int mysq(int a) {return a*a};`
  - `mysq(i++)` works as expected in this case.

Preprocessor: Conditional Compilation

- Its generally better to use inline'ed functions
- Typically you will use the preprocessor to define constants, perform conditional code inclusion, include header files or to create shortcuts
- `#define DEFAULT_SAMPLES  100`
- `#ifdef __linux`

```
static inline int64_t
  gettime(void) {...}
```

- `#elif defined(sun)`

```
static inline int64_t
  gettime(void) {return (int64_t)gethrtime()}
```

- `#else`

```
static inline int64_t
  gettime(void) {... gettimeofday()...}
```

- `#endif`

Another Simple C Program

```
int main (int argc, char **argv) {
  int i;
  printf("There are %d arguments\n", argc);
  for (i = 0; i < argc; i++)
    printf("Arg %d = %s\n", i, argv[i]);
  return 0;
}
```

- Notice that the syntax is similar to Java
- What's new in the above simple program?
  - of course you will have to learn the new interfaces and utility functions defined by the C standard and UNIX
  - Pointers will give you the most trouble

Arrays and Pointers

- A variable declared as an array represents a contiguous region of memory in which the array elements are stored.

  `int x[5]; // an array of 5 4-byte ints.`

- All arrays begin with an index of 0

memory layout for array x

- An array identifier is equivalent to a pointer that references the first element of the array
  - `int x[5], *ptr;`
    `ptr = &x[0]` is equivalent to `ptr = x;`
- Pointer arithmetic and arrays:
  - `int x[5];`
    `x[2]` is the same as `*(x + 2)`, the compiler will assume you mean 2 objects beyond element x.

Pointers

- For any type T, you may form a pointer type to T.
  - Pointers may reference a function or an object.
  - The value of a pointer is the address of the corresponding object or function
  - Examples: `int *i; char *x; int (*myfunc)();`
- Pointer operators: ***** dereferences a pointer, *&* creates a pointer (reference to)
  - ```
    int i = 3; int *j = &i;
    *j = 4; printf("i = %d\n", i); // prints i = 4
    ```
  - ```
    int myfunc (int arg);
    int (*fptr)(int) = myfunc;
    i = fptr(4); // same as calling myfunc(4);
    ```
- Generic pointers:
  - Traditional C used (char *)
  - Standard C uses (void *) – these can not be dereferenced or used in pointer arithmetic. So they help to reduce programming errors
- Null pointers: use ***NULL*** or ***0***.
- *It is a good idea to always initialize pointers to NULL.*

Pointers in C (and C++)

```
Step 1:
int main (int argc, argv) {
 int  x = 4;
 int *y = &x;
 int *z[4] = {NULL, NULL, NULL, NULL};
 int  a[4] = {1, 2, 3, 4};
...
```

Note: The compiler converts z[1] or *(z+1) to
Value at address (Address of z + sizeof(int));

In C you would write the byte address as:

```
(char *)z + sizeof(int);
```

or letting the compiler do the work for you

```
(int *)z + 1;
```

Pointers Continued

```
Step 1:
int main (int argc, argv) {
  int  x = 4;
  int *y = &x;
  int *z[4] = {NULL, NULL, NULL, NULL};
  int  a[4] = {1, 2, 3, 4};
Step 2: Assign addresses to array Z
  z[0] = a;      // same as &a[0];
  z[1] = a + 1;  // same as &a[1];
  z[2] = a + 2;  // same as &a[2];
  z[3] = a + 3;  // same as &a[3];
```

Pointers Continued

```
Step 1:
int main (int argc, argv) {
 int x = 4;
 int *y = &x;
 int *z[4] = {NULL, NULL, NULL, NULL};
 int a[4] = {1, 2, 3, 4};
Step 2:
 z[0] = a;
 z[1] = a + 1;
 z[2] = a + 2;
 z[3] = a + 3;
Step 3: No change in z's values
 z[0] = (int *)((char *)a);
 z[1] = (int *)((char *)a
                + sizeof(int));
 z[2] = (int *)((char *)a
                + 2 * sizeof(int));
 z[3] = (int *)((char *)a
                + 3 * sizeof(int));
```

Getting Fancy with Macros

```
#define QNODE(type)   \
struct {              \
  struct type *next;  \
  struct type **prev; \
}

#define QNODE_INIT(node, field)  \
  do {                           \
    (node)->field.next = (node); \
    (node)->field.prev =         \
            &(node)->field.next; \
  } while ( /* */ 0 );

#define QFIRST(head, field) \
        ((head)->field.next)

#define QNEXT(node, field) \
        ((node)->field.next)

#define QEMPTY(head, field) \
        ((head)->field.next == (head))

#define QFOREACH(head, var, field) \
  for ((var) = (head)->field.next; \
       (var) != (head);            \
       (var) = (var)->field.next)
```

```
#define QINSERT_BEFORE(loc, node, field) \
  do {                                   \
    *(loc)->field.prev = (node);         \
    (node)->field.prev =                 \
           (loc)->field.prev;            \
    (loc)->field.prev  =                 \
           &((node)->field.next);        \
    (node)->field.next = (loc);          \
  } while (/* */0)

#define QINSERT_AFTER(loc, node, field)        \
  do {                                         \
    ((loc)->field.next)->field.prev =          \
            &(node)->field.next;               \
    (node)->field.next = (loc)->field.next;    \
    (loc)->field.next = (node);                \
    (node)->field.prev = &(loc)->field.next;   \
  } while ( /* */ 0)

#define QREMOVE(node, field)                   \
  do {                                         \
    *((node)->field.prev) = (node)->field.next;\
    ((node)->field.next)->field.prev =         \
             (node)->field.prev;               \
    (node)->field.next = (node);               \
    (node)->field.prev = &((node)->field.next);\
  } while ( /* */ 0)
```

After Preprocessing and Compiling

```
typedef struct wth_t
{
  int state;
  QNODE(wth_t) alist;
} wth_t;
```

CPP

```
typedef struct wth_t {
  int state;
  struct {
    struct wth_t *next;
    struct wth_t **prev;
  } alist;
```

```
#define QNODE_INIT(node, field)                    \
  do {                                             \
    (node)->field.next = (node);                   \
    (node)->field.prev = &(node)->field.next;\
  } while ( /* */ 0 );
```

after GCC

}***head***: instance of wth_t

3 words in memory

0x100	0
0x104	0x00100
0x108	0x00104

QNODE_INIT(head, alist)

<integer> state
<address> next
<address> prev

QNODE Manipulations

before

	head
0x100	0
0x104	0x100
0x108	0x104

	node0
0x1a0	0
0x1a4	0x1a0
0x1a8	0x1a4

```
#define QINSERT_BEFORE(head, node, alist)\
  do {                                                \
    *(head)->alist.prev = (node);                     \
    (node)->alist.prev = (head)->alist.prev;          \
    (head)->alist.prev  = &(node)->alist.next;\
    (node)->alist.next = (head);                      \
  } while (/* */0)
```

QINSERT_BEFORE(head, node0, alist);

?

QNODE Manipulations

before

	head
0x100	0
0x104	0x100
0x108	0x104

	node0
0x1a0	0
0x1a4	0x1a0
0x1a8	0x1a4

```
#define QINSERT_BEFORE(head, node, alist)\
  do {                                                \
    *(head)->alist.prev = (node);                     \
    (node)->alist.prev = (head)->alist.prev;          \
    (head)->alist.prev  = &(node)->alist.next;\
    (node)->alist.next = (head);                      \
  } while (/* */0)
```

QINSERT_BEFORE(head, node0, alist);

	head
0x100	0
0x104	**0x1a0**
0x108	0x104

	node0
0x1a0	0
0x1a4	0x1a0
0x1a8	0x1a4

QNODE Manipulations

before

	head
0x100	0
0x104	0x100
0x108	0x104

	node0
0x1a0	0
0x1a4	0x1a0
0x1a8	0x1a4

```
#define QINSERT_BEFORE(head, node, alist)\
  do {                                    \
    *(head)->alist.prev = (node);         \
    (node)->alist.prev = (head)->alist.prev;  \
    (head)->alist.prev  = &(node)->alist.next;\
    (node)->alist.next = (head);          \
  } while (/* */0)
```

QINSERT_BEFORE(head, node0, alist);

QNODE Manipulations

before

	head
0x100	0
0x104	0x100
0x108	0x104

	node0
0x1a0	0
0x1a4	0x1a0
0x1a8	0x1a4

```
#define QINSERT_BEFORE(head, node, alist)\
  do {                                    \
    *(head)->alist.prev = (node);         \
    (node)->alist.prev = (head)->alist.prev;  \
    (head)->alist.prev  = &(node)->alist.next;\
    (node)->alist.next = (head);          \
  } while (/* */0)
```

QINSERT_BEFORE(head, node0, alist);

QNODE Manipulations

before

	head		node0
0x100	0	**0x1a0**	0
0x104	0x100	0x1a4	0x1a0
0x108	0x104	0x1a8	0x1a4

```
#define QINSERT_BEFORE(head, node, alist)\
  do {                                    \
    *(head)->alist.prev = (node);         \
    (node)->alist.prev = (head)->alist.prev;  \
    (head)->alist.prev  = &(node)->alist.next;\
    (node)->alist.next = (head);          \
  } while (/* */0)
```

QINSERT_BEFORE(head, node0, alist);

QNODE Manipulations

before

	head
0x100	0
0x104	0x100
0x108	0x104

	node0
0x1a0	0
0x1a4	0x1a0
0x1a8	0x1a4

```
#define QINSERT_BEFORE(head, node, alist)\
  do {                                                \
    *(head)->alist.prev = (node);                     \
    (node)->alist.prev = (head)->alist.prev;          \
    (head)->alist.prev  = &(node)->alist.next;\
    (node)->alist.next = (head);                      \
  } while (/* */0)
```

QINSERT_BEFORE(head, node0, alist);

Adding a Third Node

	head
0x100	0
0x104	0x1a0
0x108	0x1a4

	node0
0x1a0	0
0x1a4	0x100
0x1a8	0x104

	node1
0x200	0
0x204	0x200
0x208	0x204

```
#define QINSERT_BEFORE(head, node, alist)\
  do {                                                    \
    *(head)->alist.prev = (node);                         \
     (node)->alist.prev = (head)->alist.prev;             \
     (head)->alist.prev  = &(node)->alist.next;           \
     (node)->alist.next = (head);                         \
  } while (/* */0)
```

QINSERT_BEFORE(head, node1, alist);

Adding a Third Node

	head
0x100	0
0x104	0x1a0
0x108	0x1a4

	node0
0x1a0	0
0x1a4	0x100
0x1a8	0x104

	node1
0x200	0
0x204	0x200
0x208	0x204

```
#define QINSERT_BEFORE(head, node1, alist)\
  do {                                                    \
(1) *(head)->alist.prev = (node1);                        \
    (node1)->alist.prev = (head)->alist.prev;             \
    (head)->alist.prev  = &(node1)->alist.next;           \
    (node1)->alist.next = (head);                         \
  } while (/* */0)
```

QINSERT_BEFORE(head, node1, alist);

Adding a Third Node

	head
0x100	0
0x104	0x1a0
0x108	0x1a4

	node0
0x1a0	0
0x1a4	0x100
0x1a8	0x104

	node1
0x200	0
0x204	0x200
0x208	0x204

```
#define QINSERT_BEFORE(head, node1, alist)\
  do {                                                    \
     *(head)->alist.prev = (node1);                       \
(2)  (node1)->alist.prev = (head)->alist.prev;            \
     (head)->alist.prev  = &(node1)->alist.next;          \
     (node1)->alist.next = (head);                        \
  } while (/* */0)
```

QINSERT_BEFORE(head, node1, alist);

Adding a Third Node

	head
0x100	0
0x104	0x1a0
0x108	0x1a4

	node0
0x1a0	0
0x1a4	0x100
0x1a8	0x104

	node1
0x200	0
0x204	0x200
0x208	0x204

```
#define QINSERT_BEFORE(head, node1, alist)\
  do {                                                    \
 (1) *(head)->alist.prev = (node1);                       \
 (2)  (node1)->alist.prev = (head)->alist.prev;           \
 (3)  (head)->alist.prev  = &(node1)->alist.next;         \
      (node1)->alist.next = (head);                       \
  } while (/* */0)
```

QINSERT_BEFORE(head, node1, alist);

Adding a Third Node

	head
0x100	0
0x104	0x1a0
0x108	0x1a4

	node0
0x1a0	0
0x1a4	0x100
0x1a8	0x104

	node1
0x200	0
0x204	0x200
0x208	0x204

```
#define QINSERT_BEFORE(head, node1, alist)\
  do {                                                    \
 (1) *(head)->alist.prev = (node1);                       \
 (2)  (node1)->alist.prev = (head)->alist.prev;           \
 (3)  (head)->alist.prev  = &(node1)->alist.next;         \
 (4)  (node1)->alist.next = (head);                       \
  } while (/* */0)
```

QINSERT_BEFORE(head, node1, alist);

Removing a Node

	head
0x100	0
0x104	0x1a0
0x108	0x204

	node0
0x1a0	0
0x1a4	0x200
0x1a8	0x104

	node1
0x200	0
0x204	0x100
0x208	0x1a4

```
#define QREMOVE(node, alist)                                  \
  do {                                                        \
    (1) *((node)->alist.prev) = (node)->alist.next;           \
    (2) ((node)->alist.next)->alist.prev = (node)->alist.prev;\
    (3) (node)->alist.next = (node);                          \
    (4) (node)->alist.prev = &((node)->alist.next);           \
  } while ( /* */ 0)
```

QREMOVE(node0, alist);

	head
0x100	0
0x104	??
0x108	??

	node0
0x1a0	0
0x1a4	??
0x1a8	??

	node1
0x200	0
0x204	??
0x208	??

Removing a Node

	head
0x100	0
0x104	0x1a0
0x108	0x204

	node0
0x1a0	0
0x1a4	0x200
0x1a8	0x104

	node1
0x200	0
0x204	0x100
0x208	0x1a4

```
#define QREMOVE(node, alist)                                    \
  do {                                                          \
    *((node)->alist.prev) = (node)->alist.next;                 \
    ((node)->alist.next)->alist.prev = (node)->alist.prev;\
    (node)->alist.next = (node);                                \
    (node)->alist.prev = &((node)->alist.next);                 \
  } while ( /* */ 0)
```

QREMOVE(node0, alist);

Removing a Node

	head
0x100	0
0x104	0x1a0
0x108	0x204

	node0
0x1a0	0
0x1a4	0x200
0x1a8	0x104

	node1
0x200	0
0x204	0x100
0x208	0x1a4

```
#define QREMOVE(node0, alist)                                  \
  do {                                                         \
    (1) *((node0)->alist.prev) = (node0)->alist.next;    \
        ((node0)->alist.next)->alist.prev = (node0)->alist.prev;\
         (node0)->alist.next = (node0);                        \
         (node0)->alist.prev = &((node0)->alist.next);      \
  } while ( /* */ 0)
```

QREMOVE(node0, alist);

Removing a Node

	head
0x100	0
0x104	0x1a0
0x108	0x204

	node0
0x1a0	0
0x1a4	0x200
0x1a8	0x104

	node1
0x200	0
0x204	0x100
0x208	0x1a4

```
#define QREMOVE(node0, alist)                                  \
  do {                                                         \
        *((node0)->alist.prev) = (node0)->alist.next;          \
    (2) ((node0)->alist.next)->alist.prev = (node0)->alist.prev;\
        (node0)->alist.next = (node0);                         \
        (node0)->alist.prev = &((node0)->alist.next);          \
  } while ( /* */ 0)
```

QREMOVE(node0, alist);

Removing a Node

	head
0x100	0
0x104	0x1a0
0x108	0x204

	node0
0x1a0	0
0x1a4	0x200
0x1a8	0x104

	node1
0x200	0
0x204	0x100
0x208	0x1a4

```
#define QREMOVE(node0, alist)                                  \
  do {                                                         \
        *((node0)->alist.prev) = (node0)->alist.next;          \
         ((node0)->alist.next)->alist.prev = (node0)->alist.prev;\
    (3) (node0)->alist.next = (node0);                         \
        (node0)->alist.prev = &((node0)->alist.next);          \
  } while ( /* */ 0)
```

QREMOVE(node0, alist);

Removing a Node

	head
0x100	0
0x104	0x1a0
0x108	0x204

	node0
0x1a0	0
0x1a4	0x200
0x1a8	0x104

	node1
0x200	0
0x204	0x100
0x208	0x1a4

```
#define QREMOVE(node0, alist)                                  \
  do {                                                         \
        *((node0)->alist.prev) = (node0)->alist.next;          \
         ((node0)->alist.next)->alist.prev = (node0)->alist.prev;\
        (node0)->alist.next = (node0);                         \
    (4) (node0)->alist.prev = &((node0)->alist.next);          \
  } while ( /* */ 0)
```

QREMOVE(node0, alist);

Solution to Removing a Node

	head
0x100	0
0x104	0x1a0
0x108	0x204

	node0
0x1a0	0
0x1a4	0x200
0x1a8	0x104

	node1
0x200	0
0x204	0x100
0x208	0x1a4

```
#define QREMOVE(node, alist)                                  \
  do {                                                        \
    (1) *((node)->alist.prev) = (node)->alist.next;           \
    (2) ((node)->alist.next)->alist.prev = (node)->alist.prev;\
    (3) (node)->alist.next = (node);                          \
    (4) (node)->alist.prev = &((node)->alist.next);           \
  } while ( /* */ 0)
```

QREMOVE(node0, alist);

Functions

- Always use function prototypes

```
int myfunc (char *, int, struct MyStruct *);
int myfunc_noargs (void);
void myfunc_noreturn (int i);
```

- C and C++ are *call by value*, copy of parameter passed to function
  - C++ permits you to specify pass by reference
  - if you want to alter the parameter then pass a pointer to it (or use references in C++)
- If performance is an issue then use inline functions, generally better and safer than using a macro. Common convention
  - define prototype and function in header or *name*.i file
  - `static inline int myinfunc (int i, int j);`
  - `static inline int myinfunc (int i, int j) { ... }`

Basic Types and Operators

- Basic data types
  - Types: *char, int, float and double*
  - Qualifiers: *short, long, unsigned, signed, const*
- Constant: 0x1234, 12, "Some string"
- Enumeration:
  - Names in different enumerations must be distinct
  - `enum WeekDay_t {Mon, Tue, Wed, Thur, Fri};`
    `enum WeekendDay_t {Sat = 0, Sun = 4};`
- Arithmetic: +, -, *, /, %
  - prefix ++i or --i ; increment/decrement before value is used
  - postfix i++, i--; increment/decrement after value is used
- Relational and logical: <, >, <=, >=, ==, !=, &&, ||
- Bitwise: &, |, ^ (xor), <<, >>, ~(ones complement)

Operator Precedence (from “C a Reference Manual”, 5th Edition)

Tokens	Operator	Class	Precedence	Associates
names, literals	simple tokens	primary	16	n/a
***a*[k]**	subscripting	postfix		left-to-right
***f*(...)**	function call	postfix		left-to-right
.	direct selection	postfix		left-to-right
->	indirect selection	postfix		left to right
++ --	increment, decrement	**postfix**		left-to-right
(*type*){*init*}	compound literal	postfix		left-to-right
++ --	increment, decrement	**prefix**	15	right-to-left
sizeof	size	unary		right-to-left
~	bitwise not	unary		right-to-left
!	logical not	unary		right-to-left
- +	negation, plus	unary		right-to-left
&	address of	unary		right-to-left
*****	indirection (*dereference*)	unary		right-to-left

Tokens	Operator	Class	Precedence	Associates
(*type*)	casts	unary	14	right-to-left
*** / %**	multiplicative	binary	13	left-to-right
+ -	additive	binary	12	left-to-right
<< >>	left, right shift	binary	11	left-to-right
< <= > >=	relational	binary	10	left-to-right
== !=	equality/ineq.	binary	9	left-to-right
&	bitwise and	binary	8	left-to-right
^	bitwise xor	binary	7	left-to-right
|	bitwise or	binary	6	left-to-right
&&	logical and	binary	5	left-to-right
||	logical or	binary	4	left-to-right
?:	conditional	ternary	3	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	assignment	binary	2	right-to-left
,	sequential eval.	binary	1	left-to-right

Structs and Unions

- structures
  - `struct MyPoint {int x, int y};`
  - `typedef struct MyPoint MyPoint_t;`
  - `MyPoint_t point, *ptr;`
  - `point.x = 0;point.y = 10;`
  - `ptr = &point; ptr->x = 12; ptr->y = 40;`
- unions
  - `union MyUnion {int x; MyPoint_t pt; struct {int 3; char c[4]} S;};`
  - `union MyUnion x;`
  - Can only use one of the elements. Memory will be allocated for the largest element

Conditional Statements (if/else)

```
if (a < 10)
  printf("a is less than 10\n");
else if (a == 10)
  printf("a is 10\n");
else
  printf("a is greater than 10\n");
```

- If you have compound statements then use brackets (blocks)
  - ```
    if (a < 4 && b > 10) {
      c = a * b; b = 0;
      printf("a = %d, a\'s address = 0x%08x\n", a, (uint32_t)&a);
    } else {
      c = a + b; b = a;
    }
    ```
- These two statements are equivalent:
  - `if (a) x = 3; else if (b) x = 2; else x = 0;`
  - `if (a) x = 3; else {if (b) x = 2; else x = 0;}`
- Is this correct?
  - ```
    if (a) x = 3; else if (b) x = 2;
    else (z) x = 0; else x = -2;
    ```

Conditional Statements (switch)

```
int c = 10;
switch (c) {
  case 0:
    printf("c is 0\n");
    break;
  ...
  default:
    printf("Unknown value of c\n");
    break;
}
```

- What if we leave the break statement out?
- Do we need the final break statement on the default case?

Loops

```
for (i = 0; i < MAXVALUE; i++) {
    dowork();
}
while (c != 12) {
    dowork();
}
do {
    dowork();
} while (c < 12);
```

- flow control
  - **break** – exit innermost loop
  - **continue** – perform next iteration of loop
- Note, all these forms permit one statement to be executed. By enclosing in brackets we create a block of statements.

Building your program

- For all labs and programming assignments:
  - you must supply a make file
  - you must supply a README file that describes the assignment and results. This must be a text file, no MS word.
  - of course the source code and any other libraries or utility code you used
  - you may submit plots, they must be postscript or pdf

make and Makefile - Overview

- Why use make?
  - convenience of only entering compile directives once
  - make is smart enough (with your help) to only compile and link modules that have changed or which depend on files that have changed
  - allows you to hide platform dependencies
  - promotes uniformity
  - simplifies my (and hopefully your) life when testing and verifying your code
- A Makefile contains a set of rules for building a program

```
target ... : prerequisites ...
        command
        ...
```

- Static pattern rules.
  - each target is matched against target-pattern to derive stem which is used to determine prereqs (see example)

```
targets ... : target-pattern : prereq-patterns ...
        command
        ...
```

Makefiles

- Defining variables
```
MyOPS := -DWTH
MyDIR ?= /home/fred
MyVar   = $(SHELL)
```
- Using variables
```
MyFLAGS := $(MyOPS)
```
- Built-in Variables
  - $@ = filename of target
  - $< = name of the first prerequisites
- Patterns
  - use % character to determine stem
  - foo.o matches the pattern %.o with foo as the stem.
  - foo.o moo.o : %.o : %.c # says that foo.o depends on foo.c and moo.o depends on moo.c

Example Makefile for wulib

Makefile.inc

```
# Makefile.inc
# Contains common definitions

MyOS              := $(shell uname -s)
MyID              := $(shell whoami)
MyHost            := $(shell hostname)
WARNSTRICT        := -W \
                     -Wstrict-prototypes
\
                    -Wmissing-prototypes
WARNLIGHT         := -Wall
WARN              := ${WARNLIGHT}
ALLFLGS           := -D_GNU_SOURCE \
                    -D_REENTRANT \
                    -D_THREAD_SAFE

APPCFLGS          = $(ALLFLGS) \
                    $(WARN)

WUCC              := gcc
WUCFLAGS          := -DMyOS=$(MyOS) \
                    $(OSFLAGS) \
                    $(ALLFLGS) $(WARN)

WUINCLUDES        :=
WULIBS            := -lm

ifeq (${MyOS), SunOS)
OSLIBS+= -lrt
endif
```

Makefile

```
# Project specific
include ../Makefile.inc
INCLUDES    = ${WUINCLUDES} –I.
LIBS        = ${WILIBS} ${OSLIBS}
CFLAGS      = ${WUCLFAGS} –DWUDEBUG
CC          = ${WUCC}

HDRS        := util.h
CSRCS       := testapp1.c testapp2.c
SRCS        := util.c callout.c
COBJS       = $(addprefix ${OBJDIR}/, \
                          $(patsubst %.c,%.o,$(CSRCS)))
OBJS        = $(addprefix ${OBJDIR}/, \
                          $(patsubst %.c,%.o,$(SRCS)))
CMDS        = $(addprefix ${OBJDIR}/, $(basename $(CSRCS)))

all :  $(OBJDIR) $(CMDS)

install : all

$(OBJDIR) :
            mkdir $(OBJDIR)

$(OBJS)  $(COBJS) :  ${OBJDIR}/%.o :  %.c  $(HDRS)
            ${CC} ${CFLAGS} ${INCLUDES} –o $@ -c $<

$(CMDS) : ${OBJDIR}/% : ${OBJDIR}/%.o $(OBJS)
            ${CC} ${CFLAGS} -o $@ $@.o ${LIBS}
            chmod 0755 $@

clean :
            /bin/rm -f $(CMDS) $(OBJS)
```

Project Documentation

- README file structure
  - ***Section A***: *Introduction*
    describe the project, paraphrase the requirements and state your understanding of the assignments value.
  - ***Section B***: *Design and Implementation*
    List all files turned in with a brief description for each. Explain your design and provide simple psuedo-code for your project. Provide a simple flow chart of you code and note any constraints, invariants, assumptions or sources for reused code or ideas.
  - ***Section C***: *Results*
    For each project you will be given a list of questions to answer, this is where you do it. If you are not satisfied with your results explain why here.
  - ***Section D***: *Conclusions*
    What did you learn, or not learn during this assignment. What would you do differently or what did you do well.

Attacking a Project

- *Requirements and scope*: Identify specific requirements and or goals. Also note any design and/or implementation environment requirements.
  - knowing when you are done, or not done
  - estimating effort or areas which require more research
  - programming language, platform and other development environment issues
- *Approach*: How do you plan to solve the problem identified in the first step. Develop a prototype design and document. Next figure out how you will verify that you did satisfy the requirements/goals. Designing the tests will help you to better understand the problem domain and your proposed solution
- *Iterative development*: It is good practice to build your project in small pieces. Testing and learning as you go.
- *Final Touches*: Put it all together and run the tests identified in the approach phase. Verify you met requirements. Polish you code and documentation.
- *Turn it in.*

Approximation Algorithms

CLR Chapter 35

Definitions

- *Approximation Algorithm*
  - produces "near-optimal" solution
- Algorithm has *approximation ratio* $\rho(n)$ if:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n)$$

C = cost of algorithm's solution
C^* = cost of <u>optimal</u> solution
n = number of inputs = size of instance

- *Approximation Scheme*
  - (1+ε)-approximation algorithm for fixed ε
    - ε > 0 is an input
  - *Polynomial-Time Approximation Scheme*
    - time is polynomial in n
  - *Fully Polynomial-Time Approximation Scheme*
    - time is also *polynomial* in (1/ε)
    - constant-factor decrease in ε causes only constant-factor running-time increase

source: 91.503 textbook Cormen et al.

Overview

- VERTEX-COVER
  - Polynomial-time 2-approximation algorithm
- TSP
  - TSP with triangle inequality
    - Polynomial-time 2-approximation algorithm
  - TSP without triangle inequality
    - Negative result on polynomial-time $\rho(n)$-approximation algorithm
- SET-COVER
    - polynomial-time $\rho(n)$-approximation algorithm
      - $\rho(n)$ is a logarithmic function of set size
- SUBSET-SUM
  - Fully polynomial-time approximation scheme
- MAX-3-CNF Satisfiability
  - Randomized $\rho(n)$-approximation algorithm

Vertex Cover

Polynomial-Time 2-Approximation Algorithm

Vertex Cover

Vertex Cover of an undirected graph G=(V,E) is a subset

$$V' \subseteq V \text{ such that if } (u,v) \in E, \text{ then } u \in V' \text{ or } v \in V' \text{ or both}$$

NP-Complete

[GT1] VERTEX COVER

INSTANCE: Graph $G = (V,E)$, positive integer $K \leqslant |V|$.

QUESTION: Is there a vertex cover of size K or less for G, i.e., a subset $V' \subseteq V$ with $|V'| \leqslant K$ such that for each edge $\{u,v\} \in E$ at least one of u and v belongs to V'?

Reference: [Karp, 1972]. Transformation from 3SAT (see Chapter 3).

Comment: Equivalent complexity to INDEPENDENT SET with respect to restrictions on G. Variation in which the subgraph induced by V' is required to be connected is also NP-complete, even for planar graphs with no vertex degree exceeding 4 [Garey and Johnson, 1977a]. Easily solved in polynomial time if V' is required to be both a vertex cover and an independent set for G. The related EDGE COVER problem, in which one wants the smallest set $E' \subseteq E$ such that every $v \in V$ belongs to at least one $e \in E'$, can be solved in polynomial time by graph matching (e.g., see [Lawler, 1976a]).

source: Garey & Johnson

Vertex Cover

```
APPROX-VERTEX-COVER(G)
C ← ∅
E′ ← E[G]
while E′ ≠ ∅
    do let (u, v) be an arbitrary edge of E′
        C ← C ∪ {u, v}
        remove from E′ every edge incident on either u or v
return C
```

Figure 35.1 The operation of APPROX-VERTEX-COVER. **(a)** The input graph G, which has 7 vertices and 8 edges. **(b)** The edge (b, c), shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover being created. Edges (a, b), (c, e), and (c, d), shown dashed, are removed since they are now covered by some vertex in C. **(c)** Edge (e, f) is chosen; vertices e and f are added to C. **(d)** Edge (d, g) is chosen; vertices d and g are added to C. **(e)** The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices b, c, d, e, f, g. **(f)** The optimal vertex cover for this problem contains only three vertices: b, d, and e.

source: 91.503 textbook Cormen et al.

Vertex Cover

Theorem*:** ***APPROX-VERTEX-COVER *is a* ***polynomial-time 2-approximation algorithm****.*

***Proof*:** Let C^* be an optimal cover

C be cover from APPROX - VERTEX - COVER

$A \leftarrow$ edges chosen by APPROX - VERTEX - COVER

Observe: no 2 edges of *A* share any vertices due to removal of incident edges

$\Rightarrow$ Any vertex cover must include >= 1 endpoint of each edge in *A* $\Rightarrow$ $|C^*| \geq |A|$

APPROX-VERTEX-COVER adds both endpoints of each edge of *A* $\Rightarrow$ $|C| = 2A$

$$|C| = 2|A| \leq 2|C^*| \xrightarrow{\text{transitivity}} |C| \leq 2|C^*|$$

Algorithm runs in time polynomial in *n*.

Traveling Salesman

TSP with triangle inequality

Polynomial-time 2-approximation algorithm

TSP without triangle inequality

Negative result on polynomial-time $\rho(n)$-approximation algorithm

Hamiltonian Cycle

Figu 34.2 **(a)** A graph representing the vertices, edges, and faces of a dodecahedron, with a hamiltonian cycle shown by shaded edges. **(b)** A bipartite graph with an odd number of vertices. Any such graph is nonhamiltonian.

Hamiltonian Cycle of an undirected graph G=(V,E) is a simple cycle that contains each vertex in V.

NP-Complete

source: 91.503 textbook Cormen et al.

Traveling Salesman Problem (TSP)

TSP Tour of a complete, undirected graph G=(V,E) is a Hamiltonian Cycle with a designated starting/ending vertex.

TSP Decision Problem:

$$\{< G, c, k >:$$

$$c : V \times V \rightarrow Z$$

$$k \in Z$$

$$G \text{ has } TSP - tour \text{ of } \text{cost} \leq \text{k}\}$$

cost function

NP-Complete

source: 91.503 textbook Cormen et al.

Minimum Spanning Tree: Greedy Algorithms

Time: O(|E|lg|E|) given fast FIND-SET, UNION

```
MST-KRUSKAL(G, w)
A ← ∅
for each vertex v ∈ V[G]
    do MAKE-SET(v)
sort the edges of E by nondecreasing weight w
for each edge (u, v) ∈ E, in order by nondecreasing weight
    do if FIND-SET(u) ≠ FIND-SET(v)
        then A ← A ∪ {(u, v)}
            UNION(u, v)
return A
```

Invariant: Minimum weight spanning forest

Becomes single tree at end

Time: O(|E|lg|V|) = O(|E|lg|E|) slightly faster with fast priority queue

```
MST-PRIM(G, w, r)
Q ← V[G]
for each u ∈ Q
    do key[u] ← ∞
key[r] ← 0
π[r] ← NIL
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
        for each v ∈ Adj[u]
            do if v ∈ Q and w(u, v) < key[v]
                then π[v] ← u
                    key[v] ← w(u, v)
```

Invariant: Minimum weight tree

Spans all vertices at end

Produces minimum weight tree of edges that includes every vertex.

for Undirected, Connected, Weighted Graph G=(V,E)

source: 91.503 textbook Cormen et al.

TSP with Triangle Inequality

Cost Function Satisfies

Triangle Inequality

$$\forall u, v, w \in V$$

$$c(u,w) \le c(u,v) + c(v,w)$$

APPROX-TSP-TOUR(G, c)

1 select a vertex $r \in V[G]$ to be a "root" vertex
2 compute a minimum spanning tree T for G from root r
using MST-PRIM(G, c, r)
3 let L be the list of vertices visited in a preorder tree walk of T
4 **return** the hamiltonian cycle H that visits the vertices in the order L

"full walk" = abcbhbadefegeda

final approximate tour optimal tour

source: 91.503 textbook Cormen et al.

TSP with Triangle Inequality

Theorem*:** ***APPROX-TSP-TOUR *is a* ***polynomial-time 2-approximation algorithm for TSP with triangle inequality****.*

***Proof*:** Algorithm runs in time polynomial in *n*.

Let H^* be an optimal tour and T be MST

$c(T) \le c(H^*)$ (since deleting 1 edge from a tour creates a spanning tree)

Let W be a full walk of T (lists vertices when they are first visited and when returned to after visiting subtree)

$c(W) = 2c(T)$ (since full walk traverses each edge of T twice)

$$c(W) \le 2c(H^*)$$

Now make W into a tour H using triangle inequality.

$c(H) \le c(W)$ (New inequality holds since H is formed by deleting duplicate vertices from W.)

$$c(H) \le 2c(H^*)$$

TSP without Triangle Inequality

***Theorem*: If** $P \neq NP$**,** *then for any constant* $\rho \geq 1$

~~polynomial-time approximation algorithm with ratio~~ ρ

for TSP without triangle inequality.

Proof: *(by contradiction)* Suppose there **is** one --- call it *A*

Showing how to use *A* to solve NP-complete Hamiltonian Cycle problem ⇒ contradiction!

Convert instance G of Hamiltonian Cycle into instance of TSP (in polynomial time):

$$E' = \{(u,v) : u,v \in V, u \neq v\}$$ (G'=(V,E') is complete graph on V)

$$c(u,v) = \begin{cases} 1 & (u,v) \in E \\ \rho|V|+1 & otherwise \end{cases}$$ (assign integer cost to each edge in E')

For TSP problem (G',c):

G has Hamiltonian Cycle ⇒ (G',c) contains tour of cost |V|

G does not have Hamiltonian Cycle ⇒ Tour of G' must use some edge not in E

Cost of that tour of G' $\geq (\rho|V|+1)+(|V|-1) = \rho|V|+|V| > \rho|V|$

Can use *A* on (G',c)! *A finds tour of cost at most* ρ*(length of optimal tour)*

G has Hamiltonian Cycle ⇒ *A* finds tour of cost at most $\rho|V|$ ⇒ cost = |V|

G does not have Hamiltonian Cycle ⇒ *A* finds tour of cost > ρ |V|

If $P \neq NP$ *, solving NP-complete hamiltonian cycle in polynomial time is a contradiction!*

Set Cover

Greedy Approximation Algorithm

polynomial-time $\rho(n)$-approximation algorithm

$\rho(n)$ is a logarithmic function of set size

Set Cover Problem

Instance (X, F):

finite set X

family F of subsets of X

$$X = \bigcup_{S \in \mathsf{F}} S$$

NP-Complete

source: 91.503 textbook Cormen et al.

Greedy Set Covering Algorithm

```
GREEDY-SET-COVER(X, 𝓕)
U ← X
𝓒 ← ∅
while U ≠ ∅
    do select an S ∈ 𝓕 that maximizes |S ∩ U|
        U ← U − S
        𝓒 ← 𝓒 ∪ {S}
return 𝓒
```

<u>Greedy</u>: select set that covers the most uncovered elements

source: 91.503 textbook Cormen et al.

Set Cover

Theorem: ***GREEDY-SET-COVER*** *is a* ***polynomial-time $\rho(n)$-approximation algorithm for*** $\rho(n) = H(\max\{|S| : S \in \mathcal{F}\})$

Proof:

$$\boxed{d\text{th harmonic number } H_d = \sum_{i=1}^{d} \frac{1}{i} = H(d)}$$

Algorithm runs in time polynomial in *n*.

$S_i = i\text{th subset selected}$ *selecting* S_i costs 1

$c_x = \text{cost of element } x \in X$ *paid <u>only</u> when x is covered <u>for the first time</u>*

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$$

assume x is covered <u>for the first time</u> by S_i

(spread cost evenly across all elements covered for first time by S_i)

Number of elements covered for first time by S_i

Set Cover (proof continued)

Theorem: ***GREEDY-SET-COVER*** *is a* ***polynomial-time $\rho(n)$-approximation algorithm for*** $\rho(n) = H(\max\{|S|: S \in \mathcal{F}\})$

Proof: ***(continued)***

Let C^* be an optimal cover

C be cover from GREEDY-SET-COVER

$$|C| = \sum_{x \in X} c_x$$

1 unit is charged at each stage of algorithm

Cost assigned to optimal cover: $\sum_{S \in C^*} \left(\sum_{x \in S} c_x \right)$

*Each x is in >= 1 S in C**

$$\sum_{S \in C^*} \left(\sum_{x \in S} c_x \right) \geq \sum_{x \in X} c_x$$

$$|C| \leq \sum_{S \in C^*} \left(\sum_{x \in S} c_x \right)$$

Set Cover (proof continued)

Theorem: ***GREEDY-SET-COVER*** *is a* ***polynomial-time $\rho(n)$-approximation algorithm for*** $\rho(n) = H(\max\{|S|: S \in \mathcal{F}\})$

Proof: ***(continued)***

How does this relate to harmonic numbers??

$$\boxed{d\text{th harmonic number } H_d = \sum_{i=1}^{d} \frac{1}{i} = H(d)}$$

We'll show that: $\sum_{x \in S} c_x \le H(|S|)$

And then conclude that: $|C| \le \sum_{S \in C^*} H(|S|) \le |C^*| H(\max\{|S| : S \in \mathcal{F}\})$

which will finish the proof

Set Cover (proof continued)

***Proof* of:** $\sum_{x \in S} c_x \le H(|S|)$

$u_i = |S - (S_1 \cup S_2 \cup \cdots S_i)|$ *For some set S: Number of elements of S remaining uncovered after $S_1 \ldots S_i$ selected* $u_0 = |S|$

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} \left((u_{i-1} - u_i) \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \right)$$

k = least index for which u_k=0.

$$\sum_{x \in S} c_x \le \sum_{i=1}^{k} \left((u_{i-1} - u_i) \frac{1}{u_{i-1}} \right)$$

Since, due to greedy nature of algorithm: $|S_i - (S_1 \cup S_2 \cup \cdots S_{i-1})| \ge |S - (S_1 \cup S_2 \cup \cdots S_{i-1})| = u_{i-1}$

$$= \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \le \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j}$$

since j <= u_{i-1}

$$= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right)$$

$$= \sum_{i=1}^{k} \left(H(u_{i-1}) - H(u_i) \right) = H(u_0) - H(u_k) = H(u_0) - H(0) = H(u_0) = H(|S|)$$

telescoping sum

Subset-Sum

Exponential-Time Exact Algorithm

Fully Polynomial-Time Approximation Scheme

Subset-Sum Problem

Instance (S,t): $\quad S = \{x_1, x_2, \ldots, x_n\}$ positive integers

Decision Problem: $\exists S' \subseteq S : \sum_{s \in S'} s = t\,?$

Optimization Problem seeks subset with largest sum <= *t*

NP-Complete

source: 91.503 textbook Cormen et al.

Exponential-Time Exact Algorithm

P_i = set of values obtainable by selecting subset of $\{x_1, x_2, \ldots, x_i\}$ and summing elements.

Identity:

$$P_i = P_{i-1} \cup (P_{i-1} + x_i)$$

L_i is sorted list of every element of P_i <= t

$$L_i = \text{list of sums of subsets of } \{x_1, x_2, \ldots, x_i\} \le t$$

MERGE-LISTS(*L,L'*) returns sorted list = merge of sorted *L*, *L'* with duplicates removed.

```
EXACT-SUBSET-SUM(S, t)
n ← |S|
L_0 ← ⟨0⟩
for i ← 1 to n
    do L_i ← MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
        remove from L_i every element that is greater than t
return the largest element in L_n
```

source: 91.503 textbook Cormen et al.

Fully Polynomial-Time Approximation Scheme

```
TRIM(L, δ)
m ← |L|
L' ← ⟨y_1⟩
last ← y_1
for i ← 2 to m
    do if y_i > last·(1 + δ)          ▷ y_i ≥ last because L is sorted
          then append y_i onto the end of L'
               last ← y_i
return L'
```

```
APPROX-SUBSET-SUM(S, t, ε)
n ← |S|
L_0 ← ⟨0⟩
for i ← 1 to n
    do L_i ← MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
       L_i ← TRIM(L_i, ε/2n)
       remove from L_i every element that is greater than t
let z* be the largest value in L_n
return z*
```

Theorem: ***APPROX-SUBSET-SUM*** *is a* ***fully polynomial-time approximation scheme*** *for subset-sum.*

Proof: *see textbook*

source: 91.503 textbook Cormen et al.

MAX-3-CNF Satisfiability

3-CNF Satisfiability Background

Randomized Algorithms

Randomized MAX-3-CNF SAT Approximation Algorithm

MAX-3-CNF Satisfiability

- Background on Boolean Formula Satisfiability
  - <u>Boolean Formula Satisfiability</u>: Instance of language **SAT** is a boolean formula ϕ consisting of:
    - *n* boolean variables: $x_1, x_2, \ldots, x_n$
    - *m* boolean connectives: boolean function with 1 or 2 inputs and 1 output
      - e.g. AND, OR, NOT, implication, iff

$$\vee \quad \wedge \quad \neg \quad \rightarrow \quad \leftrightarrow$$

    - parentheses
    - truth, satisfying assignments notions apply

$$SAT = \{<\phi>: \phi \text{ is a satisfiable boolean formula}\}$$

NP-Complete

source: 91.503 textbook Cormen et al.

MAX-3-CNF Satisfiability
(continued)

- Background on 3-CNF-Satisfiability
  - Instance of language **SAT** is a boolean formula ϕ consisting of:
    - *literal:* variable or its negation
    - CNF = conjunctive normal form
      - *conjunction*: AND of clauses
      - *clause*: OR of literal(s)
    - 3-CNF: each clause has **exactly 3** distinct literals

MAX-3-CNF Satisfiability: optimization version of 3-CNF-SAT
- Maximization: satisfy as many clauses as possible
- Input Restrictions:
  - exactly 3 literals/clause
  - no clause contains both variable and its negation

NP-Complete

source: 91.503 textbook Cormen et al.

Definition

- Randomized Algorithm has *approximation ratio* $\rho(n)$ if, for *expected cost C* of solution produced by randomized algorithm:

$$\boxed{\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)}$$

size of input = *n*

source: 91.503 textbook Cormen et al.

Randomized Approximation Algorithm for MAX-3-CNF SAT

Theorem: *Given an instance of MAX-3-CNF satisfiability with n variables $x_1, x_2, \ldots, x_n$ with m clauses, the randomized algorithm that independently sets each variable to 1 with probability 1/2 and to 0 with probability 1/2 is a randomized 8/7-approximation algorithm.*

$$\boxed{\frac{C^*}{C} \le \frac{8}{7}}$$

Proof: independently set each variable to 0 or 1 with probability $= 1/2$

$$\text{indicator random variable} = Y_i = \mathrm{I}\{\text{event that clause } i \text{ is satisfied}\}$$

$$\Pr\{\text{clause } i \text{ is not satisfied}\} = (1/2)^3 = 1/8 \quad \Pr\{\text{clause } i \text{ is satisfied}\} = 1 - (1/8) = 7/8$$

$$\textit{Lemma}: \text{Given a sample space } S \text{ and an event } A \text{ in the sample space } S,$$

$$\text{Let } X_A = I\{A\}. \text{ Then } E[X_A] = \Pr\{A\}.$$

$$\text{By } \textit{Lemma}: E[Y_i] = 7/8 \qquad \text{Let } Y = Y_1 + Y_2 + \cdots + Y_m$$

$$E[Y] = E\left[\sum_{i=1}^{m} Y_i\right] = \sum_{i=1}^{m} E[Y_i] = \sum_{i=1}^{m} \frac{7}{8} = \frac{7m}{8} = \text{expected cost } C$$

$$\text{number of satisfied clauses} \le m \Rightarrow C^* \le m \Rightarrow \frac{C^*}{C} \le \frac{m}{7m/8} = \frac{8}{7} = \rho(n)$$

source: 91.503 textbook Cormen et al.