Competitive programming Bootcamp

Agenda

Why competitive programming?

- Write reliable and efficient programs
- Learn and practice algorithms
- Manage time when it’s very limited
- Do well at job interviews
- Join the community of highly motivated and
- smart people
- Have fun :)

Agenda

- Stage 0
  - Programming competitions
  - ACM-ICPC Contest introduction
  - ACM-ICPC Contest rules, scoring schema, team roles
  - IDE and setup
  - Online judge
  - Getting started
    - Reading in data from standard input (the console), or from a file
    - Elementary data structures
  - Testing
  - Structuring the code

Agenda

- Stage 1:
  - Data structures
  - Languages
  - Brute force solutions/Backtracking and pruning
  - Complexity and running time
  - Handling numbers
  - Other Search algorithms: Binary search, Greedy
  - Segment tree
  - Dynamic programming
- Stage 2 - Advanced
  - Graphs and traversals
  - Advanced graphs algorithms
  - Strings - Advanced handling
  - Bits - Advanced handling
  - Computational geometry

Agenda

- Stage 4 - More (NOT COVERED IN THE WORKSHOP)
  - Simple problem (codeforces – provide insight on failed test case and difficulty of problem)
  - Other online judges
  - Regional contest - Northeastern European Regional Contest (NEERC)
  - Past contests
  - Team notebook

Notation

- 6 exercices en ligne – 2 points
  - Fausse soumission -0.4 point
  - Au moins deux
  - Bonus pour les extras – Peut continuer après le cours mais n'est pas compté
- Questions – souvent non noté
- Quiz – tel que spécifié – coef 0.5

Assignment 1 - Not graded	Assignment 2 - slide 51	Assignment 3 - Quiz 1 testing	Assignment 4 - prog comp	Assignment 5- slide 106	Assignment 6- correctness	Assignment7 - Quiz 2 time complexity	Assignment 8 - quiz3 numbers	Assignment 8 - common struggles	Assign9 - greedy	Assign10 quiz4- segment tree	Assignment11 quiz5-Dp1	Assignment11 exos5- dp1	Assignment12-online judge / rattrapage	Assignment12 quiz dp2	Assignment12 exos6- dp2
	4	4	8	4	4	3	5	4	4	3	3	4	4	1	4

Cours

Code des UE	Contenu des enseignements		Enseignements		Travaux personnels de l'étudiant
	UE	ECU	Cours	TD/TP	
Unité de connaissances Fondamentales			90	0	60
ALP2200	Algorithmique et programmation	Programmation orientée objet avancée	25	20	30
		Programmation pour IPCC	25	20	30

References

- Code, solutions, problems and comments - https://github.com/NAU-ACM/ACM-ICPC-Preparation
- Book on data structures and algorithms, tips and tricks http://www.programming-challenges.com, Skiena, Revilla Programming Challenges The Programming Contest Training Manual, Springer (2003). –
- Course by skienna: https://www3.cs.stonybrook.edu/~skiena/392/audio/ or https://www.youtube.com/watch?v=3dkbFf82_b8&list=PL07B3F10B48592010

References

- Halim, Halim, Competitive Programming 3: The New Lower Bound of Programming Contests, Lulu Press (2014)
- Other online classes
  - https://www.coursera.org/lecture/algorithmic-toolbox/welcome-EagcP
  - Youtube playlist: MIT 6.006 Introduction to Algorithms, Fall 2011
  - Coursera: Princeton Algorithms course
  - Coursera: Stanford Algorithms course
  - Coursera: Competitive Programming

Other websites

- https://www.codingame.com/start
- https://www.hackerrank.com/
- C++ STL
  - Topcoder: Power up C++ with the Standard Template Library: Part 1
  - YoLinux: C++ STL Tutorial and Books at the end
  - A reference site: cplusplus.com

Credits

- IMSP
- Antoine Amarilli: https://a3nm.net/work/teaching/#y2018-inf280

Programming competition

Competitions

- Timed 2–5 hours
- Individual or team
- Several problems, solving each adds to the score
- Solutions are checked by an automated testing system

Problems

- Precisely formulated, although often through some real-world legend
- Input/output exactly of certain format
- Have tight time/memory limits
- Efficiency is key
- Often require knowledge of some algorithms/ideas

Solutions

- Program in one of the supported languages
- Usually short a few dozen lines
- Reads data in a specific format from standard input/input file
- Outputs solution in a specific format to standard output/output file
- Is repeatedly run on a testing system against prepared test cases
- Must not use external libraries, create extra files, go to the network and so on

Testing system verdicts

- ONLINE Judge verdict
  - Accepted (AC) – Congratulations!
  - Presentation Error (PE) – Check for spaces, left/right justification, line feeds, etc.
  - Accepted (PE) – Your program has a minor presentation error, but the judge is letting you off with a warning. Stop here and declare victory!
  - Wrong Answer (WA) – Your program returned an incorrect answer to one or more secret test cases.
  - Compile Error (CE) – The compiler could not figure out how to compile your program. The resulting compiler messages will be returned to you. Warning messages are ignored by the judge.
  - Runtime Error (RE) – Your program failed during execution due to a segmentation fault, floating point exception, or similar problem. Check for invalid pointer references or division by zero.
  - Submission Error (SE) – You did not correctly specify one or more of the information fields, perhaps giving an incorrect user ID or problem number.
  - Time Limit Exceeded (TL) – Your program took too much time on at least one of the test cases, so you likely have a problem with efficiency.
  - Memory Limit Exceeded (ML) – Your program tried to use more memory than the judge's default settings.
  - Output Limit Exceeded (OL) – Your program tried to print too much output, perhaps trapped in a infinite loop.
  - Restricted Function (RF) – Your source program tried to use an illegal system function such as fork() or fopen(). Behave yourself

Test cases

- Strictly formatted, no need to process typos, handle possible errors, and so on
- Range of possible values for each parameter is given in the statement
- However, you can’t assume anything about the data, except for what’s explicitly stated
- Be sure that problem authors will put test cases of any possible type
- No matter how extreme or nonsensical. Only compliance with the statement counts
- Usually, to earn score you need to pass all tests

Example of problem

Look for formal conditions/constraints

Formulation

You are given a list of

Input format

Sequence of contact

Names are non-empty

Constraints/limits
You may assume about tests only what's explicitly stated

s no more than 10 000.

Follow the format closely otherwise you get a reject

Output format

Output given names in alphabetical order. Each name on a new line.

Sample input

turing

dijkstra

knuth

Sample output

dijkstra

knuth

turing

- Intuition names are short, real-looking, distinct
- It's not stated that they belong to real people

abcdefg

aaaaa

- It's not stated that names have particular length

aaa...aa (letter 'a' 10000 times)

- It's not stated that they are distinct (line 'a' 10000 times)
- Length is not necessarily similar (letter 'a' 5000 times and line 'a' 5000 times)

- *Verify your understanding of the statement with them*
- *If something doesn't tie up reread the statement*
- *And later, check the correctness of your program*
- *But use other test cases, too!*
- *Samples are usually tested first when you submit*

Steps in solving a problem

- Read the statement
- Formalize it
- Invent a solution
- Prove it
- Implement it
- Test your implementation
- Debug if not working
- Submit and get AC (hopefully)

What is proving?

- Why not just implement an “obviously correct” solution?
- Often solutions base on wrong assumptions
- Both correctness and efficiency could depend on it
- So if you assume anything, it must be either written in the statement or proven
- Proving correctness of greedy algorithms and bounds on running time in general later in the course

Fixing a bug

- Say you've found a test case your program isn't working on
- An error could be on any step
- So you need to check all of them
- If you've found and fixed an error on some step, fix it and then all the following steps one by one
- Starting from the wrong step could be disastrous (bandaid)

...

```
if n == 5:
print(42)
```

...

ACM-ICPC

Contest Intro

Strategy

- Focus on algorithms and data structure
- Practice past tests and online contests
- Improve English or find a translator
- Know your notes
- As time is everything, find a way to do routine things faster/before the contest
  - Learn to use specific IDE/text editor, preferably lightweight one - Save time on creating new projects, opening new files, debugging
  - Prepare a template code with common includes and so on to not start from scratch each time
  - Backup code versions and tests
- Get your team formed and PRACTICE, PRACTICE, PRACTICE

Clarifications

- On competitions, if you don’t understand something in the statement, you could ask the jury for a clarification
- That is, send a specific question about the problem statement, assuming a Yes/No answer
- Most probably, the answer is already in the statement
- Questions must be about the problem, not your solution or other ones

References

- Official icpc site : http://cm.baylor.edu/welcome.icpc
- The Universidad de Valladolid judge https://uva.onlinejudge.org/
- How to prepare to icpc in one year : https://codeforces.com/blog/entry/47688

ACM-ICPC

Contest rules, scoring schema, team roles

ACM-ICPC

- https://icpc.global/
- Rules
  - https://icpc.global/worldfinals/rules
  - 3 members
  - Must attend university
  - Must attend if accepted
  - Transportation at charge of teams. Accommodation and food provided
  - 12 top teams awarded medals
  - All communications in English – Translator accepted provided it is a person or does not support math operations
  - Five hour contest
  - Ten or more problems
  - Languages: Java, C, C++, Kotlin and Python
  - A dictionary is allowed
- Programming environment (see https://icpc.global/worldfinals/programming-environment)
  - Each team member will be provided with a computer.
  - Reference material (JavaDoc, STL, etc.) provided on the computer
  - Chat, Screen sharing, Audio, File sharing provided
  - IDE: Eclipse, VSCode, Clion, IntelliJ, PyCharm, Code::Blocks provided

ACM-ICPC

- Rules
  - **Team Reference Document** allowed – see the On-Site Registration Instructions.
    - See for example https://cs.stanford.edu/group/acm/SLPC/notebook.pdf
    - See https://github.com/INSAlgo/ICPC-Notebook/blob/master/notebook_cpp.pdf for the more recent INSA lyon handbok. Full git repo at https://github.com/INSAlgo/ICPC-Notebook
    - Will be uploaded to the system
  - DO NOT TOUCH ANYTHING at the team workstations until so directed by the Finals Director.
  - Solutions to runs are marked run-time error, time-limit exceeded, wrong answer – see next slide for full list
    - The judge provides very little feedback
  - Scoring based on :
    - Number of problems solved - first
    - Total time to solve problem including penalty - second
    - Penalty time (20 min) per incorrect submission (only for problem solved)
  - See Programming Environment Web Site for additional details

Testing system verdicts

- ONLINE Judge verdict
  - Accepted (AC) – Congratulations!
  - Presentation Error (PE) – Check for spaces, left/right justification, line feeds, etc.
  - Accepted (PE) – Your program has a minor presentation error, but the judge is letting you off with a warning. Stop here and declare victory!
  - Wrong Answer (WA) – Your program returned an incorrect answer to one or more secret test cases.
  - Compile Error (CE) – The compiler could not figure out how to compile your program. The resulting compiler messages will be returned to you. Warning messages are ignored by the judge.
  - Runtime Error (RE) – Your program failed during execution due to a segmentation fault, floating point exception, or similar problem. Check for invalid pointer references or division by zero.
  - Submission Error (SE) – You did not correctly specify one or more of the information fields, perhaps giving an incorrect user ID or problem number.
  - Time Limit Exceeded (TL) – Your program took too much time on at least one of the test cases, so you likely have a problem with efficiency.
  - Memory Limit Exceeded (ML) – Your program tried to use more memory than the judge's default settings.
  - Output Limit Exceeded (OL) – Your program tried to print too much output, perhaps trapped in a infinite loop.
  - Restricted Function (RF) – Your source program tried to use an illegal system function such as fork() or fopen(). Behave yourself

ACM-ICPC

- ONLINE Problem
  - *Never* make an assumption which is not explicitly stated in the specs.
  - Don’t assume that the input is sorted,
  - Don’t assume that the graphs are connected,
  - Don’t assume that that the integers used in a problem are positive and reasonably small unless it says so in the specifications

Other things to work on

- Typing speed
- Learn an editor very well
- Teamwork : who does what
    - Roles: Math/**solution and proof**/algorithm/ faster coder/reviewer/**tester**
    - Weakness and strengths and complementarity
    - Strategy: Start with easy problem, don't get stuck on bugs
- Programming language review/re-learn - C,C++,Java, Python
    - If using Java be extra careful to compiler options
- Data structure, dynamic programming
- Simulate – time limit, team work, one computer
- PRACTICE, PRACTICE, PRACTICE, … the coach role is to drive the team to practice and help and support all the way through the competition.

Other competitions

- Google Code Jam / Facebook Hacker Cup
  - Individual, open for all
  - Annual, begin with Qualification Round, usually
  - in April (GCJ) and January (FHC)
  - All rounds except final are online
  - You solve the problem, request input and in
  - several minutes need to send output, so not
  - solution itself
  - Used for recruiting
- TopCoder
  - Regular rounds Single Round Matches (SRMs)
  - Rating system after each round your rating changes
  - Challenge phase you need to come up with a test to fail other people's solutions
  - Annual TopCoder Open multi-tiered championship
- Codeforces
  - Also regular rounds and rating system
  - Prize rounds with job opportunities by technological companies
  - Vibrant community many useful blog posts about competitive programming, and a place to ask for help
- Codechef regular rounds and practice problems
- Hackerrank rounds and challenges, strongly aimed to help companies in recruiting
- Sphere Online Judge vast problems archive
- CSAcademy poised for learning

IDE and setup

IDE

- Eclipse: Java, C/C++, PHP
  - https://www.eclipse.org/downloads/packages/
- Code block – **C, C++ and Fortran**
  - http://www.codeblocks.org/downloads
- Netbeans: Java, C/C++, PHP, Html 5, Groovy
  - https://netbeans.org/downloads
- Visual studio code
  - https://visualstudio.microsoft.com/downloads/
  - https://code.visualstudio.com/docs/languages/cpp
  - https://medium.com/@jerrygoyal/run-debug-intellisense-c-c-in-vscode-within-5-minutes-3ed956e059d6
  - https://www.youtube.com/watch?v=smUzCvqQKC8

VSCode with C++ Compiler

MINGW

- https://code.visualstudio.com/docs/languages/cpp
- Install mingw-get-setup.exe (only mingw32-base-bin and the C++ compiler). Add mingw bin to path.
- If using Code Runner - DO NOT FORGET --- Code Runner does not save your file before running

MSVC

- Alternatively use the Microsoft Visual C++ (MSVC) compiler toolset
  - https://code.visualstudio.com/docs/cpp/config-msvc

VS Code: Run, Debug & get Intellisense

- Install extension id: ms-vscode.cpptools-extension-pack
- Set in Preferences>settings
- Files.autosave
- editor.formatOnSave - to format when you save your file.
- editor.formatOnType - to format as you type (triggered on the ; character).

Online judge

UVA

- Register at https://onlinejudge.org/
  - Browse Problems> Programming Challenges…
- Debug: https://www.udebug.com/UVa
  - You can use the input/output to validate

First assignment : Problem read assignment

- For each problem you review, you are to produce a one-line review of the problem in a tab separated format with the following fields in order:
  - Problem ID – Give the problem id, with a '*' if the problem is likely to be one of the 1020 'best' of the problems you review. Best is your personal utility function measuring how interesting you think it is to think about, how well written it is, and the techniques required to solve it. • Problem title – Give the title of the problem, in quotes.
  - Academic level – How advanced a student do you need to be to solve this? Freshman/sophomore should be labeled 1, junior should be labeled 2, and senior-grad should be labeled 3. The issue is not so much how hard it is, but does the problem rest on elementary material or stuff you learn in higher levels? Many very interesting and challenging problems are labeled 1 if they rest on basic material in interesting ways.
  - Interestingness – How interesting will this problem be to solve? Here the scale I like is ”fun”, ”ok”, ”dull”. • Writing quality – How interesting is this problem to read? Here the scale I like is ”good”, ”average”, ”bad”. Give points for a clear description, good humor, and decent grammar. Take off points for irrelevant distractions, lame jokes, confusion, and hard-to-read writing.
  - Topics – Here give a comma separated list (with no white space for parsability) of a few relevant categories to file the problem under. The first should reflect which ”chapter/topic” of the book/course the problem should be filed under, although you can add a little more detail with other entries to hit at what the problem really is about. The chapter/topic should be drawn from the following list: iteration, data-structures, strings, sorting, arithmetic, combinatorics, number-theory, backtracking, graphs, dynamic-programming, grids, geometry, computational-geometry, other.
- Here are examples of my reviews for three problems:
  - 10053 "Envelopes" 1 ok average iteration,bin-packing
  - 10054* "The Necklace" 3 fun good graphs,eulerian-cycle,TSP
  - 10055 "Hashmat the Brave Warrior" 1 dull average arithmetic,subtraction

Problem read assignment

- From https://uva.onlinejudge.org
- Contest set:
  - volume CXII
  - volume CXIII
  - volume CXIV
  - volume CXV
  - volume CXVI
  - volume CXVII
  - volume CXVIII
  - volume CXIX
  - volume CXX
  - volume CXXI
  - volume CXXII
  - volume CXXIII

Problem read assignment

- From https://uva.onlinejudge.org
- Problem set:
  - volume IV
  - volume V
  - volume VI
  - volume VII
  - volume VIII
  - volume IX
  - volume X
  - volume XI

Getting started

I/O and simple data structure

Input/Output

C	C++

#include <stdio.h>
int main()
{
int p, q, r;
while (scanf (“%d” %d”, &p, &q)!=EOF)
{
if (p!=0 && q!=0)
{
// call a function to do something
printf (“%d\n”, r);
}
}
}
	#include <iostream>
	void main()
	{
	int p, q, r;
	while (cin >> p >> q)
	{
	if (p!=0 && q!=0)
	{
	// call a function to do something
	cout << c << endl;
	}
	}
	}

Java I/O is not trivial. Get a template done and reuse.

Programming Hints

- Write comments first
- Document each variable
- Use symbolic constants
- Use the debugger
- Use functions
- Keep it simple
- Some online judges may have issues with preprocessor directives like #ifdef. Avoid and use comment instead

Programming Hints: Input/Output

- By default, synchronizes with stdio, which makes it slow
  - turn off with
    - ios_base::sync_with_stdio(false);
- cout buffer flushes on each cin, which makes it slow
  - turn off with cin.tie(0);
- Changing synchronization
  - Interactive problems
  - Debug output — confusing when it prints not where it’s in the code
- use cout<<"\n" instead of cout <<endl as cout<<endl flushes
- Scanf and printf are fast but does no type checks
- Cin and cout can be as fast as scanf and printf if tuned properly and does type checks

Elementary data types

ATA TYPE	SIZE (IN BYTES)	RANGE
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	
double	8	
long double	12	
wchar_t	2 or 4	1 wide character

Elementary data types

- Array
  - Declaration:
    - value_type[size]
  - Fixed size, Be careful of bounds.
  - You can use sentinels

```
i = n;
a[0] = - MAXINT;
while (a[i] >= x) {
a[i] = a[i-1];
i=i-1;
}
a[i+1] = x;
```

  - Initialization
    - a[0] = - MAXINT;
    - memset(t, 0, sizeof(value_type) * size) (more efficient than a loop)
  - There is also a class array<value_type,size> in C++2011 STL which allows smoother interactions with other collections in STL.
  - Best memory occupation, O(1) to access an element, no suppression, no addition
- Multidimensional array
- Record – be careful

Getting started assignment

- Week 1 - UVA problem numbers
- 100 The 3n+1 problem (max input is 1000000 – NOT 10000) UVA
- 10189 Minesweeper UVA
- 10137 The Trip UVA
- 10033 Interpreter UVA

Testing

Testing

- Run your program locally on some inputs
- Incorrect attempts are penalized
- You need a test for debug
- In this lesson:
  - Common types of test cases
  - Testing workflow
  - Stress-testing

Types of tests - What to check

- **Correctness:** compare your output with the correct answer
  - Need to know the answer — get it manually or otherwise
- **Reliability:** make sure that your program doesn’t crash
  - Asserts help — check invariants without correct answer
- **Limits:** check working time and memory on large inputs
  - Locally — detailed information on performance

Types of tests – Sample tests

- Always are given, with the answer
- Test your understanding of the statement
- You could’ve gotten it wrong
- Test your solution before implementing
- Save time by realizing you’re wrong earlier
- Samples check general correctness and sometimes special cases
- Do not rely on samples only!

Types of tests – Minimal tests

- Test of minimal size/minimal input values
- Given: integer N ($1 \le N \le 10^6$), then a sequence of N nonnegative integers, each not greater than 10^9
  1
  0
- Often is “special”
- Easy to construct
- Something else could be minimized, e.g. answer size

Types of tests – Maximal tests

- Maximal size/maximal input values
- Given: integer N ($1 \leq N \leq 10^6$), then a sequence of N nonnegative integers, each not greater than 10^9
  1000000
  1000000000 1000000000 1000000000 ...
- Hard to compute the answer
- Checks crashes (e.g. array sizes)
- TL/ML — but max time not always on any max size test
- Integer overflow — if negative answer when should be nonnegative

Types of tests – Maximal tests

- How to obtain the maximal test
  - Generate by another program

```
1 in t n = 1000000;
2 cout << n << ’ \n ’ ;
3 for ( in t i = 0 ; i < n ; ++i ) {
4             cout << in t (1 e9 ) << ’  ’ ;
5 }
```

  - Plug in inside your code

```
1 in t n ;
2 // c i n >> n ;
3 n = 1000000;
4 for ( in t i = 0 ; i < n ; ++i ) {
5             // c i n >> a [ i ] ;
6             a [ i ] = in t (1 e9 ) ;
7 }
```

Types of tests – Maximal tests

- How to obtain the maximal test
  - Better to have special function for reading data, to replace it as a whole

```
1 v o i d  r e a d I n p ut ( ) {
2  c i n >> n ;
3  for ( in t i = 0 ; i < n ; ++i ) {
4              c i n >> a [ i ] ;
5  }
6 }

7 v o i d s et I n p ut ( ) {
8     n = 1000000;
9     for ( in t i = 0 ; i < n ; ++i ) {
10             a [ i ] = in t (1 e9 ) ;
11    }
12 }

13 in t main ( ) {
14 // r e a d I n p ut ( ) ;
15 s et I n p ut ( ) ;
16 }
```

Type of tests – Specific problem types tests

- String problems
  aaaaaa
  abcdef
- Problems about divisibility — prime numbers, numbers with many divisors
  2, 3, 11, 31, 997, $10^9 + 7$ are prime
  48 has 10 divisors, 931 170 240 has 1344 divisors
- Graphs, geometry, . . .

Type of tests – Use your program structure

- Test all branches in your code
  1 **i f** (c o n d it i o n) {
  2 . . .
  3 } **el se** {
  4 . . .
  5 }
  Include test with condition true, and condition false
- Different answer types (YES/NO, -1 for there is no answer, etc)
- Test different parts separately, each right after it’s finished

Types of Tests – Custom tests

- Make “interesting” tests — but note that they are not necessarily interesting for your solution
- Test different run patterns, special cases, pathological cases — depends on the solution and its proof
- Combine all of the above

Testing Workflow

- When to test and why
  - Before submission — to not waste attempts
  - After submission — to find a test case for debugging
- How long to test
  - Trade-of between test well and test fast
  - Depends on the rules, complexity and how sure you are of your solution
  - First test on samples to make sure the program works and the format is correct
  - Nearly always test on cases other than samples
- After fixing a bug
  - Save all tests you've come up with and run them again
  - Check on all your tests on one run

Stress testing

- You can make the computer invent tests for you!
- Write a *generator* program, which outputs some random input
- Repeatedly:
  - Generate a random input
  - Run your solution on it
  - Check if the output is correct
  - If not — stop and output the test case
- Fully automated, thousands tests per second!

Stress testing

- Use the computer to generate test randomly
  - Use parameters to easily change the test size
  - Use small test size to be sure that you have a good chance to cover the special cases
  - Do not lose generality in the generator (only ‘a’ e.g.)
  - Initialize random with the time to get different tests
- How to check correctness –
  - If detecting crashes use asserts
  - Otherwise use a trivial solution generator -> small tests

Generators

- Generate small tests — faster (esp. for trivial solution), easier debuging
- Make parameters to easily tweak test size
- Example:
  - Fails only on aaaaa, zzz, . . .
  - Random ’a’–’z’ strings of length 10: probability of $26^{-9} \simeq 2 \cdot 10^{-13}$
  - Compared to: Only ’a’, ’b’, ’c’ and string length 5: $3^{-4} \simeq 0.01$
- Do not lose generality
- Strings of ’a’ far less interesting than strings of ’a’ and ’b’
- Correctly initialize random to get different tests

Stress testing for crash

```
1 for ( ( te s t =1; ; te s t++ ) )
2 do
3       echo Test $t e st
4       . / g e n er at e > in
5       . / s o l u t i o n < in > out
6       i f [ $? -ne 0 ]
7       then
8               echo Runtime e r r o r
9               break
10      f i
11 done
```

Terminates on error, so error test is in the **in** file afterwards

Stress testing for correctness

```
1 for ( ( te s t =1; ; te s t++ ) )
2 do
3         echo Test $t e st
4         . / g e n er at e > in
5         . / s o l u t i o n < in > out
6         . / s o l u t i o n _ t r i v i a l < in > ans
7         d i f f out ans
8         i f [ $? -ne 0 ]
9         then
10                echo Wrong answer
11                break
12        f i
13 done
```

error test is in the **in** file afterwards

Stress testing workflow

- Stress-test after manual testing
- No point if generator/trivial solution/checker is too complex
- Start with very small test sizes
- Couple of minutes running is usually enough
- While running do something else useful
- If nothing is found, generate larger tests or rethink the generator

Testing - Summary

- Test your solution before and after submitting
- Start with samples
- “Interesting” manual cases — min/max, problem type specific, and anything you could imagine
- Test different parts separately
- If everything else fails, run a stress-test
- Watch out for the generator, Generate small tests

Testing assignment

- Quiz1-testing
- Getting started assignment – see slide 51-
  - Find a minimal test and explain
  - Find a maximal test and explain

Structuring the code

Structuring the code

- Why?
  - Simplifying the process of debugging.
  - Making your code more understandable.
- **Example task:** you have information about ***n*** people.
  Your goals are:
  Compute the number of people employed.
  Compute the sum of ages of all people.

Structuring the code

```
person a[n];
int employed = 0;
for (int i = 0; i < n; i++) {
        read(a[i]);
        if (a[i].isEmployed) employed++;
}
int sumAges = 0;
write(employed);
for (int i = 0; i < n; i++)
        sumAges += a[i].age;
write(sumAges);
```

Non structured

Structuring the code

```
person a[n];
int employed = 0;
int sumAges = 0;
for (int i = 0; i < n; i++)
        read(a[i]);
for (int i = 0; i < n; i++)
        if (a[i].isEmployed) employed++;
for (int i = 0; i < n; i++)
        sumAges += a[i].age;
write(employed);
write(sumAges);
```

Well structured

Structuring the code

- Code is more readable when logical blocks do not mix.
- Use meaningful variable names
- Preserve invariants
  - Compute sum of all I such that $1<=i^3<5000$
  - Pre and post condition loops
  - Invariants: $1<=i^3<5000$ and sum is correct
  - What happens to each loop type when we take instead $1<=i^3<1$

Assignment

- assignment-progCompetitions

Data structures

Arrays

- Array
  - Size is fixed
  - Could take/set an element by index
  - This operation is really fast
- Dynamic array (vector/list)
  - Same as a usual array
  - Size could be changed
  - Could take twice as much space, as the total size of elements

String

- Array of characters + useful tools
- Concatenate, extract/find substring
- Split, trim (strip)
- Convert to/from numbers
- Regular expressions

Array-like structures

- Bitset — an array of bits
  - Each bit takes a bit in memory, not a byte as in an array of booleans
  - Bits are addressed in an array of integers, as if they are concatenated
  - Could count ones, do bitwise and, or, xor, etc — in about n/32 int operations
- Big integers — arbitrary-size integer numbers
- Big decimals — arbitrary-precision floating point numbers

Queues

- Queue
  - Push to the back
  - Take from the front
- Stack
  - Push to the front
  - Take from the front
- Deque
  - Push to the front/back
  - Take from the front/back
  - Could be used as a queue/stack

Queues

- First in, first out
- When to use: when order matters
  - Example deck of cards
- Key operations
  - *Enqueue(x,q)* — Insert item *x* at the back of queue *q*.
  - *Dequeue(q)* — Return (and remove) the front item from queue *q*
  - *Initialize(q), Full(q), Empty(q)* — Analogous to these operation on stacks.
  - No search

Priority Queues

- First in, out based on priority
- When to use: when order matters
  - Example: maintain schedules and calendars
- Key operations
  - *Insert(x,p)* — Insert item *x* into priority queue *p*.
  - *Maximum(p)* — Return the item with the largest key in priority queue *p*.
  - *ExtractMax(p)* — Return and remove the item with the largest key in *p*.

Stacks

- Last in, first out
- When to use: when order does not matter
  - Example dinner plates stack
- Key operations
  - *Push(x,s)* — Insert item *x* at the top of stack *s*.
  - *Pop(s)* — Return (and remove) the top item of stack *s*.
  - *Initialize(s)* — Create an empty stack.
  - *Full(s), Empty(s)* — Test whether the stack can accept more pushes or pops, respectively.
  - No search

(Sub)Sets

- Insert an element
- Check if some value is contained
- Ordered set — could binary search for a value “find the greatest value less than 10^9 in the set”
- Unordered set (hash set): O(1) per operation — hash table
- Ordered set (tree set): O(log n) per operation — binary search tree
- On practice, ordered set is only slightly slower
- In both, much slower operations than in an array and more space per element

(Sub)Sets

- Unordered collections of elements drawn from a given universal set *U*. Need to know which elements from *U* are *not* in the given subset.
- When to use: when checking if some value is contained
  - Example:
- Key operations
  - *Member(x,S)* — Is an item *x* an element of subset *S*?
  - *Union(A,B)* — Construct subset $A \cup B$ of all elements in subset *A* or in subset *B*.
  - *Intersection(A,B)* — Construct subset $A \cap B$ of all elements in subset *A* and in subset *B*.
  - *Insert(x,S), Delete(x,S)* — Insert/delete element *x* into/from subset *S*.
- Unbounded U -> sets = dictionaries

Dictionaries (Map)

- Associative array: any type could be used for keys
- Space proportional to the number of elements
- Just a set of key-value pairs
- Unordered (hash map) and ordered (tree map)

Dictionaries (Map)

- Retrieve data based on content not position
- When to use: when content matters
  - Example: Hashmap
- Key operations
  - *Insert(x,d)* — Insert item *x* into dictionary *d*.
  - *Delete(x,d)* — Remove item *x* (or the item pointed to by *x*) from dictionary *d*.
  - *Search(k,d)* — Return an item with key *k* if one exists in dictionary *d*.
- Static dictionary – Get built once and do not change -> array
- Semi-dynamic dictionary – Insertion and search but no deletion -> Hash table with a large enough array to handle collisions
- Fully dynamic dictionary – Insertion, search and deletion -> Hash table with chained list

Algorithms

- Sort — O(n log n)
  - Stability — order on equals
- Binary search — O(log n)
- Random numbers generator
  - Start with some seed
  - Generate next “random” number
  - The sequence depends only on the seed
  - Random integer in [l, r)
  - Shuffle a sequence

C++ Standard Template Library

- References
  - Topcoder: Power up C++ with the Standard Template Library: Part 1 and Part 2
  - **YoLinux: C++ STL Tutorial and Books at the end: http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html**
  - https://www.geeksforgeeks.org/the-c-standard-template-library-stl/
  - **https://www.hackerearth.com/practice/notes/standard-template-library/**
- Structures
  - *Stack* — Methods include S.push(), S.top(), S.pop(), and S.empty(). Top returns but does not remove the element on top; while pop removes but does not return the element. Thus always top on pop [Seu63]. Linked implementations ensure the stack will never be full.
  - *Queue* — Methods include Q.front(), Q.back(), Q.push(), Q.pop(), and Q.empty() and have the same idiosyncrasies as stack.
  - *Dictionaries* — STL contains a wide variety of containers, including hash map, a hashed associative container binding keys to data items. Methods include H.erase(), H.find(), and H.insert().
  - *Priority Queues* — Declared priority queue<int> Q;, methods include Q.top(), Q.push(), Q.pop(), and Q.empty().
  - *Sets* — Sets are represented as sorted associative containers, declared set<key, comparison> S;. Set algorithms include set union and set intersection, as well as other standard set operators.

C++ Standard Template Library

- When the program is using STL, it should #include the appropriate standard headers. For most containers the title of standard header matches the name of the container, and no extension is required e.g. *#include <stack>*
- Always add "*using namespace std;*"
- When making nested constructions, make sure that the "brackets" are not directly following one another – leave a blank between them e.g. *vector<vector<int> >* not *vector<vector<int>>*
- You should remember one more very important thing: When vector is passed as a parameter to some function, a copy of vector is actually created so use

```
void some_function(vector< int >& v) { // OK
// ...
}
```

C++ Standard Template Library

- The type of iterator can be constructed from a type of container by appending “::iterator”, “::const_iterator”, “::reverse_iterator” or “::const_reverse_iterator” to it.
- Iterate on a collection (C++ 2011)
  - *for(auto t : collection) {*
    / do something with t */*
    }
  - Use *auto & to modify the collection*
    vector<int> v = {1, 2, 3};
    for(auto& t : v) {
    t++;
    }
    // will display 2, 3, et 4
    for(auto t : v) {
    printf("%d\n", t);
    }
  - *unordered_map<int, set<int>>::iterator itr;*
    itr = colouradj.begin();
    while (itr != colouradj.end())
    {
    ++itr;
    }

C++ Standard Template Library

- Iteration algorithm
  - *reverse (v.begin(), v.end())*
  - *If (find (v.begin(), v.end(),49) != v.end())*
  - *Get the index of an element int i = find (v.begin(), v.end(),49) – v begin()*
  - *int v1 = *max_element(X.begin(), X.end()); // Returns value of max element in vector*
  - *int i1 = min_element(X.begin(), X.end()) – X.begin; // Returns index of min element in vector*
  - *// Shift all elements from second to last to the appropriate number of elements.*
    // Then copy the contents of v2 into v.
    v.insert(1, all(v2));
  - *sort(X.begin(), X.end()); // Sort array in ascending order*
    sort(X.rbegin(), X.rend()); // Sort array in descending order using with reverse iterators

C++ Standard Template Library

- Variable arrays
- Vector
  - Declaration
    - *vector<value_type>*
    - *vector< string > names(20, “Unknown”);*
  - Take up to twice the memory space of a fixed array
  - O(1) to access any element, Addition/Suppression at the end O(1) amortized, Addition/Suppression at a different place O(n).
  - Used to index elements by integer when there is no (or few) gaps in the indexing integers.
- Deque
  - Same as vector but Addition and suppression in O(1) at beginning and end and non contiguous in memory

C++ Standard Template Library

- Lists
- Double linked list
  - Declaration
    - list<value_type>
    - Access to last or first element in O(1), Access to previous or next element in O(1). Access to an arbitrary element in O(n), Addition/ suppression at an arbitrary location O(1)
- Forward linked list
  - forward_list<value_type>

C++ Standard Template Library

- Pairs
  - Declaration
    - pair<type1, type2>,
  - Initialization
    - make_pair(a, b);
  - Access
    - p.first and p.second
  - Operateur < on pairs uses lexicographic order. To overload, define for example
    using namespace std;
    typedef pair<int, int> pii;
    *bool operator< (pii a, pii b) { return a.first * b.second - b.first * a.second; }*

C++ Standard Template Library

- Sets
  - Declaration
    - Set<value_type>
  - Duplicate elements are discarded. Order of element in set does not matter
  - Elements are iterated over in natural order or as defined by the overloaded < operator (see previous slide).
  - Access to an element in O(log(n)), addition/suppression of an element in O(log(n))
  - multiset<value_type> allows to have same element multiple times

C++ Standard Template Library

- Unordered Sets
  - Declaration
    - Unordered_set<value_type>
  - Duplicate elements are discarded. Order of element in set does not matter
  - Elements are iterated over in arbitrary order (the hash function order).
  - Access to an element in O(1) amortized, addition/suppression of an element in O(1) amortized
  - Unordered_multiset<value_type> allows to have the same element multiple times

C++ Standard Template Library

- Stack
  - Last in First out
  - Declaration
    - stack<value_type>
  - Access to the first element in O(1), addition/suppression at the beginning in O(1)
  - Access only to first element, insertion only at the beginning

C++ Standard Template Library

- Queue
  - First in First out
  - Declaration
    - queue<value_type>
  - Access to the first element in O(1), suppression at the beginning in O(1), addition at the end O(1)
  - Access only to first element, suppression only at beginning, insertion only at the end

C++ Standard Template Library

- Priority Queue
  - Elements are inserted and kept by natural order or by an order of priority defined by a comparator. Two elements can have the same priority
  - Declaration
    - Priority_queue<value_type>
  - Access to the highest priority element in O(1), suppression of the highest priority element in O(log(n)), addition of an element O(log(n))
  - Access to other positions impossible, modification of priorities impossible

C++ Standard Template Library

- Map
  - Store key – values pairs
  - Declaration
    - map<key_type, value_type>
  - Elements are iterated over in natural order or as defined by the overloaded < operator (see previous slide).
  - Access to any element by key in O(log(n)), Addition/suppression of any element by key in O(log(n)),
  - multimap<key_type, value_type> : allows many values for a key

C++ Standard Template Library

- Unordered map
  - Declaration
    - unordered_map<key_type, value_type>
  - Elements are iterated over in arbitrary order (the hash function order).
  - Access to any element by key in O(1) amortized, Addition/suppression of any element by key in O(1) amortized,
  - unordered_multimap<key_type, value_type>: allows many values for a key

How to debug STL classes

- Look inside the _M_impl member
- For vector e.g Myvector._M_impl._M_start[i] gives the i^{th} element

Data structure assignment

- 12269 Lawn mower ACM or UVA
- 3359 - That is your queue ACM or UVA
- 1120 - No change ACM or UVA

Languages

C++

Strings

- C way: arrays of char
- C++ way: string
- Fixed-size vs dynamic
- Functions (like strcmp, strcat) vs members
- Slightly faster vs convenient
- string is used more often

Programming Hints: Input/Output

- By default, synchronizes with stdio, which makes it slow
  - turn off with
    - ios_base::sync_with_stdio(false);
- cout buffer flushes on each cin, which makes it slow
  - turn off with cin.tie(0);
- Changing synchronization
  - Interactive problems
  - Debug output — confusing when it prints not where it’s in the code
- use cout<<"\n" instead of cout <<endl as cout<<endl flushes
- Scanf and printf are fast but does no type checks
- Cin and cout can be as fast as scanf and printf if tuned properly and does type checks

Undefined behavior (UB)

- Compiler may do anything: your program could crash, or work incorrectly, or even correctly. It could depend on compiler version, flags, memory before running
- Know common UBs
- Some may be detected by compiler warnings:
  - turn on as much as possible
  - g++ -Wall -Wextra ...
- Platform-dependent flags
- Linking libs with pedantic implementations: e.g. std::vector which always checks indices
  - -pedantic-errors

Undefined behavior

- Incorrect array indices: negative or too large
  - int a[10];
  - a[-1] = a[10] = 0;
- Using local variables before assignment
  - int a;
  - a++;
  - cout << a;
- Non-void functions without return
- Signed integer overflow

Other remarks

- Segmentation fault — use a debugger to find the exact place
- Compilation errors — always start from the first
- using namespace std; — shortens code, but occupies variable names (potential conflict with variables in std space)
- Compiler differences — perfomance, variable size (long double), scanf/printf templates
- Assigning/passing structures — O(n) time!
  - Use pointers or references where needed
- C++11 features – be aware when using them
  - unordered_set
  - vector<int> a = {1, 2, 3};
  - for (auto x : a)

Java

Input/Output

- Scanner is convenient, but very slow
  - Only small inputs, less than 10 000 integers
- BufferedReader is fast, but it only reads whole lines
- Strategy
  - Pass lines to a StringTokenizer
  - Parse numbers from tokens by e.g. Integer.parseInt
  - Put it in a separate class with Scanner-like methods, e.g. nextInt()
  - Include in your template code, to not write every time
- For output, PrintWriter is fine

Collections

- Collections always store objects, not primitives
- Could use a primitive wrapper like Integer. But Integer takes 16 bytes, not 4!
- Object overhead with collections
  - ArrayList<Integer> — much worse than int[]
- Collections: unsynchronised and synchronised
  - ArrayList vs Vector
  - Use unsynchronised — optimised for a single thread

Strings

- String — immutable
- Every operation produces a new object, so most are linear
- s += ’a’ is also O(n)!
- StringBuilder — special class for growing strings
  - append method — O(1)
  - Elements are char — no object overhead

Other remarks

- Size of any object — at least 8 bytes more than the size of fields
- Collections.sort — merge sort: stable, always O(n log n)
- Arrays.sort — a version of quick sort:
  - unstable, faster on average, but could take n^2 on specific tests!
  - Shuffle the array before sorting
- Do not forget to clone

Python

Speed up

- Local variables are faster than global
  - Local — list, global — dict
- Put global code in a separate function, to not use global variables

```
def main():
    # write global code here
main()
```

- Appends with + create new object, so linear time
  - s = s + ’a’ + ’b’
  - a = a + [0]
  - Use += or append
- Instead of input and print use file I/O — like read or write
- Read and write all at once
  - sys.stdin.read()
  - sys.stdin.readlines()
  - sys.stdout.write(’ ’.join(map(str, a)))

Lists

- A lot of useful tools for lists: standard functions like sum, min, join and the module itertools
- Not only shorten the code, but are also faster than *for*:
  - s = sum(a)

  vs
  - s = 0
  - for x in a:
  - s += x

  Additions are performed inside the C code of sum!

Other remarks

- Different versions of Python: Python 2 and Python 3
  - In Python 2, range(n) creates a list, and xrange(n) — a generator, which is much faster
  - In Python 3, range(n) — a generator
- On average, Python 2 is slightly faster
- PyPy — another interpretator — could be faster, especially PyPy 2
- Max depth of recursion is 1000 by default
  - Use **sys.setrecursionlimit** to increase

Other remarks

- eval and exec help in some implementation problems
- No compiler — no prior checks
  - Test solutions even more carefully
- No compile errors with compiler’s message
  - Everything — a runtime error
- Do not forget to clone
  - b = a[:] for lists
  - [[]] * n — beware - all sublists are the same one!
  - [[] for i in range(n)] — correct

Comparing languages

C++	Java	Python
• Most popular language on competitions • Very fast, decent standard library • Undefined behavior situations and uninformative crashes may be hard to debug	• Still fast enough — only 1.5-2 times slower than C++, on average • Standard library in some cases overpowers that of C++, e.g. BigInteger • Need to implement fast reading • More checks and limitations than in C++ • Informative RuntimeException • Codes are longer	• 10–100 times slower than C++ — some problems could not be solved at all • Standard library lacks sorted set and bitset • More high-level, programs are shorter • Useful where C++ is too cumbersome, or big integers are needed • Implementation/math problems

Brute force/Backtracking

Agenda

- Intuitive solution
- Search space
- Backtracking
- Pruning

Intuitive Solution

Introduction

- Sometimes the first solution you come up with is the correct one.
- But sometimes it’s not.
- In this lesson we are going to develop a method for designing solutions which are always correct
- BUT they are going to be slow

Example 1: Digits ordering

- Largest number
  Input: list of digits.
  Output: the largest number that can be made of the digits.
  Sample input: 3,7,5
  Sample output: 753.
  The solution is to order the digits from the biggest one to the smallest one.

Example 2: Robber's problem (aka knapsack problem)

- Robber's problem
  - Input: a list of items with weights (kg) and costs ($) as well as the capacity of a bag (kg).
  - Output: the maximum total cost of items that fit in the bag

	Capacity/Weight (v)	Cost/Value (c)
Bag	5	
Item1	3	2
Item2	2	3
Item3	5	6

Intuitive Solution - Tempting approach

- It’s natural to process items in order of decreasing $ per kg.
- Let’s calculate utility (cost/ weight) for each item.
- The better the utility the better the item.
- Therefore we should try to put items with maximum utility first.
- Nice and easy. But, unfortunately, wrong.

Intuitive Solution - Tempting approach

	Capacity/Weight (v)	Cost/Value (c)	Utility	Selected
Bag	5			
Item1	3	2	2/3	Y
Item2	2	3	3/2	Y
Item3	5	6	6/5	N

- Total cost = 5
- But we could do better with the third item only for a total cost of 6.

How to prove?

- Thus, the initial intuitive idea turned out to be wrong.
- Then how to convince yourself that your approach is correct?
- The simplest thing to do is to check your algorithm with pen and paper against sample tests.
- But what to do if your solution got “wrong answer” verdict from the judge?
- It’d be good to have a solution which is always conceptually correct.
- And that’s what we’ll do!

Search Space

First concept: Search Space

- Almost every combinatorial problem falls in one of the following categories
  1. Find an element of a set *A* satisfying some property (or a number of such elements).
  2. Find an element of a set *A* such that some objective function is minimized/maximized.
- We will call the set *A* search space.

Example 1

- Superstring
  - Given *m* strings *s*1, …, *sm* consisting of letters “a” and “b” only and an integer *n*. Find a string *s* of length *n* containing each *si* (for all $1 \le i \le m$) as a substring.
- Sample
  Input: *m* = 2; *n* = 3; *s*1 = ab, *s*2 = ba
  Output: aba (aba, aba) (another valid output is bab)

Example 1 Search Space

- One way to solve a problem is to simply go through all possible candidate solutions. For the superstring problem, the search space consists of all strings of length *n* over the alphabet *{a; b}*. For each such string, we check whether it is indeed a superstring of *s*1*, …,sm*
- Let's consider another testcase: *n* = 4, *s*1 = bab, *s*2 = abb.
- There are only $2^4 = 16$ strings of four letters "a" and "b".

Example 1 Search Space

Candidate	bab	abb	Candidate	bab	abb
aaaa	×	×	baaa	×	×
aaab	×	×	baab	×	×
aaba	×	×	baba	baba	×
aabb	×	aabb	**babb**	babb	babb
abaa	×	×	bbaa	×	×
abab	abab	×	bbab	bbab	×
abba	×	×	abba	×	×
abbb	×	abbb	bbbb	×	×

Example 2

- Maximum subarray problem
  - Given an array $a1, \ldots, an$. What is the largest possible sum $al + al+1 + : : : + ar-1 + ar$ for $1 \le l \le r \le n$?
    Note that ai could be negative.
  - Sample
    Input: a = (4; 1; -2; 3; -10; 5)
    Output: The best subarray is (4; 1; -2; 3; -10; 5) and the sum is 4 + 1 + (-2) + 3 = 6, so l=1, r=4

Example 2 search space

- The Search space for the maximum subarray problem is the set of all subarrays of the array *a*.
- A Subarray is determined by its first and last elements: *l* and *r*.
- For this problem, understanding what the search space is instantly provides us with the solution.

Example 2 solution

- Enumerate all pairs ($l; r$) such that $l \leq r$, for each pair compute the sum $a_l + \cdots + a_r$, and take the maximum.

Example 3

- Robber's problem
  - You have a knapsack of volume *V* and *n* items of volumes *v*1, …,*vn* and costs *c*1, …,*cn*. What is the largest total cost of the set of items whose total weight does not exceed *W* ?
  - Input: *V* = 5; *n* = 3
    *v*1 = 3 *v*2 = 2 *v*3 = 5
    *c*1 = 2 *c*2 = 3 *c*3 = 6
  - Output: The best solution is to put the last item to the knapsack and get the total cost 6.

	Capacity/Weight (v)	Cost/Value (c)	Utility	Selected
Bag	5			
Item1	3	2	2/3	N
Item2	2	3	3/2	N
Item3	5	6	6/5	Y

Example 3 search space

- What is the search space for the robber's problem?
- It's all sets of items.
- For the given example, possible sets are the following:
  ø; {1}; {2}; {3}; {1; 2}; {1; 3}; {2; 3}; {1; 2; 3}.
- Not all of these sets fit into the backpack, but it's easy to check: compute the total weight of the set and check whether it exceeds the capacity of the backpack.

Search space: summary

Problem	Search space
Superstring	strings consisting of letters “a” and “b”
Robber’s problem	all possible sets of items
Maximum subarray	pairs (l; r) such that $l \leq r$

Backtracking

Exploring the search space

- For the maximum subarray problem the search space gives us the solution instantly.
  - We can try all possible pairs with two nested for cycles.
- For the substring problem we want to try all possible strings of *n* symbols.
  - It'd be good to have *n* nested for cycles iterating through letters "a" and "b".
  - Your pseudo-Python code will look like this for the superstring problem:

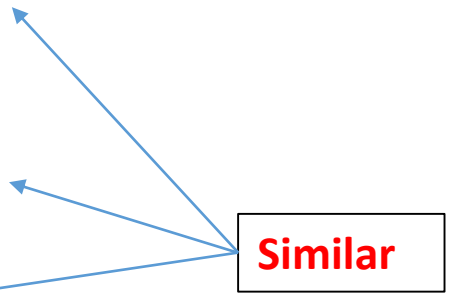
```
for x [ 0 ] in [ ' a ' , ' b ' ] :
    for x [ 1 ] in [ ' a ' , ' b ' ] :
            # . . .
                    for x [ n-1] in [ ' a ' , ' b ' ] :
                            # check i f x c o nt a i n s
                            # a l l s t r i n g s s [ 1 ] , . . . s [m]
```

  - But there is no way to right *n* nested for loops if *n* is not a fixed predefined number.

Second concept: Backtracking

- Backtracking is roughly the way how to write *n* nested for cycles.
- The simplest possible way to simulate this “code” with an actual code is via recursion.
- Let’s first do it for three nested for cycles.

```
def threeFors (n, x):
        for x [0] in [’a’, ’b’]:
                twoFors (n, x)
def twoFors (n, x):
        for x [1] in [’a’, ’b’]:
                oneFors (n, x)
def oneFors (n, x):
        for x [2] in [’a’, ’b’]:
                print (n, x)
```




```
nestedFors (n, x, counter)
Counter = 0 -> threeFors
Counter = 1-> twoFors
Counter = 2-> oneFors

def nestedFors(n, counter, x)
        if counter =n-1
                for x[counter] in [’a’, ’b’]:
                        print (n,x)
        else
                for x[counter] in [’a’, ’b’]:
                        nestedFors (n, counter+1, x)

nestedFors (3,0,x);
```

Second concept: Backtracking

- The last loop can further be factored in the code as follows.

```
def nestedFors(n, counter, x)
        if counter =n-1
                for x[counter] in ['a', 'b']:
                        print (n,x)
        else
                for x[counter] in ['a', 'b']:
                        nestedFors (n, counter+1, x)

nestedFors (3,0,x);
```

```
def nestedFors(n, counter, x)
        if counter =n:
                print (n,x)
        else:
                for x[counter] in ['a', 'b']:
                        nestedFors (n, counter+1, x)

nestedFors (3,0,x);
```

Alternative

```
def nestedFors (n, firstFor, x):
        if firstFor < n:
                for x[firstFor] in ['a', 'b']:
                        nestedFors (n, firstFor + 1, x)
        else:
                print (x)
```

Example 1: Robber's problem

- Search space for the robber's problem is **the set of all sets of items.**
- How to enumerate all sets of n items?
- Basically, it is the same as enumerating all strings over {0, 1} of length *n.*

Recall our example:
$n = 3$; $W = 5$ and
$w_1 = 3$ $w_2 = 2$ $w_3 = 5$
$c_1 = 2$ $c_2 = 3$ $c_3 = 6$

Items 1 2 3	**Set**	**Weight**	**Cost**
0 0 0	Ø	0	0
1 0 0	{1}	3 2	
0 1 0	{2}	2 3	
1 1 0	{1, 2}	2+3=5	2+3=5
0 0 1	{3}	5 6	
1 0 1	{1, 3}	3+5=8	2+6=8
0 1 1	{2, 3}	2+5=7	3+6=8
1 1 1	{1, 2, 3}	2+3+5=10	3+2+6=11

Therefore we reduced robber's problem to the same *n* nested for cycles!

Pruning

Pruning

- Cut out branches not leading to a solution as soon as we know about them
  - For example the solution so far is above the optimum

```
def nestedFors(n, counter, optimum, x)
        cost=evaluate (n,x);
        if cost > optimum //pruning
                return;
        else:
                if counter =n:
                        print (n,x)
                else:
                        for x[counter] in [’a’, ’b’]:
                                nestedFors (n, counter+1, x)

nestedFors (3,0,x);
```

Brute force/Backtracking summary

- Designing a brute force solution:

1. Identify the search space.
2. Design a way of enumerating all its elements.
3. Turn it into a solution.

The resulting solution is usually slow, but it is clearly correct and can be used for debugging.

Some optimization can be achieved by pruning if possible

Brute force/Backtracking assignment

- assignment-correctnessFirst

Complexity and running time

Introduction

- Time limits are tight.
- The whole point – to do the thing quickly.
- Measure quickness.
- Predict before implementing.
- Make solutions faster.

Example

Substring problem

Given two strings *s* and *t* check if *s* is a substring of *t*.

Input:	***s*** = abac; ***t*** = abacabad
Output: Yes: abacabad	
Input:	***s*** = cac; ***t*** = abacabad
Output: No	
Input:	***s*** = abab; ***t*** = abacabab
Output: Yes: abac**abab**	

Algorithm

- ***n***: = **length(*s*)**
  m: **= length(*t*)**
  For all substrings of ***t*** of length ***n***:
  - Compare characters of ***s*** and this substring one by one.
  - If there is a mismatch, move on to the next substring.
  - If all characters are equal, return Yes.
  - If none of substrings matches, return No.

Examples

***s* = abab; *t* = abacabad;**

a	b	a	b				
a	b	a	c	a	b	a	d

a	b	a	c	a	b	a	b
				a	b	a	d

Cost= 4 + 4 + 4 + 4 + 4 = 20

***s* = abac; *t* = abacabad;**

a	b	a	c				
a	b	a	c	a	b	a	d

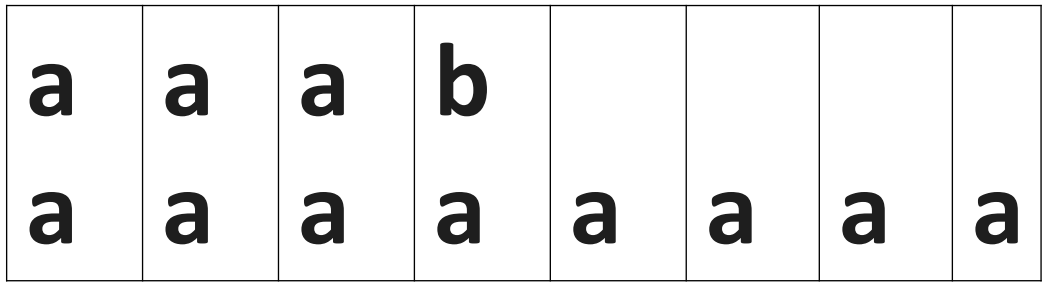
Cost= 4

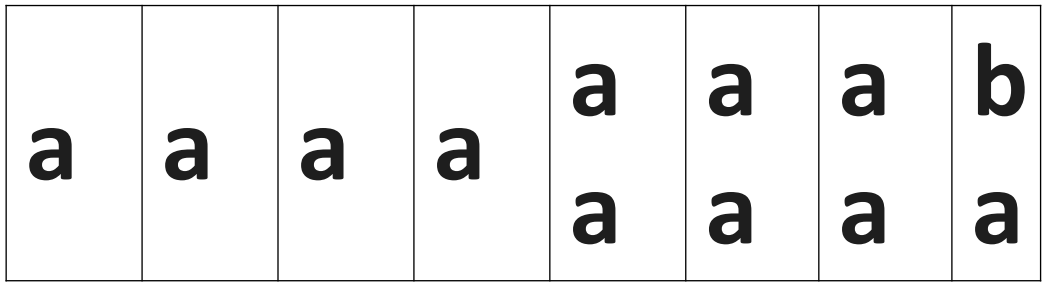
Operations could vary

- The number of operations could be different.
- If your program is fast on the samples or even on some custom tests, that doesn't mean it'll ***always*** be this way.
- Your program should work quickly on the ***worst*** possible test.
- The worst possible test for our previous algorithm:
  – the answer is "No" – we will check every substring;
  – on every substring we will compare characters until the last.

Example of worst case

***s* = aaab; *t* = aaaaaaaa;**





Cost= 4 + 4 + 4 + 4 + 4 = 20

Observations

- The worst test is not just any big enough test.
- It could be hard to construct it.
- Goal – to estimate the number of operations on any test without finding the worst possible test explicitly.

Which operations to count and how?

Unit operations

We'll count operations taking some small fixed amount of time:

- number operations (+, -, *, /, %, <, >, =);
- logical operations (or, and, not, xor);
- accessing a value from an array;
- defining a new variable

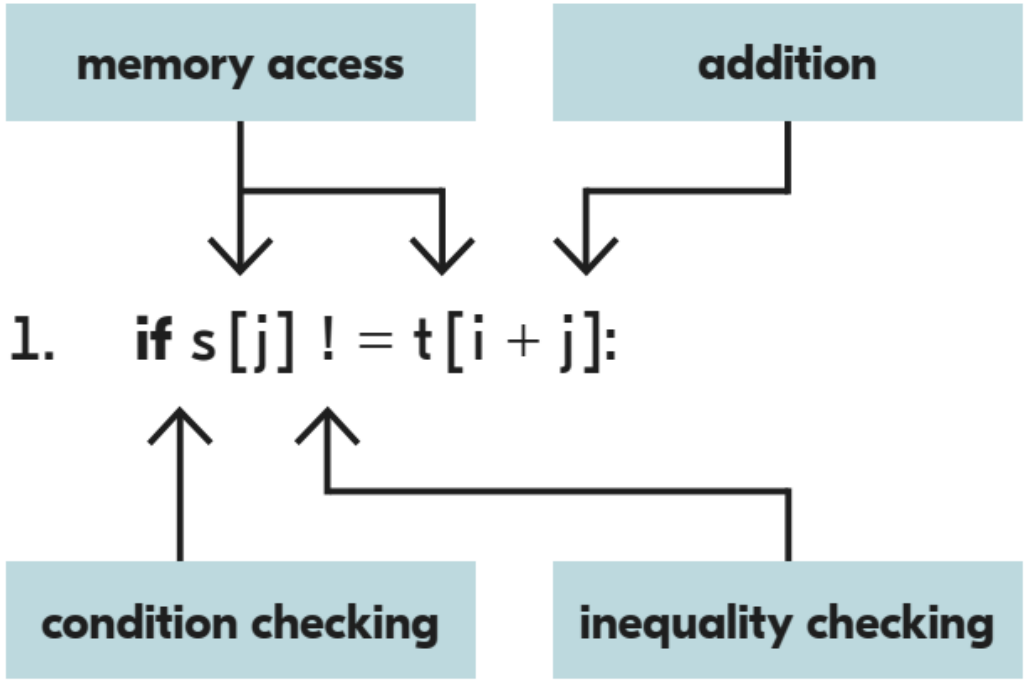
Non unit operations

Some operations take more time:

- comparing strings or lists;
- defining a string or a list with many elements;
- concatenating two strings.

Strings and lists consist of small elements.
The operations are applied to each element.
So if there are many of them, it could take much time

Simplify the count



- Tedious to count all operations.
- The number of operations in that line is ***independent*** of the input.
- A constant number of operations – no need to count explicitly.

We can drop constants.

Example 1: Substring problem

```
1. for i in range (m − n + 1):          ← no more than m times
2.     match = True
3.     for j in range(n):               ← n times
4.         if s[j] ! = t[i + j]:
5.             match = False
6.             break
7.     if match:
8.         break
```

constant

We can conclude that :

- The algorithm does no more than $m \cdot n \cdot \text{constant}$ operations, without checking particular tests!

First concept: Big-O notation

- **Upper** bounds up to a ***constant*** multiplier.
- Constants are similar for different solutions, but what matters is the dependence on input.
- n – some input parameter, and $f(n)$ – some function of n.
- An algorithm has asymptotic time complexity of $O(f(n))$ if it does no more than $C \cdot f(n)$ operations on any input, where C is some constant number.
- Could be several parameters.
  Our superstring algorithm is $O(m \cdot n)$.

Properties of O

- Upper bounds: $\boldsymbol{O(\ldots)}$ if ≤ constant x ... operations.
- Could be less!
  May be $\boldsymbol{O(n^2)}$, but also $\boldsymbol{O(n)}$.
  If $\boldsymbol{O(n)}$, then $\boldsymbol{O(n^2)}$, as $\boldsymbol{n \leq n^2}$.
- Optimal bounds may be very non-trivial.
- But we could get some simple bounds.

O for some operations

- Unit operations – ***O* (1)**.
- Built-in functions/structures need to know in advance.
  Comparing strings – ***O* (size)**.
  Requires passing through elements – at least size operations.
- Own function – bound separately.

Loops

- **for** i **in range**(n):
  $O(f(n))$
- Inside part of loop $O(f(n))$ on each iteration.
- $O(n \cdot f(n) + n)$ in total as iterating is constant $\cdot\, n$ by itself.

Recursion

- Enumerating all strings ***x*** over **{*a*, *b*}** of length ***n***:

```
1. for x[0] in ['a', 'b'] :
2.     for x[1] in ['a', 'b'] :
3.             ...
4.             for x[n - 1] in ['a', 'b']:
5.                     print(x)
```

- ***n*** nested for loops, each runs over 2 letters.
- So $\mathbf{2 \cdot 2 \cdot \cdots 2 = 2^n}$ iterations in total. print **(*x*)** outputs every element of ***x***, length is ***n***, so it's $\boldsymbol{O(n)}$ by itself.
- Overall, $\boldsymbol{O(n \cdot 2^n)}$.

Approach to solving a problem

1. Invent a solution.
2. Check if it's correct.
3. Get ***O*** **(**...**)** bound – could be done without implementing!
4. Check if it's fast enough.
5. If not, invent another or get a better bound.

Is it fast enough?

- ***O*** **(...)** operations – some function of input variables like ***O*** **(** n^3 **)** or ***O*** **(** $n \cdot m$ **)**.
- These values are bound by the statement – plug the limits in your ***O*** estimate.
  - ***O*** **(** n^3 **)**, $n \leq 100$: $100^3 = 10^6$.
    O **(** $n \cdot m$ **)**, $n \leq 10^4$, $m \leq 10^6$: $10^4 \cdot 10^6 = 1010$.
  - Compare with how many operations could be done in a second.
    - Expected to be 10^8–10^9 unit (simple) operations, in C++ or Java.
    - Less for Python, about 10^7.
    - 10^6 – will pass even with quite big constant.
    - 10^{10} – won't pass.
    - What if we're somewhere in between?

When you need to make a decision

- ***O* (*n*)** is better than ***O* (n^2)**.
  But if it's **$10^6 \cdot n$** vs **$10 \cdot n^2$** and **$n \leq 100$**?
- Multiply by large factors even when formally constants.

```
1. for i in range (n):
2.     for c in 'a' .. 'z':
3.             some thing in O(1)
```

- Formally ***O* (*n*)** – as the second loop doesn't depend on input.
- But when estimating operations, use **26 · n** instead of just ***n***

When you need to make a decision

- Operations differ

Light:

- +, −
- logical
- *

Heavy:

- %
- appending to strings/lists
- recursion
- math functions like sqrt
- I/O

When you need to make a decision

- When you've got the number of operations under ***O*** and still in doubt.
  - Think about what constant will it be multiplied by.
    - Few light operations per one cycle – larger bound (10^8 or 10^9) is still fine.
    - Many and/or heavy per one cycle – you need a smaller bound, like 10^7, it could also TL.
    - You should consider **only** frequent operations. Sqrt is heavier than + but if you have 1 of sqrt and 10^6 of +, sqrt doesn't matter but + does.
- If still too slow try to improve on the bottleneck

Finding the bottleneck locally

- 1. **for** i **in range**(n):
  2. ...
  3. **for** j **in range**(m):
  4. ...
  5. doSomething()
  6. ...
  7. ...

- Our contribution: ***O*** **(*n*** · ***m*** ·**time(*doSomething*))**
- May be a bottleneck: if the program is overall ***O*** **(*n*²** · ***m*****)** and **time(*doSomething*) = *O* (*n*)**, it contributes ***O*** **(*n*** · ***m*** · ***n*****) = *O* (*n*²** · ***m*****)**. So up to a constant, this line has as much operations, as the entire program. If you want faster solution, you need to optimize that line doSomething().
- Or not: if the program is overall ***O*** **(*n*³** · ***m*****)**, then no sense making doSomething faster as it contributes only ***O*** **(*n*²** · ***m*****)** – ***n*** times smaller than something else. Find that something else and optimize it.

Making a solution faster - summary

- Your solution is too slow.
- First, try to improve asymptotically (reduce O(…))
  - Only in bottleneck parts.
- If you couldn’t get better asymptotically and your solution is just above the TL, try to optimize constants, but only **in this case**.
  - Get rid of heavy operations.
    - AND especially of large debug output.
  - Avoid recomputing.

Measure locally

- How many times a function is called:
  1. **def** someFunction():
  2. counter + = 1
  3. ...
- Whole program
  time [command] – UNIX-like systems.
- See how much time has elapsed inside the program:
  1. start = getTime( )
  2. ...
  3. **print**(getTime( ) - start)
  Could measure the whole program, or just some parts, and see how much do they actually contribute

Memory

- Aside from time, your program should also fit in the memory limit.
- But it's usually weaker than TL. Too much appends to lists nearly always TL, not ML.
- The most common cause of ML – large arrays.
  But their size is easy to calculate explicitly.
  Only need to know sizes of variables.

Summary

- Your program is expected to work fast on worst-case inputs.
- You should always get ***O*** bound before implementing.
- To check, plug limits in the bound and compare with possible number of operations in one second.
- Speed up only in bottlenecks.
- First optimize asymptotically. Only if this fails and you need to optimize constants.
- Could be useful to measure actual time.

Complexity assignment

- Quiz2-timeComplexity

Handling numbers

Agenda

- Integer types and overflow
- Dealing with overflow
- Non integers
- Fixed point numbers and errors
- Floating point numbers
- Where and how to use doubles
- More on floating point

Integer types and overflow

Integer Overflow

- int a = 50000;
  int b = 50000;
  cout << a * b;

- What is the result?

- -1794967296

Numbers inside a computer

- Memory — a sequence of binary digits — bits
  01011110100100111000100001111010. . .
- A number must come in binary
  $13 = 1101_2 = 1 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 + 1 \cdot 8$
- Bits come in chunks of 8 — bytes
  01011110 10010011 10001000 01111010 . . .
  int a = 13;
  00000000 00000000 00000000 00001101

int type

- 4 bytes, or 32 bits
- Only 2^{32} different values!
  From -2^{31} to $2^{31} - 1$
  $2^{31} = 2\ 147\ 483\ 648$
  Slightly more than 2 billion
- $50\ 000 \cdot 50\ 000 = 2.5$ billion — couldn't fit!

Python

- In C++ and Java basic integers have fixed size
  — 4 bytes in int, for example
- In Python they grow as needed
- So no overflow there
  But larger values — more space and time

long type

- 8 bytes, or 64 bits
- **long long** or **int64**_t in C++, **long** in Java
- 2^{64} different values
- From -2^{63} to $2^{63} - 1$
  Slightly more than $9 \cdot 10^{18}$
- $2^{31} \cdot 2^{31} = 2^{31+31} = 2^{62} < 2^{63}$
  Product of two ints always fits!
- $10^9 \cdot 10^9 = 10^{18} < 9 \cdot 10^{18}$

Dealing with overflow

How to beat overflow

```
int a = 50000;
int b = 50000;
long long c = a * b;
```

Wrong!

```
int a = 50000;
int b = 50000;
long long c = a * b;
long long c = 50000 * 50000;
```

Also wrong!

Integer constants are 32 bit almost everywhere

How to beat overflow correctly

- Have at least one factor of long type
  ```
  long long a = 50000;
  long long b = 50000;
  long long c = a * b;
  ```
- Cast explicitly
  ```
  int a = 50000;
  int b = 50000;
  long long c = (long long)a * b;
  ```

Keep in mind

- Long type — twice the memory, most likely slower
- Always check products for overflow using the limits from the statement
- Sums could lead to overflow just as easily

```
int a = 50000;
int b = 50000;
long long res = 0;
for (int i = 0; i < b; ++i) {
res += a;
}
```

Wrong!

If even 64 bits is not enough — think again

Summary

- Always check products and sums for overflow
- Estimate magnitude using worst-case input values
- Use 64 bit type when needed

Non integers

Non-integer arithmetic

- Simple arithmetics:
  $a/b \cdot b = a$
- $1/49 \cdot 49 \; != 1$
  0.9999999999999988898
- Close enough to one, but not *exactly* one

Rational numbers

- As fractions A/B where A, B are integers
- Easy to store — just a pair of integers
- Could do arithmetical operations:
  $A/B + C/D = (A \cdot D + C \cdot B)/(B \cdot D)$
  $A/B \cdot C/D = (A \cdot C)/(B \cdot D)$
  Exact value:
  different fractions $\Leftrightarrow$ different pairs (A, B) (if irreducible)
  $1/49 \cdot 49 = 49/49 = 1$

Rational numbers

- Not only the *magnitude* is bound — $A < 2^k$, but also the *precision* — the smallest positive number we could store is $1/(2^k-1)$
- No way of storing arbitrarily small positive numbers — infinitely many numbers of form 1/k
- *A* and *B* have to fit in an integer type, but
  $1/1 + ½ + \cdots + 1/25 = 34\,052\,522\,467/8\,923\,714\,800$
  — values grow fast even in sums!
- Irrational numbers: $\sqrt{2}$, π, e

Decimal fractions

- 2.37 = 237/100
  0.125 = 125/1000
- Whole number without the point/$10^{\text{how many digits after the point}}$
- $\sqrt{2}$ = 1.41421356 . . .
  2/3 = 0.66666666 . . .
- $\sqrt{2} \rightarrow$ 1.414
  2/3$\rightarrow$ 0.667
  Error $\leq 10^{-3}/2$

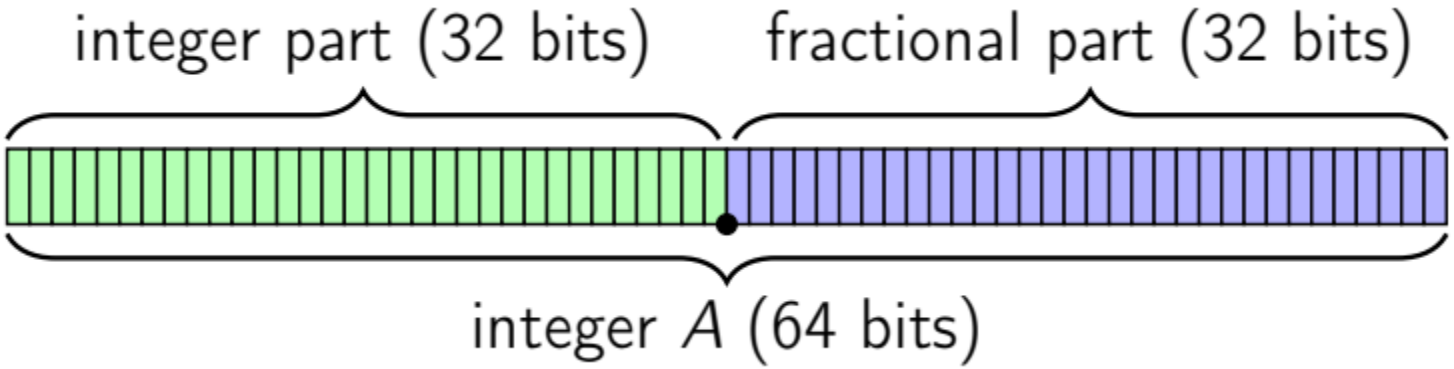
Binary fractions

- $1.01_2 = 101_2/4 = 5/4$
  $0.001_2 = 1_2/8 = 1/8$
- whole number without the point $/2^{\text{how many digits after the point}}$
- $2/3 = 0.10101010\ldots_2$
  $2/3 \rightarrow 0.101_2$
  Error $\leq 2^{-3}/2 = 2^{-4}$

Fixed point numbers and errors

Fixed point

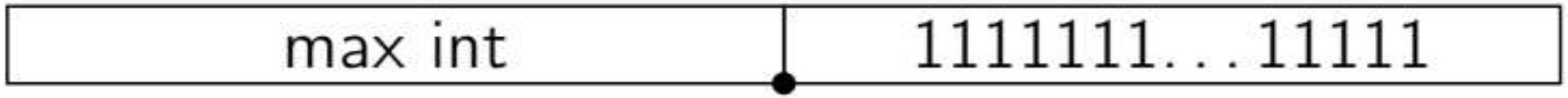
- Idea: always keep some fixed number of digits after the point



- Actually store integer *A*, but think of it as $A/2^{32}$

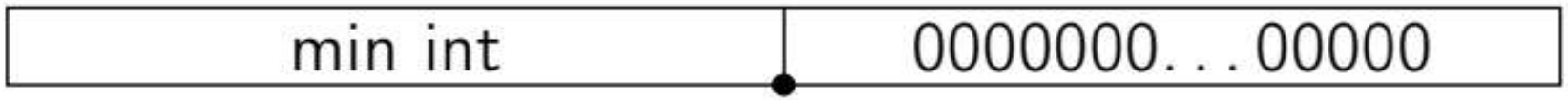
Fixed point

- Maximum value:



$$\frac{2^{63}-1}{2^{32}} = 2^{31} - \frac{1}{2^{32}} \simeq 2000000000$$

- Minimum value:

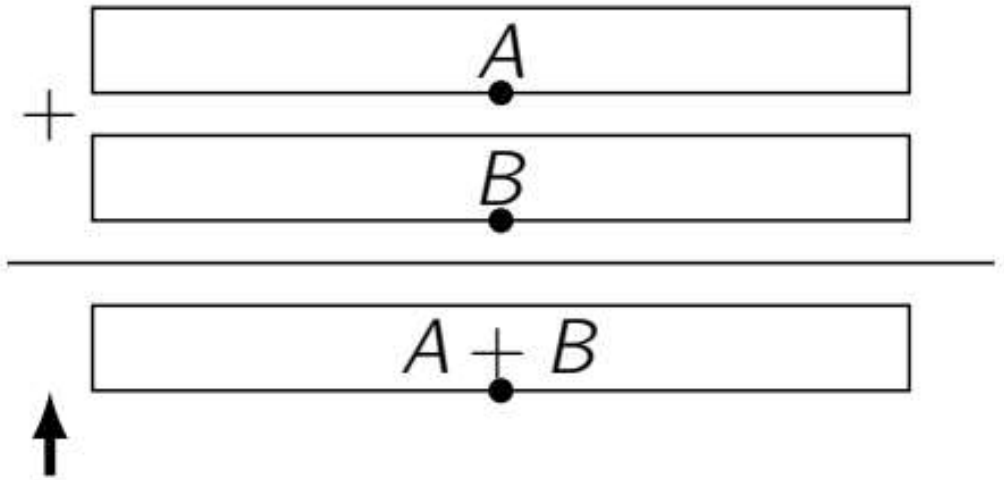


$$\frac{-2^{63}}{2^{32}} = -2^{31} \simeq -2000000000$$

Fixed point

- We could store any real number from -2^{31} to $2^{31} - \frac{1}{2^{32}}$ with error $\leq \frac{1}{2^{33}}$
- Addition:

$$\frac{A}{2^{32}} + \frac{B}{2^{32}} = \frac{A+B}{2^{32}}$$

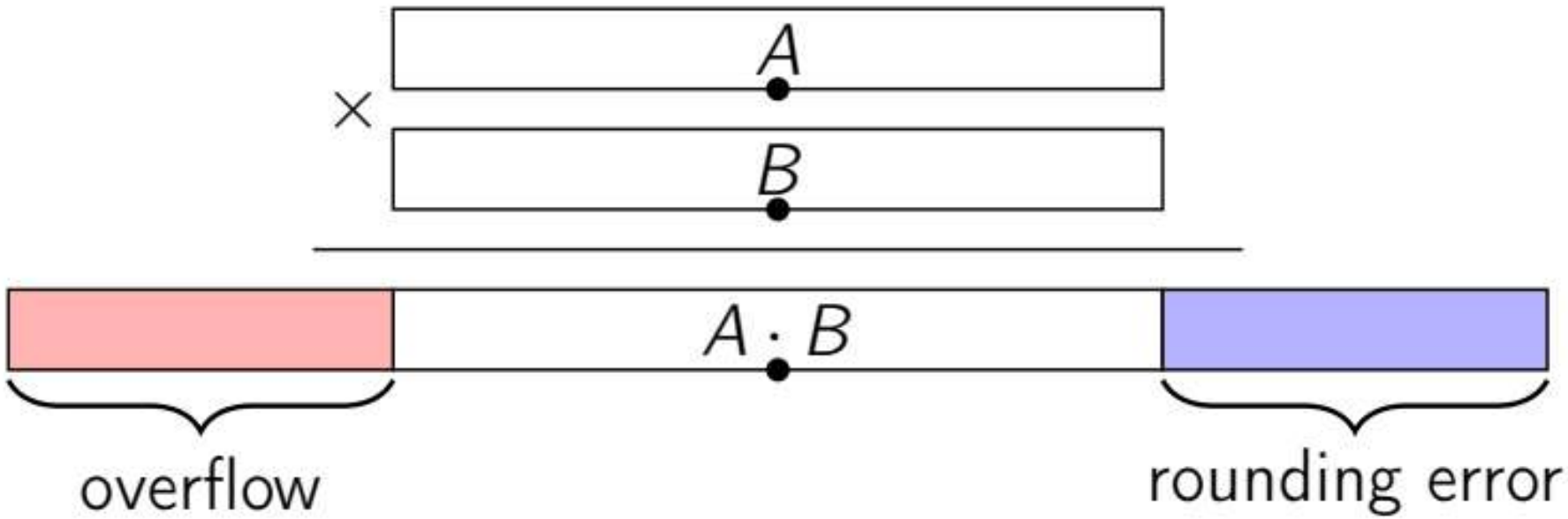


overflow is possible!

Fixed point

- Multiplication:

$$\frac{A}{2^{32}} \cdot \frac{B}{2^{32}} = \frac{A \cdot B}{2^{64}} = \frac{A \cdot B / 2^{32}}{2^{32}}$$



Absolute error

- The *absolute error* — the absolute difference between the value we have and the value we want
- Storage: some real number $a \to$ our 64-bit fixed-point representation $\widehat{a}$: $|a - \widehat{a}| \leq 2^{-33}$
- Addition: $|(a + b) - (\widehat{a} + \widehat{b})| \leq |a - \widehat{a}| + |b - \widehat{b}|$
  $\widehat{a}$ and $\widehat{b}$ just rounded from a and b, then
  error $\leq 2 \cdot 2^{-33} = 2^{-32}$
  More operations — larger error

Absolute error

- Multiplication:

$$|a \cdot b - \widehat{a} \cdot \widehat{b}| \leq$$

$$|a \cdot b - \widehat{a} \cdot b + \widehat{a} \cdot b - \widehat{a} \cdot \widehat{b}| \leq$$

$$|b| \cdot |a - \widehat{a}| + |\widehat{a}| \cdot |b - \widehat{b}|$$

- $a = \widehat{a} = 10^9$, $b = 1$, $\widehat{b} = 1 + 10^{-9}$
  $a \cdot b = 10^9$, $\widehat{a} \cdot \widehat{b} = 10^9 + 1$
  So the error's grown from 10^{-9} to 1!

Relative error

- The *relative error* — the absolute error divided by the magnitude of the exact value

$$\frac{|a - \widehat{a}|}{|a|}$$

- Multiplication:

$$\frac{|a \cdot b - \widehat{a} \cdot \widehat{b}|}{|a \cdot b|} \leq \frac{|b| \cdot |a - \widehat{a}| + |\widehat{a}| \cdot |b - \widehat{b}|}{|a \cdot b|}$$

$$\simeq \frac{|a - \widehat{a}|}{|a|} + \frac{|b - \widehat{b}|}{|b|}$$

Relative error

- $a = \widehat{a} = 10^9$, $b = 1$, $\widehat{b} = 1 + 10^{-9}$
  $a \cdot b = 10^9$, $\widehat{a} \cdot \widehat{b} = 10^9 + 1$
  Relative error

$$\frac{|10^9 - (10^9 + 1)|}{10^9} = 10^{-9}$$

- Addition: $a = \widehat{a} = 10^9$, $b = -10^9 + 1$, $\widehat{b} = -10^9$

$$\frac{|(a + b) - (\widehat{a} + \widehat{b})|}{|a + b|} = \frac{1}{1} = 1,$$

from

$$\frac{|a - \widehat{a}|}{|a|} = 0, \quad \frac{|b - \widehat{b}|}{|b|} \simeq 10^{-9}$$

Summary – Fixed point

- Fixed point behaves well with the absolute error
- But the relative error depends on the magnitude!
  $a \simeq 2^{31}$ — 64 correct binary digits
  $a \simeq 2^{-32}$ — only one correct binary digit!
- On “average” for a number $a \simeq 1$, the first half of the digits is not used
  We can do better!

Floating point numbers

Floating point

- Idea: Each number has its own most important digits

$$\ldots 000101001.0100110 \ldots$$

- The space is limited
  So it's natural to store some fixed number of *first* digits

$$1.01001010 \cdot 2^{5}$$

and the distance between the first one and the point — to know the actual position of the point

$$01001010 \quad 0101$$

Floating point

- We could store any fraction between 1.00000000 and 1.11111111 with any shift from -8 to 7
- Maximum value

$$1.11111111 \cdot 2^{7}$$

$$11111111.1 = 255.5$$

- Minimum positive value

$$1.00000000 \cdot 2^{-8} = \frac{1}{256}$$

- For any number, we round to first 9 digits, so the relative error $\leq 2^{-9}$

Floating point addition

$$1.01110101 \cdot 2^3 + 1.10010110 \cdot 2^{-1}$$

$$\begin{array}{rr} & 1011.10101 \\ + & 0.110010110 \\ \hline & 1100.011100110 \end{array}$$

$$1.10001110 \cdot 2^3$$

Tail of the smaller gets rounded!

Floating point multiplication

$$1.01110101 \cdot 2^3 \times 1.10010110 \cdot 2^{-1} =$$

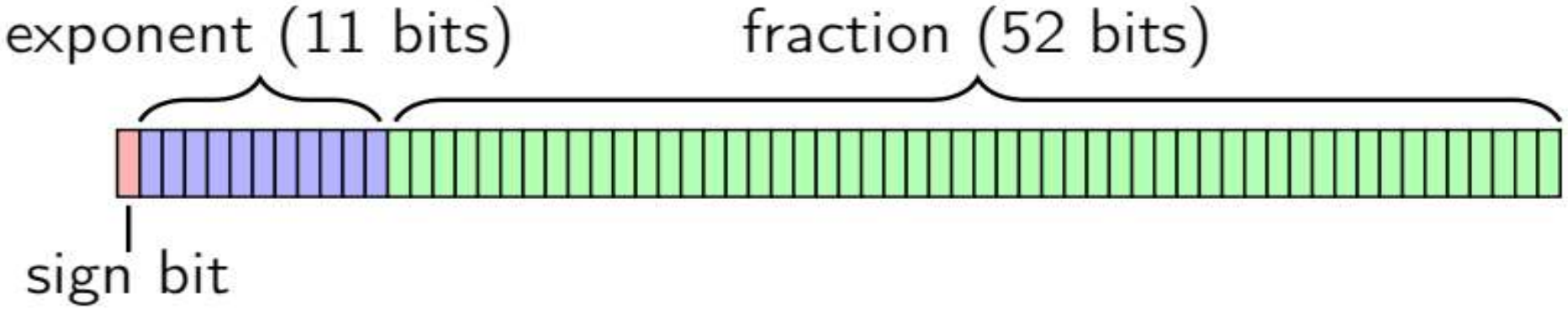
$$1.01110101 \times 1.10010110 \cdot 2^{3+(-1)} =$$

$$1.10010011\,1110001110 \cdot 2^2$$

$$1.10010100 \cdot 2^2$$

The product has more digits, need to round!

Double type



$$(-1)^s(1.f_0f_1\ldots f_{51})_2 \cdot 2^e$$

- Maximum value $2^{2^{10}} \simeq 10^{309}$
- Minimum positive value $2^{-2^{10}} \simeq 10^{-309}$
- Huge range of magnitude for 11 bits
  Would be 2 kilobytes for fixed point
- Each number with 53 accurate binary digits,
  about 16 decimal digits

Where and how to use doubles

Double vs 64 bit integer

- For doubles
  - Less actual digits: 53 vs 64 — no exponents in integers
  - More time on computations — doubles could be 1.5-2 times slower
  - No overflow in doubles
    Fractional part is always first digits
    Possible overflow in exponent, but only at 10^{300}
  - Errors everywhere
    1.0 / 49, sqrt(2)
    >>> 0.1 + 0.2
    0.30000000000000004
    and they grow!

Integers wherever possible

- Rational numbers $\frac{9}{13}$
- Decimal fractions $\$2.49 = 249¢$
- Roots:

  `while i < sqrt(n)` $\rightarrow$ `while i * i < n`
- Comparing lengths:

  $|(x, y)| = \sqrt{x^2 + y^2}$

  $\sqrt{a} < \sqrt{b} \iff a < b$

Integers as long as possible

- $\frac{11}{7} + \frac{1}{2} + \frac{5}{14} = \frac{34}{14}$
  five floating point operations vs **one**
- $1/2/3/5 = 1/(2 \cdot 3 \cdot 5)$
  three floating point operations vs **one**
- $\sqrt{5} \cdot 2 \cdot 3 = \sqrt{2^2 \cdot 3^2 \cdot 5}$
  three floating point operations vs **one**
- But watch for integer overflow!

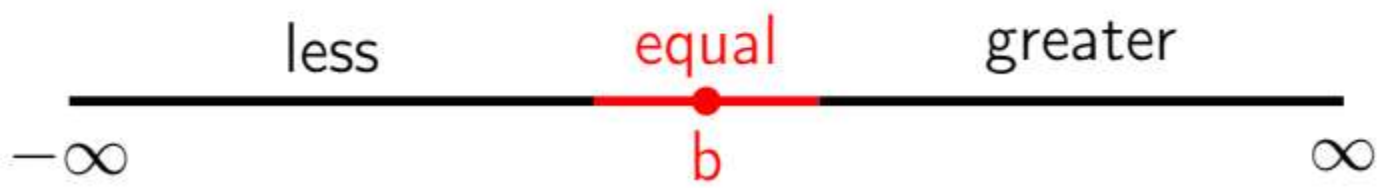
Doubles are needed

- Most common case: floating point in output
  "Output ... with absolute or relative error no more than 10^{-6}.“
- Print answer with some fixed large number of digits after the point
  cout << fixed << setprecision(20) << ans;
  System.out.format("%.20f", ans);
  printf("%.20f" % ans)

How to live with errors

- Consider values of small difference equal

statement	integers	doubles
a is equal to b	`a == b`	`abs(a - b) < eps`
a is *strictly* less than b	`a < b`	`a < b - eps`
a is less than or equal to b	`a <= b`	`a < b + eps`



- If strictness is not important, usual a < b is still better (e.g. while sorting)
- Truncate carefully with floor and ceil:
  instead of floor(a),
  better floor(a + eps)
  or else if a should be 1, but has an error of 10^{-9}, floor(0.999999999) is zero

How to choose eps

- It could be proven, that certain value of eps is enough
  Knowing how errors grow while
  - Storing
  - Summing
  - Multiplying
  - …
    numbers, you could bound the difference from the exact value and use it as eps
- But this is rarely done in contests
- Usually it’s enough to take some reasonable value like 1e-8 or 1e-9

How to choose eps

- What if your eps doesn't work?
- By usual means of debugging, find where errors appear first
- If indeed eps is the root cause then either:
  - eps is too big, and unequal values are treated as equal, then you should decrease eps — take the next power of 10, e.g. $10^{-8} \rightarrow 10^{-9}$
  - or eps is too small, and equal values are treated as unequal, then you should increase eps — e.g. $10^{-8} \rightarrow 10^{-7}$

More on floating point

Order of computations

- `>>> (1e18 + 1) - 1e18`
  `0.0`
  Values of different magnitude sum up with large errors! Try to avoid that
- $x^2 - y^2 = (x - y) \cdot (x + y)$
  `>>> y = 1e9`
  `>>> x = y + 1`
  `>>> x**2 - y**2`
  `2000000000.0`
  `>>> (x - y) * (x + y)`
  `2000000001.0`
  $(10^{18} + 2 \cdot 10^9 + 1) - 10^{18}$
  $(10^9 + 1 - 10^9) \cdot (10^9 + 1 + 10^9) = 1 \cdot (2 \cdot 10^9 + 1)$

Other types

- Single precision float — 32 bit analogue of double
  Do not use!
- C++: long double — 80 or 64 bit, depending on the compiler
  cout << numeric_limits<long double>::digits;
  64 or 53
- Java/Python: BigDecimal/decimal — has many leading digits as needed but costs space and time

Special values of double

- cout << 1.0 / 0;
  inf
  Positive infinity
- cout << 1.0 / 0 - 1.0 / 0 << '\n';
  -nan
  Not a number
- You want to avoid inf and nan in output
  cout << sqrt(-1e-9);
  -nan
  sqrt(x) — could lead to nan
  sqrt(max(x, 0)) — good

Summary – Handling numbers

- Doubles always come with errors
- Use integers where possible
- Always compare doubles with eps
  Never use ==
- Reorder computations — try not to add values of different magnitude
- Watch out for inf and nan

Handling numbers

- Quiz 3 – Numbers
- Assignment-commonStruggles

Other Search algorithms

Agenda

- Greedy algorithms
  - Warm-up
  - Proving correctness
  - Activity selection
  - Maximum Scalar Product
  - Greedy ordering
- Binary search

Warm up

Greedy Algorithm

- Makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem.
- Often important to sort first. Something a greedy approach will work only with the right order

Greedy Algorithm

- Largest number
  Input: A sequence of digits *d*0*, . . . , dn*-1(i.e., integers from 0 to 9).
  Output: The largest number that can be obtained by concatenating the given digits in some order.
- Example
  Input: 2, 3, 9, 3, 2
  Output: 93322

Greedy Algorithm

- Idea
  - Start with the largest digit
  - What is left is the same problem: concatenate the remaining digits to get as large number as possible
- Code

```
1 def largest(digits):
2         result = []
3
4         while len(digits) > 0:
5                 max_digit = max(digits)
6                 digits.remove(max_digit)
7                 result.append(max_digit)
8
9     return "".join(map(str, result))
```

- Running time: $O(n^2)$

Money Change

- Money change
  Input: Non-negative integer *m*.
  Output: The minimum number of coins with denominations 1, 5, and 10 that changes *m*.
- Example
  Input: 28
  Output: 6 (10 + 10 + 5 + 1 + 1 + 1)
- Idea

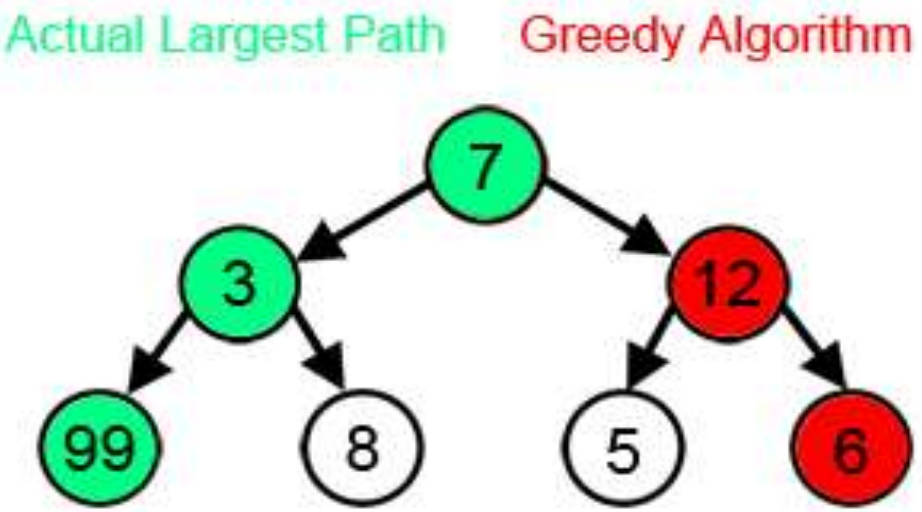
  Take a coin *c* with the largest denomination that does not exceed *m*
  What is left is the same problem: change (*m* - *c*) with the minimum number of coins

Code

- 1 **def** change (m, c o i n s) :
  2 r e s u l t = [ ]
  3
  4 **while** m > 0 :
  5 max_coin = **max**(c **for** c **in** c o i n s **i f** c <= m)
  7 m -= max_coin
  8 r e s u l t . append (max_coin)
  9
  10 **return** "+" . j o i n (**map**(**s tr** , r e s u l t))
- change (28 , [1 , 5 , 1 0])
- 10+10+5+1+1+1
  Running time: $O(m \cdot |coins|)$

Greedy Algorithm - Limitations

- A greedy strategy may not produce an optimal solution because it makes decisions based only on the information it has at any one step, without regard to the overall problem



n will choose what appears to be the optimal immediate choice, so it wi

Examples of greedy applications

- Huffman encoding which is used to compress data
- Dijkstra's algorithm, which is used to find the shortest path through a graph
- Bicoloring a graph

Analysis

- Compact and efficient solutions
- But it is more of a coincidence that they work correctly!
  - largest([2, 21]) returns 212 instead of 221
  - change(8, [1, 4, 6]) returns 6+1+1 instead of 4+4
- A priori, there should be no reason why a sequence of locally optimal moves leads to a global optimum
- In rare cases when a greedy strategy works, one should be able to prove its correctness

Proving correctness

Proving correctness

- At each step a greedy algorithm restricts the search space by selecting the most profitable piece of a solution
  - Largest number: instead of considering all numbers that can be obtained by concatenating the given digits, let's consider only number starting with the maximum digit
  - Change problem: instead of considering all ways of changing the given amount, let's consider only ways including a coin with the largest denomination
- One needs to show that the restricted search space contains at least one optimum solution

Template for Proving Correctness

- Take some optimum solution
- If it belongs to the restricted search space, then we are done
- If it does not belong to the restricted search space, tweak it so that it is still optimum (or even better) and belongs to the restricted search space

Largest Number: Correctness

- Lemma
  Let *N* be the largest number that can be obtained by concatenating digits *d*0, . . . , *dn*-1 in some order.
  Then *N* starts with the largest digit *di*.
- Proof
  Assume the contrary: $N = dj\alpha di\beta$, where $dj < di$ and α, β are sequences of digits. But then $N' = di\alpha dj\beta$ is greater than *N*, a contradiction. (**It is essential here that *di* and *dj* are single digit integers**!)

Money Change: Correctness

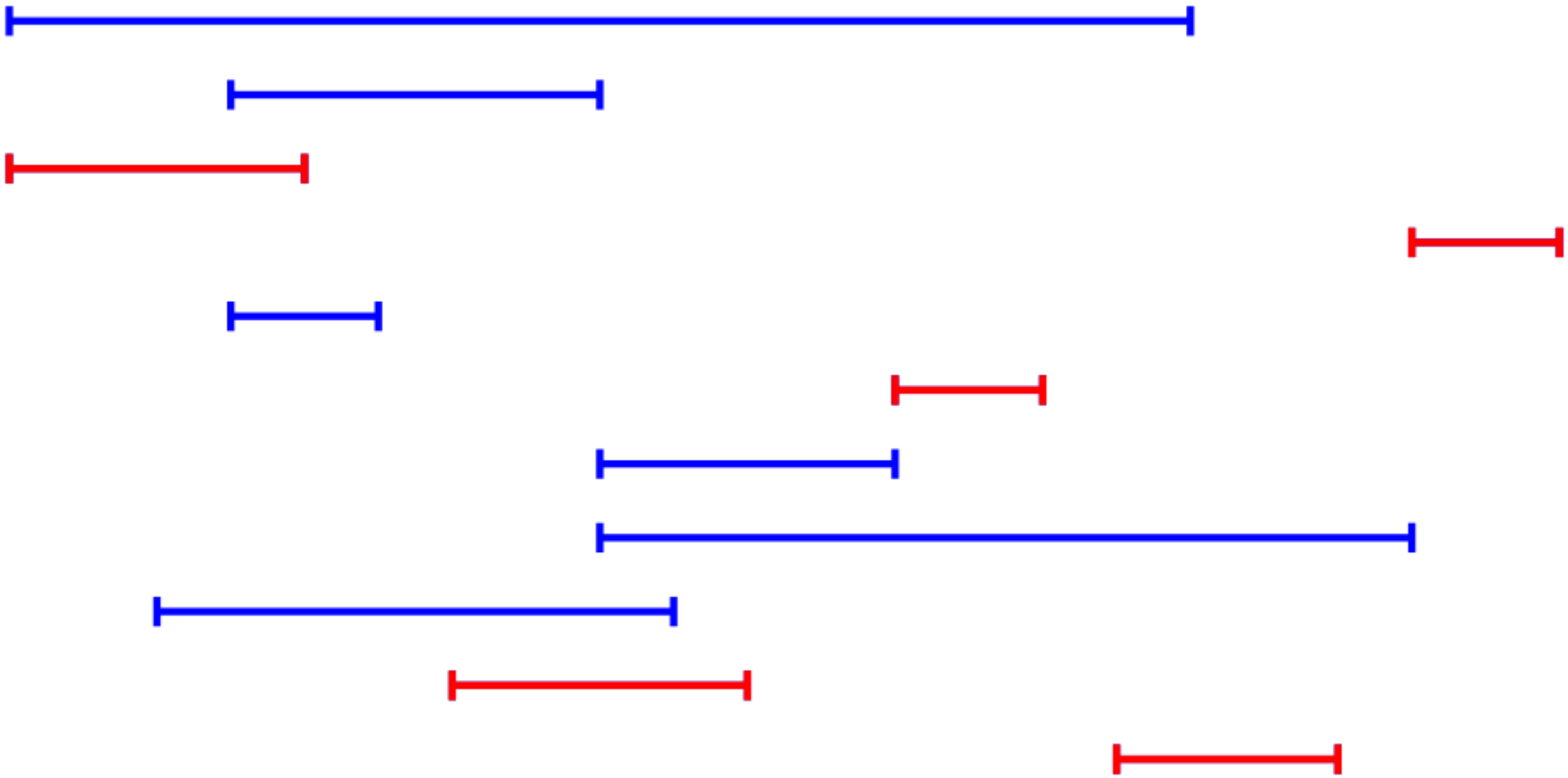
- Lemma
  For any positive integer *m*, there exists an optimal way of changing *m* using a coin with the largest denomination $D \in \{1, 5, 10\}$ that does not exceed *m*.
- Proof
  - m < 5, D = 1: m is changed using 1's only
  - 5 ≤ m < 10, D = 5: if 5 is not used, then there are at least five 1's; replace them with 5
  - 10 ≤ m, D = 10:
    - if there are at least two 5's, replace them with 10;
    - if there is just one 5, then there must be at least five 1's, replace them with 10;
    - if there are no 5's, there must be ten 1's, replace them with 10
- Observation
  - It is the last case where the analysis breaks for {1, 4, 6}
  - 6 ≤ m, D = 6?:
    - if there are at least two 4's, cannot replace them with 6+1+1 as it makes it longer;

Activity Selection

Activity selection

- Activity selection
  Input: A set of *n* segments on a line.
  Output: The maximum number of non-overlapping segments.
- Example
  Input: [2, 6], [1, 4], [7, 9], [3, 8]
  Output: 2 ([1, 4], [7, 9])

Example

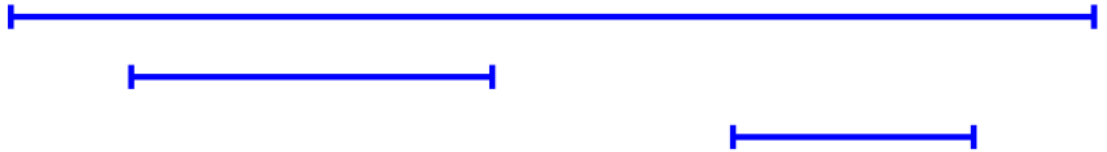


Greedy Strategy

- Unlike the previous two problems, in this case it is not immediate what the most profitable move is
- Wild guesses:
  - Take the shortest segment
  - Take the segment with the minimal left endpoint
  - Take the segment with the minimal right endpoint

- Taking the shortest segment does not work:

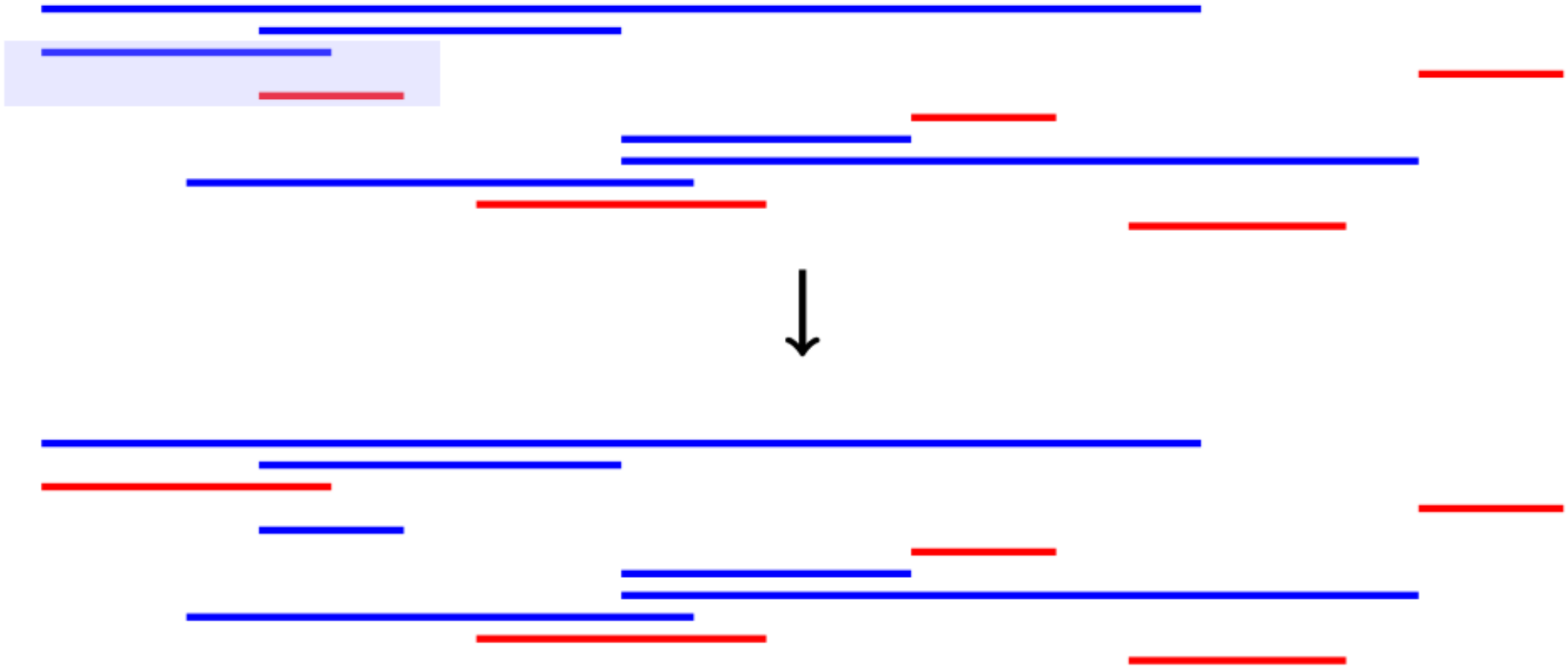
- Taking the segment with the minimal left endpoint does not work:



Greedy Strategy

- Lemma
  There exists an optimal solution containing the segment with the smallest right endpoint.
- Proof
  Let [*a, b*] be a segment with the smallest right endpoint and let *S* be an optimal solution such that [*c, d*] is its segment with the minimal right endpoint.
- $b \leq d$. If $b = d$, nothing needs to be done, so assume that $b < d$
- Replace [*c, d*] with [*a, b*] in *S*. Then, it is still a solution (if [*c, d*] does not intersect any other segment in *S*, then neither does [*a, b*]) and it is optimal

Visually



Algorithm

- Algorithm
  - Take the segment with the minimal right endpoint into a solution
  - Remove all segments that intersect it
  - Repeat
- Running time
  - $O(n^2)$

Maximum Scalar Product

Maximum Scalar product

- Maximum Scalar Product
  Input: Two sequences of n integers:
  $A = [a0, \ldots, an\text{-}1]$ and $B = [b0, \ldots, bn\text{-}1]$.
  Output: The maximum value of $a0c0 + \ldots + an\text{-}1cn\text{-}1$, where $c0, \ldots, cn\text{-}1$ is a permutation of $b0, \ldots, bn\text{-}1$.
- Example
  Input: [2, 3, 9], [7, 4, 2]
  Output: 79 ($79 = 2 \cdot 2 + 3 \cdot 4 + 9 \cdot 7$)

Greedy Strategy

- Guess?
- Maximum of A paired with maximum of B
- Lemma
  There exists an optimal solution where the maximum element *ai* of *A* is paired with the maximum element *bj* of *B*
- Proof
  Consider an optimal solution S
  - If it pairs ai and bj, then we are done
  - Otherwise S = aibp + aqbj + · · ·
  - Let's swap these two pairs:
    - S′ = aibj + aqbp + · · ·
    - S′ is not worse than S: S′ − S = aibj + aqbp − aibp − aqbj = (ai − aq)(bj − bp) ≥ 0

Code

```
1 def s c a l ar_ pr o d u ct (A, B ) :
2       a s s e r t len (A) == len (B)
3       r e s u l t = 0
4       while len (A) > 0 :
5             am, bm = max(A) , max(B)
6             r e s u l t += am * bm
7             A. remove (am)
8             B. remove (bm)
9       return r e s u l t
```

- Running time: $O(n^2)$

Greedy ordering

Greedy Ordering

- A greedy strategy usually defines a greedy ordering in a natural way
  - Largest number: the larger digit is better
  - Money change: the larger denomination is better
  - Activity selection: the activity with a smaller ending time is better
  - Scalar product: the larger bi is better
- Then, everything boils down to sorting with respect to this ordering

Compact code

- 1 **def** largest (digits) :
  2 **return** "" . j o i n (**map**(**s t r** , **so r ted** (digits , reverse=True)))

- 1 **def** change (m) :
  2 **return** m // 10 + (m % 10) // 5 + (m % 5)

- 1 **def** scalar _ product (A, B) :
  2 assert **len** (A) == **len** (B)
  3 A, B = **sorted** (A) , **sorted** (B)
  4 **return sum**(A[i] * B[i] **for** i **in range** (**len** (A)))

Ordering Correctness

Proving that a specific ordering leads to a correct greedy strategy: if in a solution $a_1, a_2, \ldots, a_n$, $a_i \not\preceq a_{i+1}$ for some i, then swapping a_i and a_{i+1} can only improve this solution

Summary Greedy algorithms

- Construct a solution piece by piece, always choosing the most profitable piece
- Pros: efficient, easy to implement
- Cons: rarely work, not so easy to prove correctness

Binary search

Binary search - Principle

- Find an element in a sorted array
- Generally find a boundary between two regions (min(max))
- O(log(n))

Binary search – Code/Pseudo-code

```
/* Searches value 'x' in sorted array a[left], a[left+1], ..., a[right].
   Returns the index of 'x' in array, or -1 if it cannot be found. */
search(a, left, right, x)
    if left > right then
        return -1
    mid := left+floor((right-left) / 2)
    if a[mid] = x then
        return mid
    else if a[mid] > x then
        return search(a, left, mid-1, x)
    else
        return search(a, mid+1, right, x)
```

```
// Search value 'x' in array a [0], a [1], ..., a [n - 1]
int search (int a [], int n, int x) {
    int r, c;
    for (int l = 0, r = n - 1; l <= r;) {
        c = l + (r - l) / 2;
        if (a [c] > x) r = c - 1;
        else if (a [c] < x) l = c + 1;
        else return c;
    }
    return -1;
}
```

Note: It's always safe to choose the mid point by taking mid=left + (right-left)/2 because it prevents the overflow when the array index is large and both right and left fall near the highest index

Assignment

- Assignment-greedy

Segment tree

References

- See
- https://www.coursera.org/lecture/competitive-programming-core-skills/segment-tree-structure-uJWRn
  - **https://cp-algorithms.com/data_structures/segment_tree.html#toc-tgt-1**
  - https://fr.wikipedia.org/wiki/Arbre_de_segments

Assignment

- Quiz4-segment tree

Dynamic programming

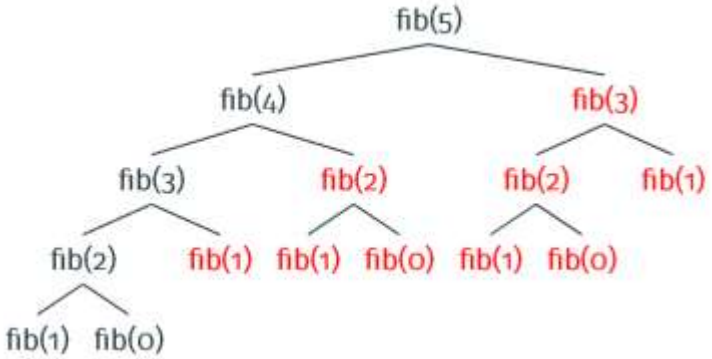
Dynamic programming

- Extremely powerful algorithmic technique with applications in optimization, scheduling, planning, economics, bioinformatics, etc
- At contests, probably the most popular type of problems
- A solution is usually not so easy to find, but when found, is easily implementable
- **Need a lot of practice!**

Fibonacci

- Fibonacci
  - Fib(0)=0
  - Fib(1) =1
  - Fib(i) = Fib(i-1)+Fib(i-2)

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .
- We want to compute fib(n)
  - Input: An integer $n \geq 0$.
    Output: The n-th Fibonacci number Fn.
- 1 **def** fib (n) :
  2 **if** n <= 1 :
  3 **return** n
  4 **return** fib (n - 1) + fib (n - 2)

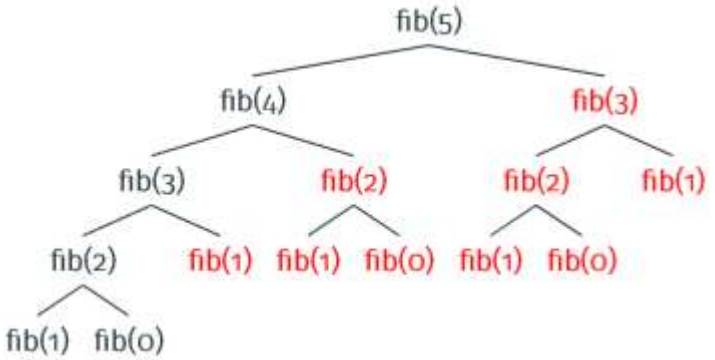


Running Time

- Essentially, the algorithm computes Fn as the sum of Fn ones
  - Hence its running time is O(Fn)
- But Fibonacci numbers grow exponentially fast:
  - Fn ≈ φ^n, where φ = 1.618 . . . is the golden ratio
- E.g., F_{150} is already 31 decimal digits long
- The Sun may die before your computer returns F150

Reason

- Many computations are repeated
- “Those who cannot remember the past are condemned to repeat it.” (George Santayana)
- A simple, but crucial idea: instead of recomputing the intermediate results, let’s store them once they are computed



Memoization

- int M[100];
- int fib(int i) {

```
if (i == 0) return 0;
if (i == 1) return 1;
if (M[i]) return M[i];
return M[i] = fib(i-1) + fib(i-2);
}
```

- Think of initial values, and resetting it if needed, setting up the memory table and maximal size for the memory table. If values are not dense use (unordered_) map for memory table

Memoization

```
1  def fib(n):
2    if n <= 1:
3      return n
4    return fib(n - 1) + fib(n - 2)
```

```
1  T = dict()
2
3  def fib(n):
4    if n not in T:
5      if n <= 1:
6        T[n] = n
7      else:
8        T[n] = fib(n - 1) + fib(n - 2)
9
10   return T[n]
```

Hmmm…

- But do we really need all this fancy stuff (recursion, memoization, dictionaries) to solve this simple problem?
- After all, this is how you would compute F_5 by hand:

1. $F_0 = 0, F_1 = 1$
2. $F_2 = 0 + 1 = 1$
3. $F_3 = 1 + 1 = 2$
4. $F_4 = 1 + 2 = 3$
5. $F_5 = 2 + 3 = 5$

Dynamic Programming – Bottom up

- Example: Fibonacci
  - Let remove the recursion/ reformulate the problem

```
int T[100];
int fib(int n) {
T[0] = 0; T[1] = 1;
for (int i = 2; i <= n; i++)
        T[i] = T[i-1] + T[i-2];
return T[n];
}
```

```
1  def fib(n):
2    T = [None] * (n + 1)
3    T[0], T[1] = 0, 1
4
5    for i in range(2, n + 1):
6      T[i] = T[i - 1] + T[i - 2]
7
8    return T[n]
```

Hmmm again…

- But do we really need to waste so much space ?
- No need to keep everything in memory
- Example: Fibonacci

```
int T[3];
int fib(int n) {
T[0] = 0; T[1] = 1;
for (int i = 2; i <= n; i++)
        T[i%3] = T[(i-1)%3] + T[(i-2)%3];
return T[n%3];
}
```

```
1  def fib(n):
2    if n <= 1:
3      return n
4
5    previous, current = 0, 1
6    for _ in range(n - 1):
7      new_current = previous + current
8      previous, current = current, new_current
9
10   return current
```

Running time

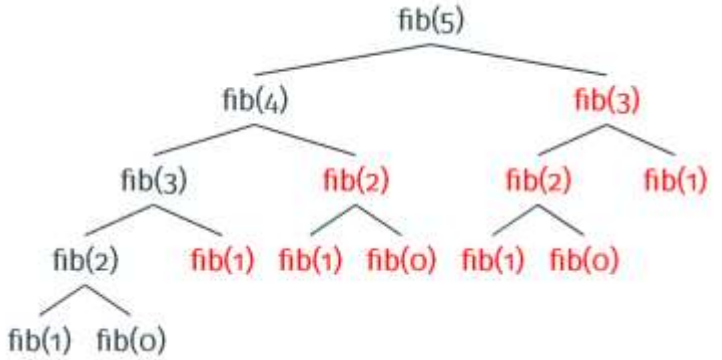
- O(n) additions
- On the other hand, recall that Fibonacci numbers grow exponentially fast: the binary length of Fn is O(n)
- In theory: we should not treat such additions as basic operations
- In practice: just F100 does not fit into a 64-bit integer type anymore, hence we need bignum arithmetic

Dynamic Programming

- Dynamic programming gives us a way to design custom algorithms which systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency).
- Dynamic programming algorithms are based on recursive algorithms.
- There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping sub-problems.
  - If a problem can be solved by combining optimal solutions to *non-overlapping* sub-problems, the strategy is called "divide and conquer" instead. This is why merge sort and quick sort are not classified as dynamic programming problems.
- *Optimal substructure* means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub-problems. Such optimal substructures are usually described by means of recursion.
  - Shortest path from u to v

Dynamic Programming

- *Overlapping* sub-problems means that the space of sub-problems must be small, that is, any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems. Example of Fibonacci
- Two approaches:
  - top-down -store the solution to sub-problems in a table (Memoization)
  - Bottom-up – reformulate the problem in bottom up and solve the sub-problems first and use their solution to build on a bigger solution
  - At the same time, the case of Fibonacci numbers is a slightly artificial example of dynamic programming since it is clear from the very beginning what intermediate results we need to compute the final result



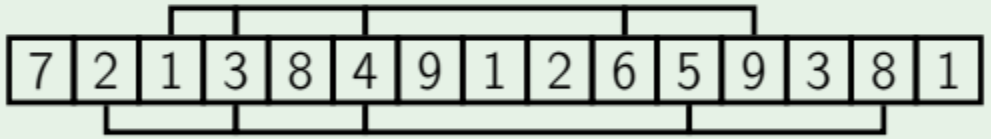
Subproblems

Longest Increasing Subsequence

Input:	An array $A = [a0, a1, \ldots, an\text{-}1]$.
Output:	A longest increasing subsequence (LIS), i.e., $ai1, ai2, \ldots, aik$ such that $i1 < i2 < \ldots < ik$, $ai1 < ai2 < \cdots < aik$, and k is maximal.

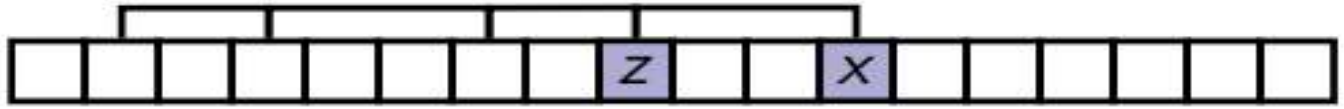
Example

Example

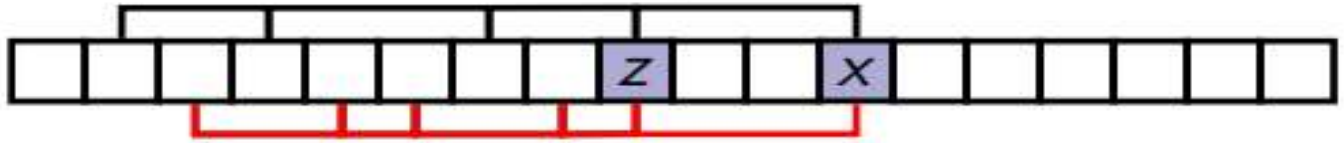


Analyzing an Optimal Solution

- Consider the last element x of an optimal increasing subsequence and its previous element z:



- First of all, $z < x$
- Moreover, the prefix of the IS ending at z must be an optimal IS ending at z as otherwise the initial IS would not be optimal:



- Optimal substructure by "cut-and-paste" trick

Subproblems and Recurrence Relation

- Let *LIS*(*i*) be the optimal length of a LIS ending at *A*[*i*]
- Then
  LIS(*i*) = 1+max{***LIS(j)*****:** *j* < *i* and *A*[*j*] < *A*[*i*]}
- Convention: maximum of an empty set is equal to zero
- Base case: *LIS*(0) = 1 – You MUST define the base case(s)

The generalization is important

$$LIS(i) = 1 + LIS\ (j) for\ a\ given\ j\ \ so\ that\ j < i\ and\ A[i] < A[i]$$

As j is unknown , we need to generalize.
For two prefixes, we are maximizing LIS
So $LIS(i) = 1 + \max(LIS\ (j) for\ a\ given\ j\ \ so\ that\ j < i\ and\ A[i] < A[i])$

Algorithm

- When we have a recurrence relation at hand, converting it to a recursive algorithm with memoization is just a technicality
  - We will use a table *T* to store the results: ***T*** **[*i*] = *LIS*(*i*)**
  - Initially, *T* is empty. When *LIS*(*i*) is computed, we store its value at *T* [*i*] (so that we will never recompute *LIS*(*i*) again)
  - The exact data structure behind *T* is not that important at this point: it could be an array or a hash table

Recursion

```
def lis (A, i ) :
        for j in range ( i ) :
                if A[ j ] < A[ i ] :
                        if (lis (A, j ) + 1 > T[i]):
                                T[i]= lis (A, j ) + 1
        return T[i]

A = [ 7 , 2 , 1 , 3 , 8 , 4 , 9 , 1 , 2 , 6 , 5 , 9 , 3 ]
print (max( lis (A, i ) for i in range ( len (A ) ) ) )
```

Memoization

```
T = dict ( ) //dictionary

def lis (A, i ) :
  //initialization by base case and pruning based on dictionary
  if i not in T:
            T[ i ] = 1

            for j in range ( i ) :
                        if A[ j ] < A[ i ] :
                                    T[ i ] = max(T[ i ] , lis (A, j ) + 1) //rewrite recursion based on memoization

  return T[ i ] //return value

A = [ 7 , 2 , 1 , 3 , 8 , 4 , 9 , 1 , 2 , 6 , 5 , 9 , 3 ]
print (max( lis (A, i ) for i in range ( len (A ) ) ) )
```

Running Time

- The running time is quadratic ($O(n^2)$): there are n “serious” recursive calls (that are not just table look-ups), each of them needs time $O(n)$ for the for loop (not counting the inner recursive calls)

Table and Recursion

- We need to store in the table *T* the value of *LIS*(*i*) for all *i* from 0 to *n* – 1
- Reasonable choice of a data structure for *T*: an array of size *n*
- Moreover, one can fill in this array iteratively instead of recursively

Iterative Algorithm

```
def lis (A) :
        T = [None] * len (A)

        for i in range (len(A)):
                T[i] = 1
                for j in range (i) :
                        if A[j] < A[i] and T[i] < T[j] + 1:
                                T[i] = T[j] + 1

        return max(T[i] for i in range (len(A)))
```

- Crucial property: when computing $T[i]$, $T[j]$ for all $j < i$ have already been computed
- Running time: $O(n^2)$ so memorization is good enough

Reconstructing a Solution

Reconstructing a Solution

- How to reconstruct an optimal IS?
- In order to reconstruct it, for each subproblem we will keep its optimal value and a choice leading to this value

Reconstructing a Solution

```
T = dict ( )
previous =  dict () // declaration

def lis (A, i ) :
  if i not in T:
    T[ i ] = 1
    previous[i] = -1  // initialisation

    for j in range ( i ) :
      if A[ j ] < A[ i ] :
            //rewrite max
            temp = lis (A, j ) + 1
            if temp > T[i] :
                        T[ i ] = temp
                        previous[i] = j  // update
 return T[ i ]

A = [ 7 , 2 , 1 , 3 , 8 , 4 , 9 , 1 , 2 , 6 , 5 , 9 , 3 ]
print (max( lis (A, i ) for i in range ( len (A ) ) ) )
```

```
def lis (A) :
        T = [None] * len (A)
        previous =  [None] * len(A)  // declaration

        for i in range (len(A)):
                T[i] = 1
                previous[i] = -1  // initialisation
                for j in range (i) :
                        if A[j] < A[i] and T[i] < T[j] + 1:
                                T[i] = T[j] + 1
                                previous[i] = j  // update


        return max(T[i] for i in range (len(A)))
```

Example

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1
previous	-1	-1	-1	1	3	3	4	-1	2	5	5	9	8	9	-1

Sol[4] = A[1]=2
previous(1) = -1

Sol[3] = A[3]=3
previous(3) = 1

Sol[2] = A[5]=4
previous(5) = 3

Sol[1] = A[9]=6
previous(9) = 5

Max (T) = 5
i[max(T)]=13
Sol[0] = A[13]=8
previous(13) = 9

Unwinding the solution

```
last = 0
//find the max and its index
for i in range (1,len(A)) :
        if T[i] > T[last] :
                last = i

sol= []
current = last
while current >= 0 :
        //add the index
        sol.append(current)
        current=prev[current]
//reverse
sol.reverse()
//print a[index]
return [A[i] for i in sol]
```

Reconstructing without prev

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	7	2	1	3	8	4	9	1	2	6	5	9	3	8	1
T	1	1	1	2	3	3	4	1	2	4	4	5	3	5	1

i=2 so that T[i]= current_T-1
And A[i]<3
Sol[4] = A[2]=1
current_T= 1

i=5 so that T[i]= current_T-1
And A[i]<5
Sol[2] = A[5]=4
Current_T= 3

i=3 so that T[i]= current_T-1
And A[i]<4
Sol[3] = A[3]=3
Current_T=3

i=10 so that T[i]= current_T-1
And A[i]<8
Sol[1] = A[10]=5
Current_T= 4

Current_T=Max (T) = 5
i[max(T)]=13
Sol[0] = A[13]=8

LIS - Summary

- Optimal substructure property: any prefix of an optimal increasing subsequence must be a longest increasing subsequence ending at this particular element
- Subproblem: the length of an optimal increasing subsequence ending at i-th element
- A recurrence relation for subproblems can be immediately converted into a recursive algorithm with memoization
- A recursive algorithm, in turn, can be converted into an iterative one
- An optimal solution can be recovered either by using an additional bookkeeping info or by using the computed solutions to all subproblems

Dynamic Programming - Summary

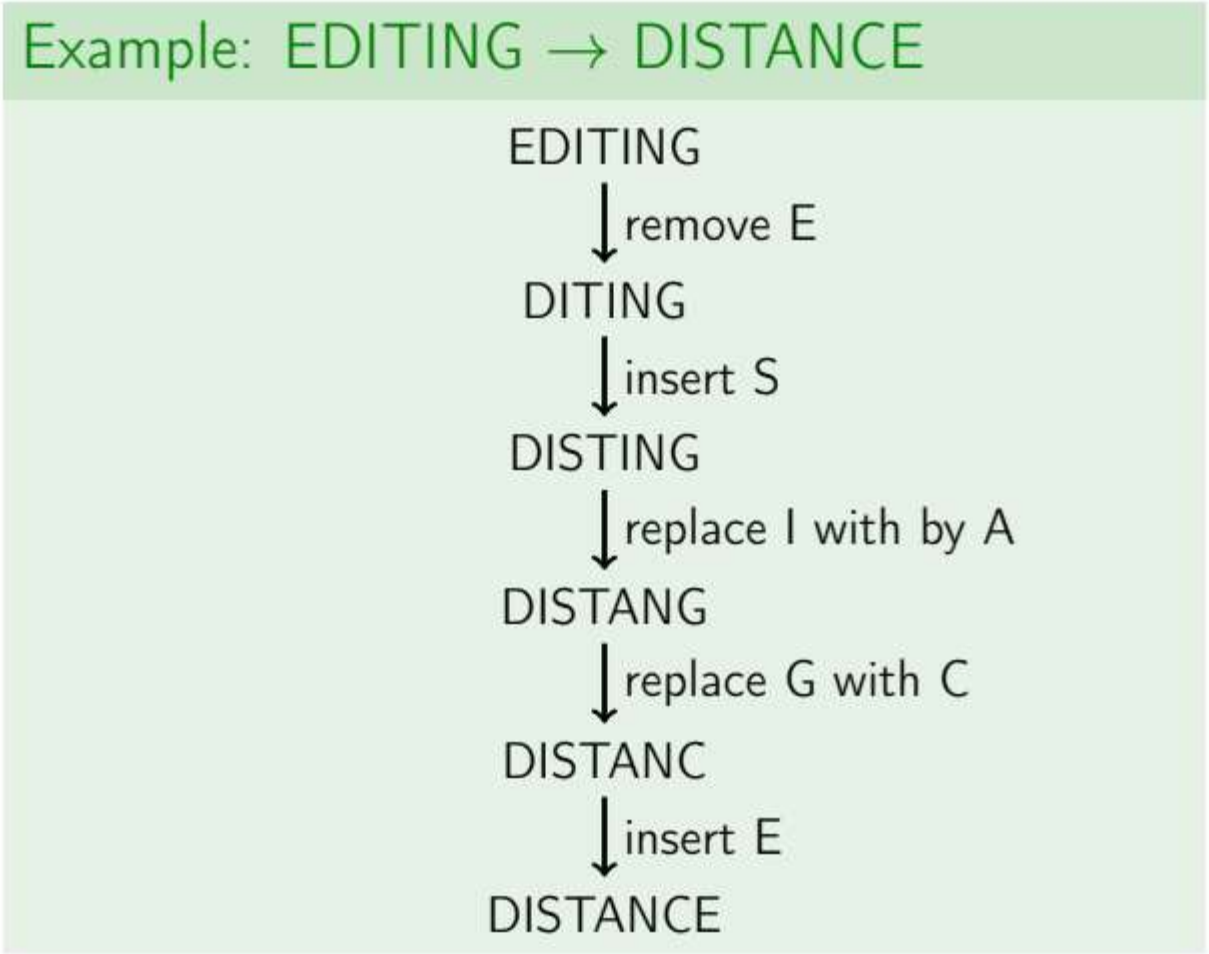
- **Subproblems (and recurrence relation on them) is the most important ingredient of a dynamic programming algorithm**
- Two common ways of arriving at the right subproblem:
  - Analyze the structure of an optimal solution
  - Implement a brute force solution and optimize it

Edit Distance

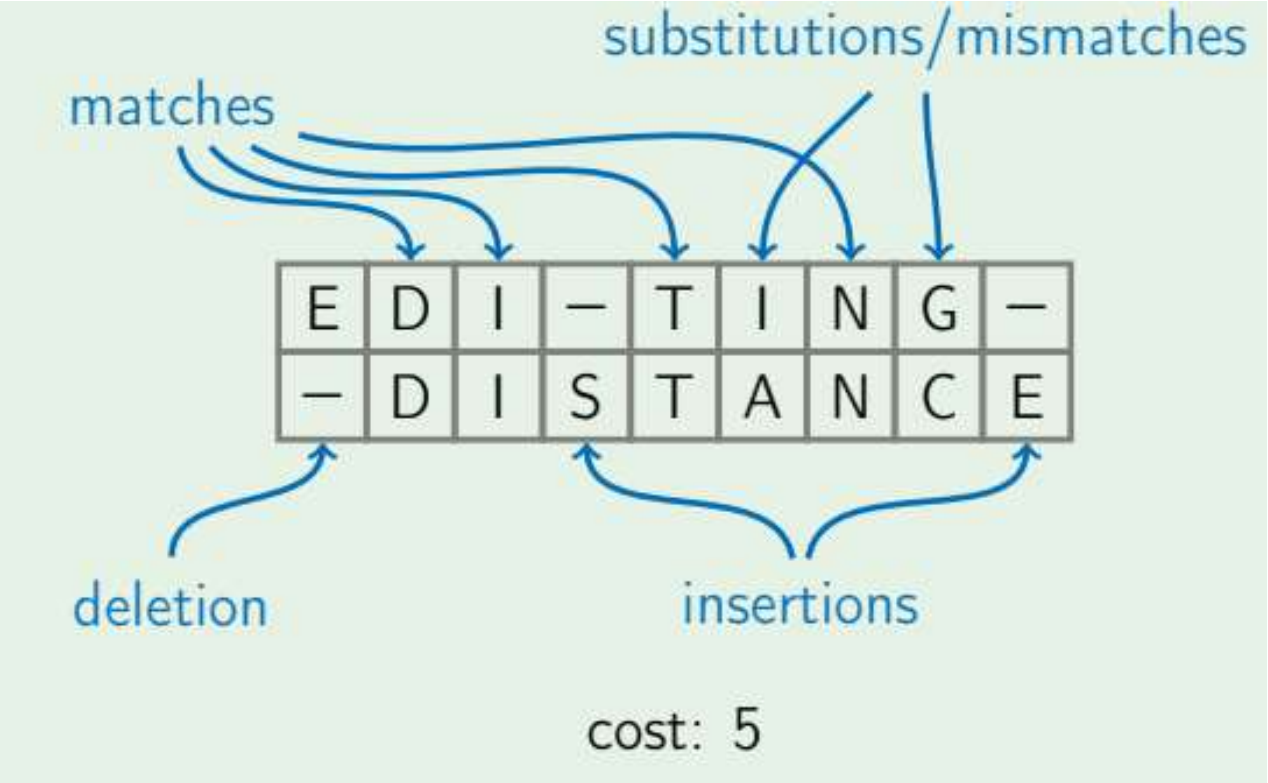
Structure of an optimal solution

Edit Distance Problem

- Input: Two strings $A[0 \ldots n-1]$ and $B[0 \ldots m-1]$.
- Output: The minimal number of insertions, deletions, and substitutions needed to transform A to B. This number is known as edit distance or Levenshtein distance.

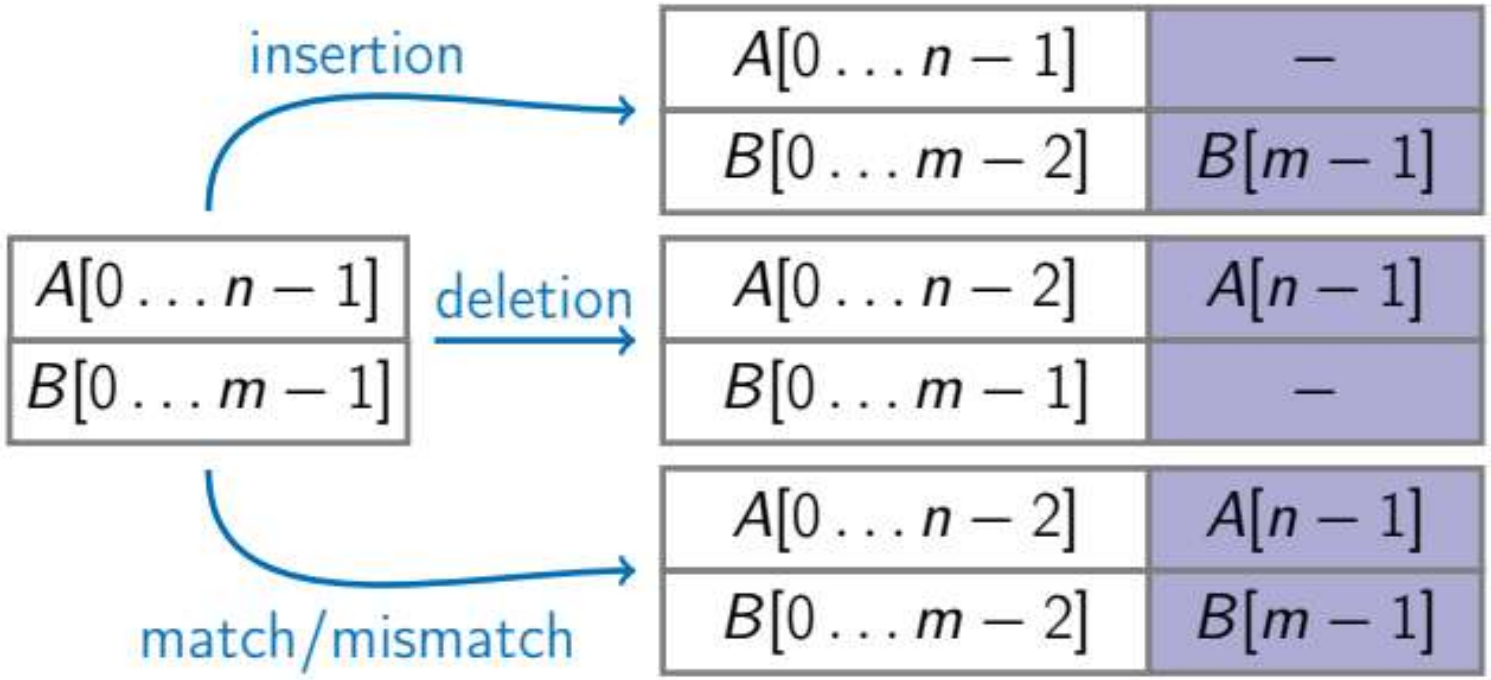


Alignment



- **2** replacements/substitutions /mismatches
- **2** insertions
- **1** deletion
- 4 matches

Analyzing an Optimal Alignment



Subproblems

- Let *ED*(*i, j*) be the edit distance of *A*[0 . . . *i* - 1] and *B*[0 . . . *j* - 1].
- We know for sure that the last column of an optimal alignment is either an insertion, a deletion, or a match/mismatch or replacement.
- What is left is an optimal alignment of the corresponding two prefixes (by cut-and-paste).

$$ED(i,j) = \begin{cases} ED(i,j-1)+1 \text{ "}insertion\text{"} \\ ED(i-1,j)+1 \text{ "}deletion\text{"} \\ ED(i-1,j-1)+(A[i]!=B[j]:1;0) \text{ "}mismatch\ or\ match\text{"} \end{cases} \quad for\ a\ given\ (i,j)$$

By definition, ED(i,j) is the minimal number of insertion/deletion/mismatch

So $ED(i,j) = min \begin{cases} ED(i,j-1)+1 \text{ "}insertion\text{"} \\ ED(i-1,j)+1 \text{ "}deletion\text{"} \\ ED(i-1,j-1)+(A[i]!=B[j]:1;0) \text{ "}mismatch\ or\ match\text{"} \end{cases} \quad for\ all\ (i,j)$

- Base cases: ED(i,0)= i for any i; ED(0,j)=j for any j

Recursive Algorithm

```
def edit_distance (a,b,i,j):
                if i == 0 : return= j
                elif j == 0 : return= i
                else :
                        diff = 0 if a[i−1] == b[j−1] else 1
                        return min(
                                        edit_distance(a,b,i−1,j) + 1 ,
                                        edit_distance(a,b,i,j−1)+ 1 ,
                                        edit_distance(a,b,i−1,j−1) + diff)


 print(edit_distance(a="editing",b="distance",i=7,j=8))
```

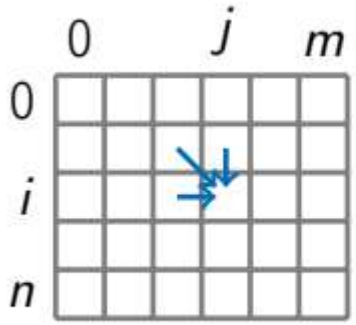
Memoization

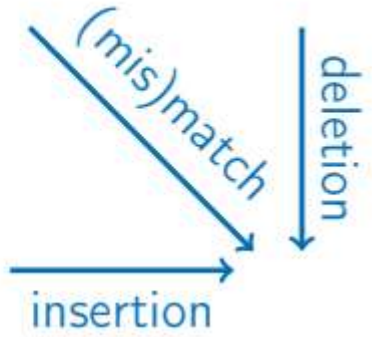
```
T= dict() //dictionary

def edit_distance (a,b,i,j):
        //initialization by base case and pruning based on dictionary
        if not (i,j) in T:
                if i == 0 : T[i,j] = j
                elif j == 0 : T[i,j] = i
                else :
                        diff = 0 if a[i−1] == b[j−1] else 1

                        //rewrite recursion based on memoization
                        T[i,j] = min(
                                        edit_distance(a,b,i−1,j) + 1 ,
                                        edit_distance(a,b,i,j−1)+ 1 ,
                                        edit_distance(a,b,i−1,j−1) + diff)

        return T[i,j] // return value


 print(edit_distance(a="editing",b="distance",i=7,j=8))
```

- Running time: O(i*j)

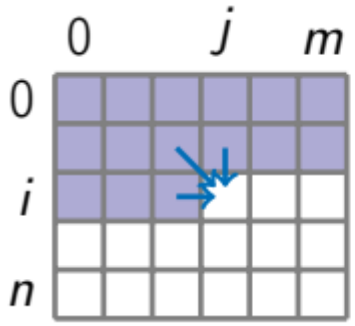
Converting to an iterative algorithm

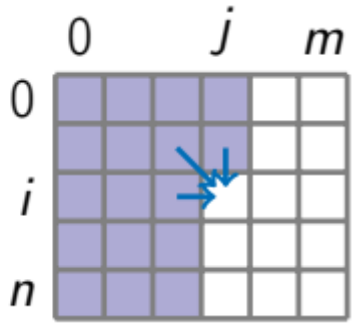
- Use a 2D table to store the intermediate results
- *ED*(*i, j*) depends on *ED*(*i* - 1*, j* - 1), *ED*(*i* - 1*, j*), and *ED*(*i, j* - 1):





- Fill in the table row by row or column by column:





Iterative Algorithm

```
def edit_distance(a,b) :
        T = [[float("inf")]*(len(b)+1)
                for _ in range (len(a)+1)]
        for i in range (len(a)+1):
                T[i][0] = i
        for j in range (len(b)+1):
                T[0][j]=j

        for i in range (1,len(a)+1):
                for j in range (1,len(b)+1):
                        diff = 0 if a[i−1]==b[j−1] else 1
                        T[i][j] = min(T[i−1][j] + 1 ,
                                T[i][j−1] + 1 ,
                                T[i−1][j−1]+diff)

        return T[len(a)][len(b)]

print (edit_distance (a="distance",b="editing"))
```

Example

			E	D	I	T	I	N	G
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
D	1	1	1	1	2	3	4	5	6
I	2	2	2	2	1	2	3	4	5
S	3	3	3	3	2	2	3	4	5
T	4	4	4	4	3	2	3	4	5
A	5	5	5	5	4	3	3	4	5
N	6	6	6	6	5	4	4	3	4
C	7	7	7	7	6	5	5	4	4
E	8	8	7	8	7	6	6	5	5

Edit Distance

Brute force

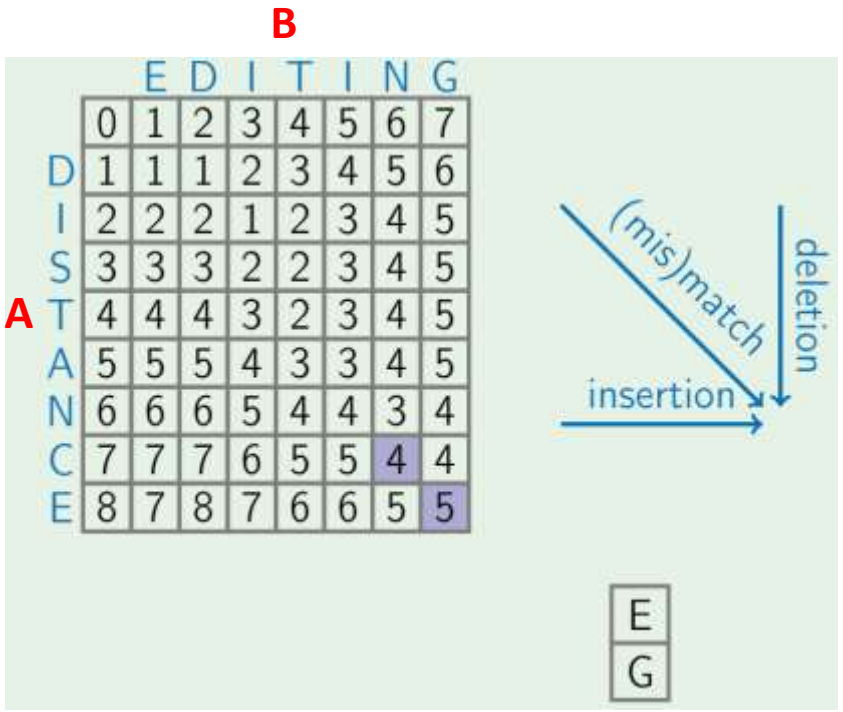
Brute Force

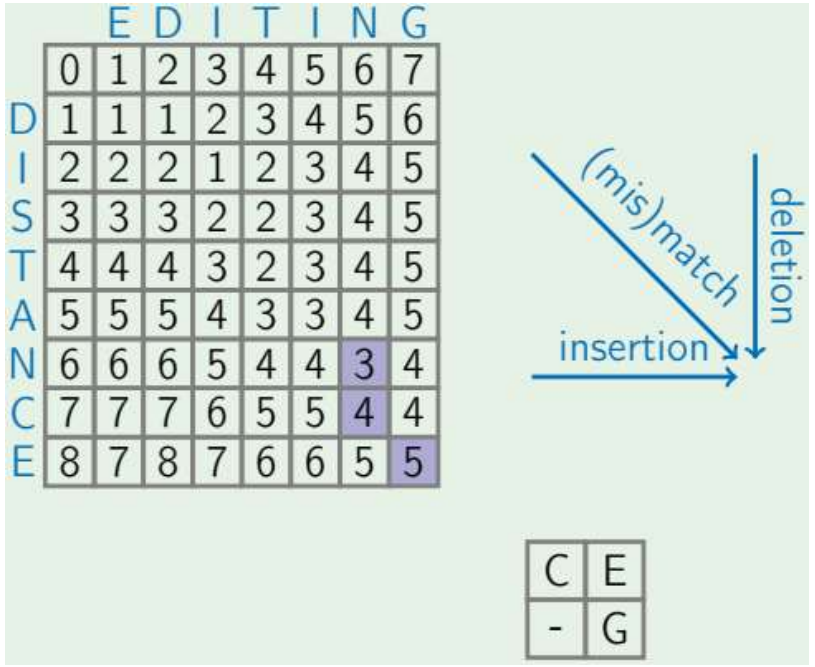
- Recursively construct an alignment column by column
- Then note, that for extending the partially constructed alignment optimally, one only needs to know the already used length of prefix of *A* and the length of prefix of *B*

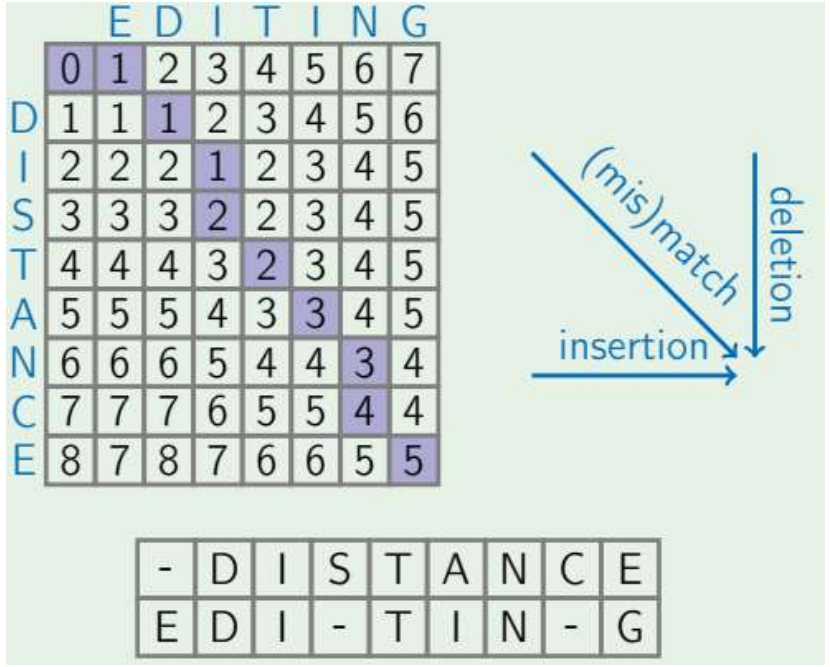
Reconstructing a Solution

- To reconstruct a solution, we go back from the cell(*n, m*) to the cell(0*,* 0)
- If *ED*(*i, j*) = *ED*(*i* - 1*, j*) + 1, then there exists an optimal alignment whose last column is a deletion
- If *ED*(*i, j*) = *ED*(*i, j* - 1) + 1, then there exists an optimal alignment whose last column is an insertion
- If *ED*(*i, j*) = *ED*(*i* - 1*, j* - 1) + diff(*A*[*i*]*, B*[*j*]), then match (if *A*[*i*] = *B*[*j*]) or mismatch (if *A*[*i*] *!= B*[*j*])

Reconstructing a Solution



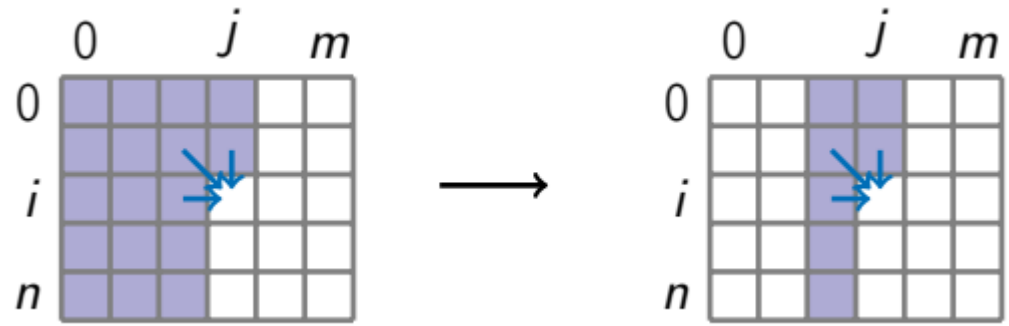




$$if\ T(i-1,j) = T(i,j) - 1, then\ deletion\ of\ A(i)$$
$$if\ T(i-1,j-1) = T(i,j) - 1, then\ mismatch\ of\ A(i)\ and\ B(j)$$
$$if\ T(i-1,j-1) = T(i,j), then\ match\ of\ A(i)\ and\ B(j)$$
$$if\ T(i,j-1) = T(i,j) - 1, then\ insertion\ of\ B(j)$$

Saving space

- When filling in the matrix it is enough to keep only the current column and the previous column:



- Thus, one can compute the edit distance of two given strings $A[1 \ldots n]$ and $B[1 \ldots m]$ in time $O(nm)$ and space $O(\min\{n, m\})$.
- However we need the whole table to find an actual alignment (we trace an alignment from the bottom right corner to the top left corner)
- There exists an algorithm constructing an optimal alignment in time $O(nm)$ and space $O(n + m)$ (Hirschberg's algorithm)

Weighted Edit Distance

- The cost of insertions, deletions, and substitutions is not necessarily identical
- Spell checking: some substitutions are more likely than others
- Biology: some mutations are more likely than others

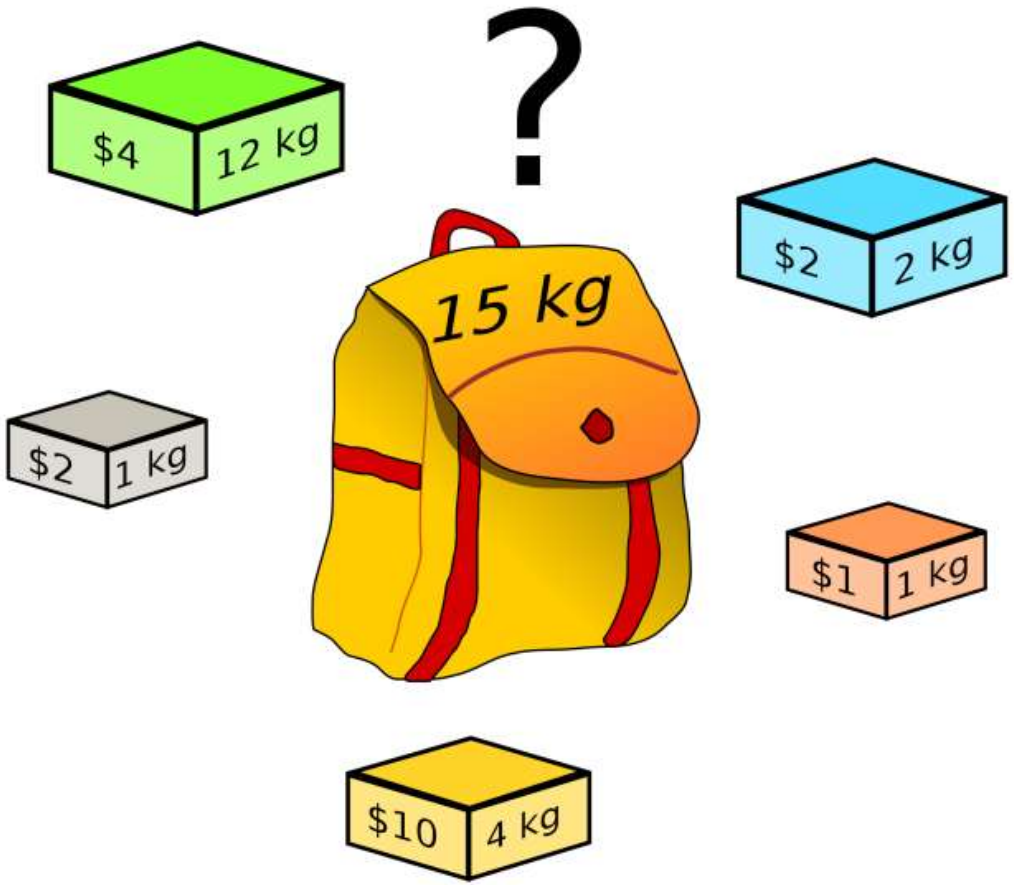
$$ED(i,j) = min \begin{cases} ED(i,j-1) + inscost(B[j]) \text{ "insertion"} \\ ED(i-1,j) + delcost(A[i]) \text{ "deletion"} \\ ED(i-1,j-1) + substcost(A[i],B[j]) \text{ "mismatch or match"} \end{cases} \quad for\ all\ (i,j)$$

Assignment

- Quiz5-Dynprog1
- assignment-DP1

More Dynamic programming

Knapsack Problem



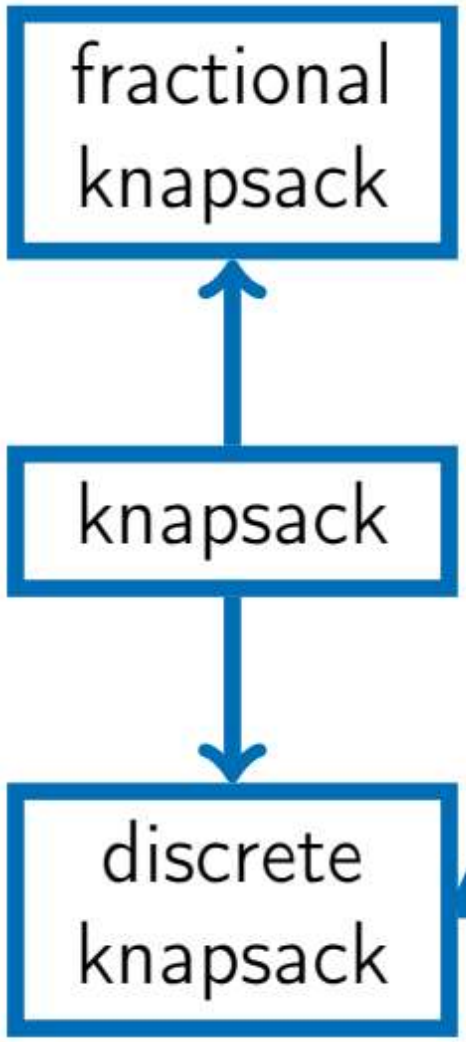
Goal

Maximize value ($) while limiting total weight (kg)

Applications

- Classical problem in combinatorial optimization with applications in resource allocation, cryptography, planning
- Weights and values may mean various resources (to be maximized or limited):
  - Select a set of TV commercials (each commercial has duration and cost) so that the total revenue is maximal while the total length does not exceed the length of the available time slot
  - Purchase computers for a data center to achieve the maximal performance under limited budget

Problem Variations



Greedy algorithm

Greedy does not work for discrete knapsack!
Need to design a dynamic programming solution

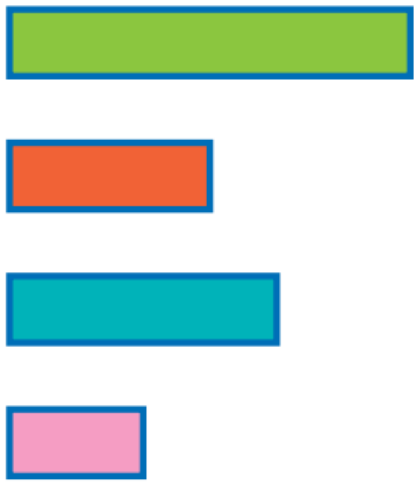
Example

Knapsack with Repetitions

Knapsack with Repetitions

With repetitions:
unlimited quantities

Without repetitions:
one of each item



Knapsack with repetitions problem

- **Input**: Weights *w*0, . . . , *wn*-1 and values *v*0, . . . , *vn*-1 of *n* items; total weight *W* (*vi*'s, *wi*'s, and *W* are non-negative integers).
- **Output**: The maximum value of items whose weight does not exceed *W*. Each item can be used any number of times.

Analyzing an Optimal Solution

- Consider an optimal solution and an item in it:

- If we take this item out then we get an **optimal** solution for a knapsack of total weight W - wi.

Subproblems

- Let *value*(u) be the maximum value of knapsack of weight u
  - $value(u) = \max_{i:w_i \leq w} \{value(u - w_i) + v_i\}$
- Base case: *value*(0) = 0
- This recurrence relation is transformed into a recursive algorithm in a straightforward way

Recursive Algorithm with memoization

```
T = dict ( )

def knapsack (w, v , u ) :
        if u not in T:
                T[ u ] = 0
                for i in range ( len (w ) ) :
                        if w[ i ] <= u :
                                T[ u ] = max(T[ u ] ,
                                        knapsack (w, v , u − w[ i ] ) + v [ i ] )
 return T[ u ]

 print ( knapsack (w=[6 , 3 , 4 , 2 ] ,  v =[30 , 14 , 16 , 9 ] , u=10))
```

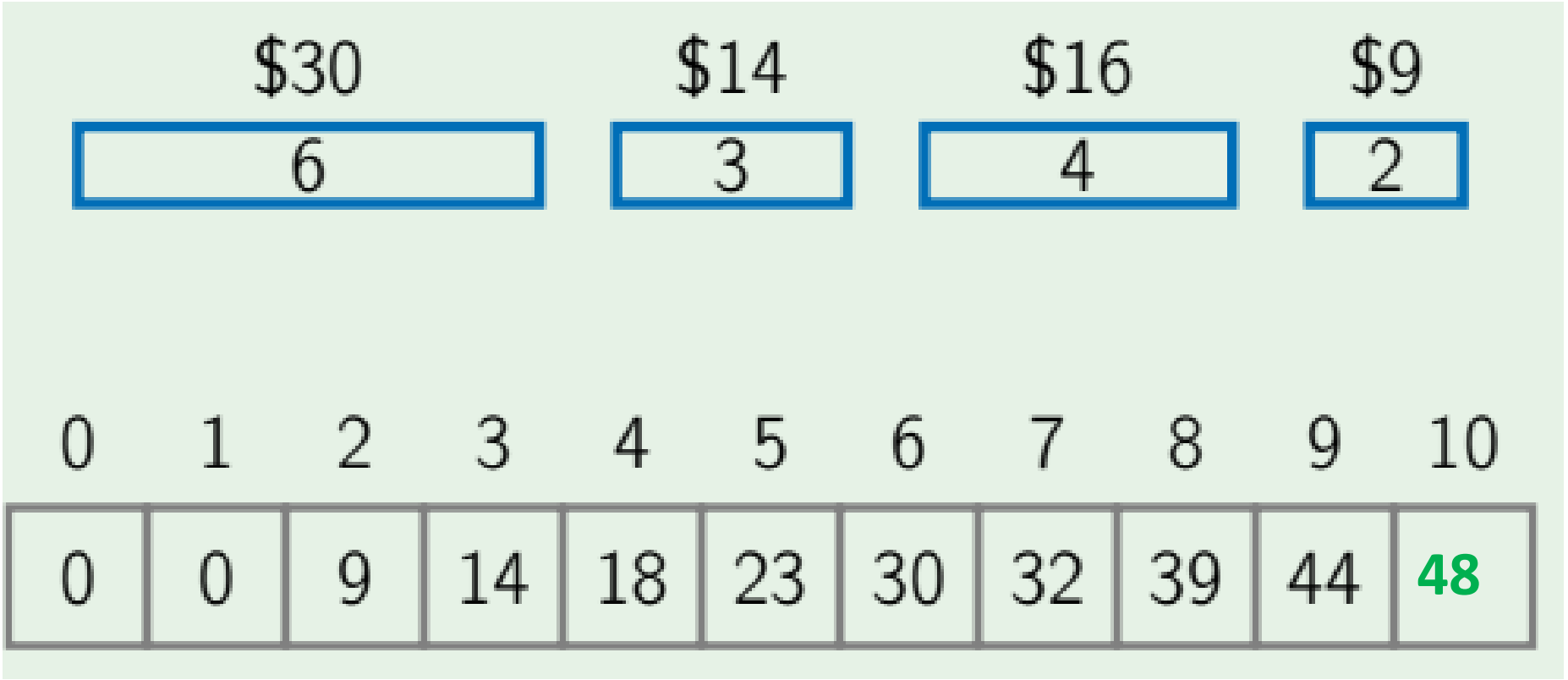
Recursive into Iterative

- As usual, one can transform a recursive algorithm into an iterative one
- For this, we gradually fill in an array T: $T[u] = value(u)$

Iterative Algorithm

```
def knapsack (W, w, v ) :
        T = [ 0 ] * (W + 1)

        for u in range ( 1 , W + 1 ) :
                for i in range ( len (w ) ) :
                        if w[ i ] <= u :
                                T[ u ] = max(T[ u ] , T[ u − w[ i ] ] + v [ i ] )

        return T[W]

print ( knapsack (W=10, w=[6 , 3 , 4 , 2 ] , v =[30 , 14 , 16 , 9 ] ) )
```

Example: W=10



Subproblems Revisited

- Another way of arriving at subproblems: optimizing brute force solution
- Populate a list of used items one by one

Brute force: Knapsack with Repetitions

```
def knapsack (W, w, v , items ) :
        weight = sum(w[ i ] for i in items )
        value = sum( v [ i ] for i in items )

        for i in range ( len (w ) ) :
                if weight + w[ i ] <= W:
                        value = max( value ,
                                knapsack (W, w, v , item s + [ i ] ) )

return value

print ( knapsack (W=10, w=[6 , 3 , 4 , 2 ] ,  v =[30 , 14 , 16 , 9 ] , item s = [ ] )
```

Subproblems

- It remains to notice that the only important thing for extending the current set of items is the weight of this set
- One then replaces items by their weight in the list of parameters

Knapsack without Repetitions

Knapsack without Repetitions

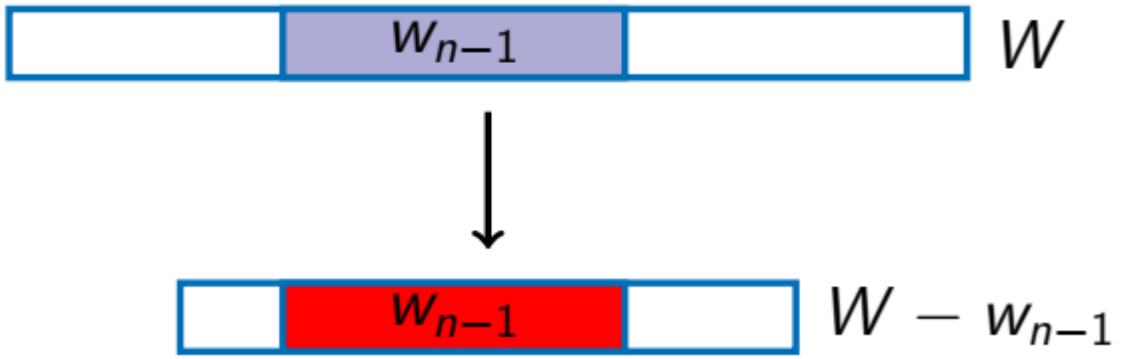
With repetitions:
unlimited quantities

Without repetitions:
one of each item

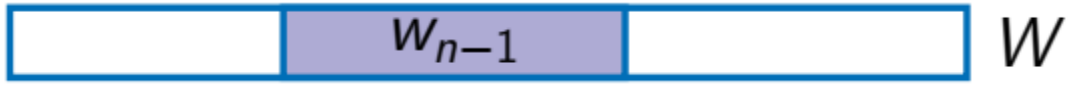
Knapsack without repetitions problem

- **Input**: Weights *w*0, . . . , *wn*-1 and values *v*0, . . . , *vn*-1 of *n* items; total weight *W* (*vi*'s, *wi*'s, and *W* are non-negative integers).
- **Output**: The maximum value of items whose weight does not exceed *W*. **Each item can be used at most once.**

Same Subproblems?



- If the last item is taken into an optimal solution:



  then what is left is an optimal solution for a knapsack of total weight $W - w_{n-1}$ using items 0, 1, . . . , n - 2.
- If the last item is not used, then the whole knapsack must be filled in optimally with items 0, 1, . . . , n - 2.

Subproblems

- For $0 \leq u \leq W$ and $0 \leq i \leq n$, *value*(u, i) is the maximum value achievable using a knapsack of weight u **and the first i items**.
- Base case: *value*$(u, 0) = 0$, *value*$(0, i) = 0$
- For $i > 0$, the item $i - 1$ is either used or not:

$$value(u, i) = \max\{value(u - w_{i-1}, i - 1) + v_{i-1}, value(u, i - 1)\}$$

Recursive Algorithm with memoization

```
T = dict ( )

def knapsack (w, v , u , i ) :
        if ( u , i ) not in T:
                if i == 0 :
                        T[ u , i ] = 0
                else :
                        T[ u , i ] = knapsack (w, v , u , i − 1)
                        if u >= w[ i − 1 ] :
                                T[ u , i ] = max(T[ u , i ] ,
                                        knapsack (w, v , u − w[ i − 1 ] , i − 1) + v [ i − 1 ]

return T[ u , i ]

print ( knapsack (w=[6 , 3 , 4 , 2 ] ,  v =[30 , 14 , 16 , 9 ] , u=10, i =4))
```

Recursive into Iterative

- As usual, one can transform a recursive algorithm into an iterative one
- For this, we gradually fill in an array T: $T[u,i] = value(u,i)$

Iterative Algorithm

```
def knapsack (W, w, v ) :
        T = [ [ None ] * ( len (w) + 1) for _ in range (W + 1 ) ]

        for u in range (W + 1 ) :
                T[ u ] [ 0 ] = 0

        for i in range ( 1 , len (w) + 1 ) :
                for u in range (W + 1 ) :
                        T[ u ] [ i ] = T[ u ] [ i − 1 ]
                        if u >= w[ i − 1 ] :
                                T[ u ] [ i ] = max(T[ u ] [ i ] ,
                                        T[ u − w[ i − 1 ] ] [ i − 1 ] + v [ i − 1 ] )

        return T[W] [ len (w ) ]


print ( knapsack (W=10, w=[6 , 3 , 4 , 2 ] , v =[30 , 14 , 16 , 9 ] ) )
```

Analysis

- Running time: $O(nW)$
- Space: $O(nW)$
- Space can be improved to $O(W)$ in the iterative version: instead of storing the whole table, store the current column and the previous one

Reconstructing a Solution

- As it usually happens, an optimal solution can be unwound by analyzing the computed solutions to subproblems
- Start with $u = W$, $i = n$
- If $value(u, i) = value(u, i - 1)$, then item $i – 1$ is not taken. Update i to $i – 1$
- Otherwise
  $value(u, i) = value(u - wi\text{-}1, i - 1) + vi\text{-}1$ and the item $i - i$ is taken. Update i to $i - 1$ and u to $u - wi\text{-}1$

Subproblems Revisited

- How to implement a brute force solution for the knapsack without repetitions problem?
- Process items one by one. For each item, either take into a bag or not

Brute force: Knapsack without Repetitions

```
def knapsack (W, w, v , items , last ) :
        weight = sum(w[ i ] for i in items )

        if last == len (w) − 1 :
                return sum( v [ i ] for i in items )

        value = knapsack (W, w, v , items , last + 1)
        if weight + w[ last + 1 ] <= W:
                items.append(last + 1)
                value = max( value ,
                        knapsack (W, w, v , items , last + 1 ) )
                items.pop ( )

        return value

print ( knapsack (W=10, w=[6 , 3 , 4 , 2 ] , 17 v =[30 , 14 , 16 , 9 ] , 18 items =[] , last =−1))
```

Final remarks

Recursive vs Iterative

- If all subproblems must be solved then an iterative algorithm is usually faster since it has no recursion overhead
- There are cases however when one does not need to solve all subproblems and the knapsack problem is a good example: assume that *W* and all *wi*'s are multiples of 100; then *value*(*w*) is not needed if *w* is not divisible by 100

Polynomial Time?

- The running time $O(nW)$ is not polynomial since the input size is proportional to $\log W$, but not W
- In other words, the running time is $O(n2^{\log W})$.
  E.g., for W = 10 345 970 345 617 824 751 (twenty digits only!) the algorithm needs roughly 10^{20} basic operations
- Solving the knapsack problem in truly polynomial time is the essence of the P vs NP problem, the most important open problem in Computer Science (with a bounty of $1M)

Chain matrix multiplication

Chain matrix multiplication

- **Input**: Chain of n matrices A0, . . . , An−1 to be multiplied.
- **Output**: An order of multiplication minimizing the total cost of multiplication

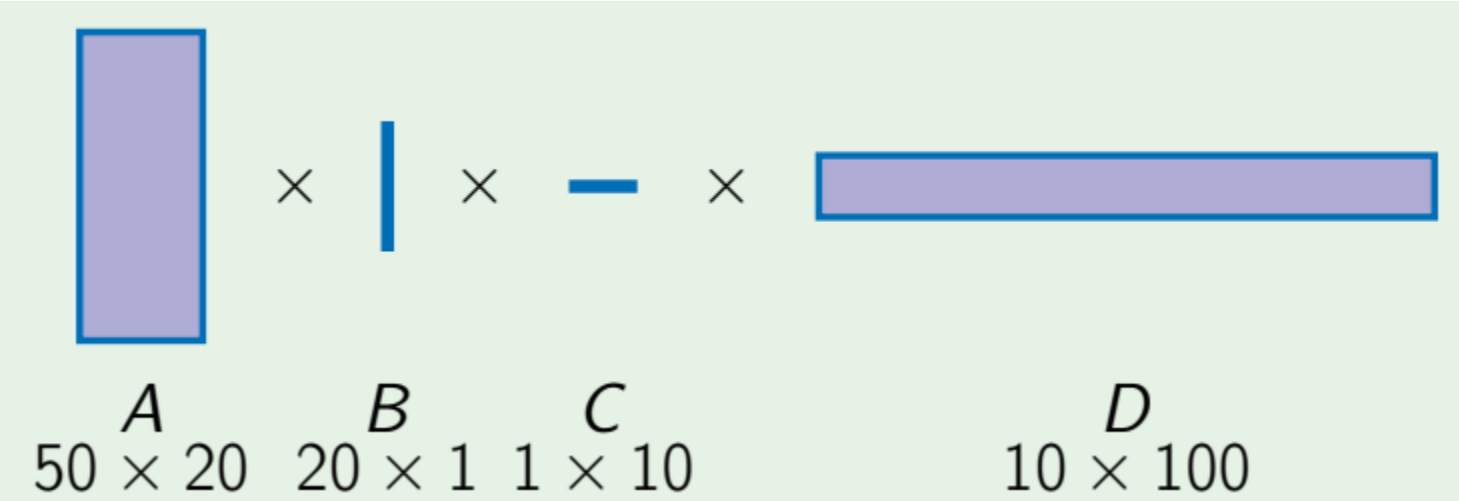
Clarifications

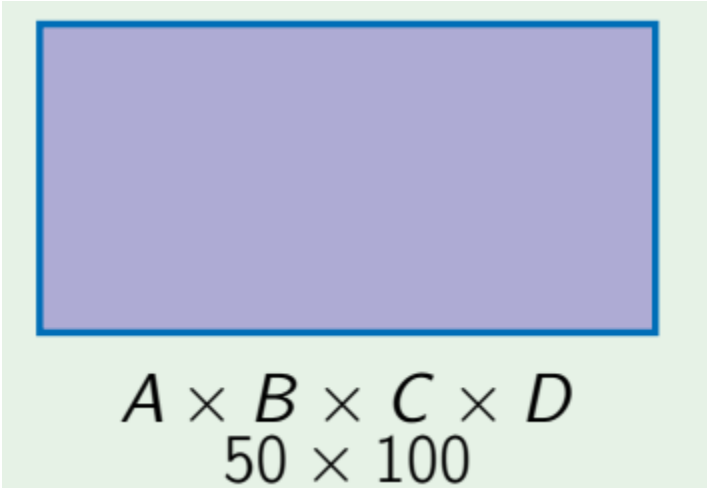
- Denote the sizes of matrices $A_0, \ldots, A_{n-1}$ by

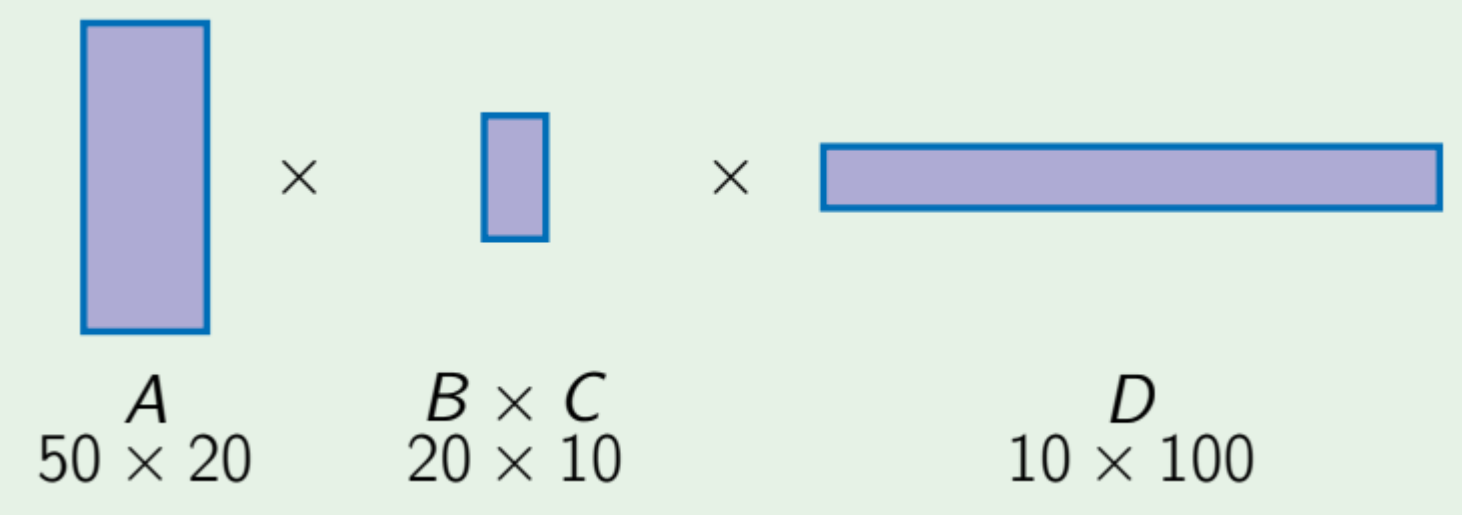
$$m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$$

  respectively. I.e., the size of A_i is $m_i \times m_{i+1}$
- Matrix multiplication is not commutative (in general, $A \times B \neq B \times A$), but it is associative: $A \times (B \times C) = (A \times B) \times C$
- Thus $A \times B \times C \times D$ can be computed, e.g., as $(A \times B) \times (C \times D)$ or $(A \times (B \times C)) \times D$
- The cost of multiplying two matrices of size $p \times q$ and $q \times r$ is pqr

Example: A x ((B x C) x D)



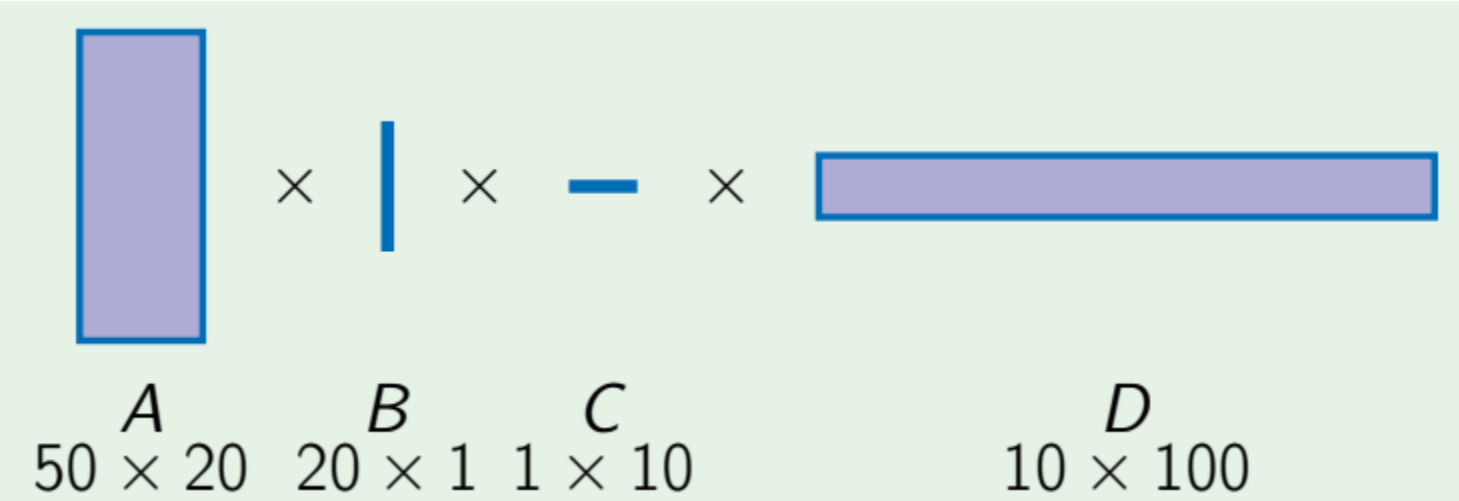


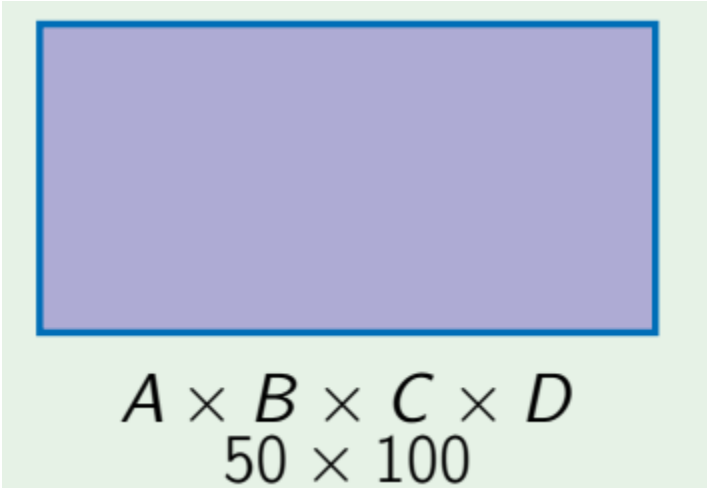


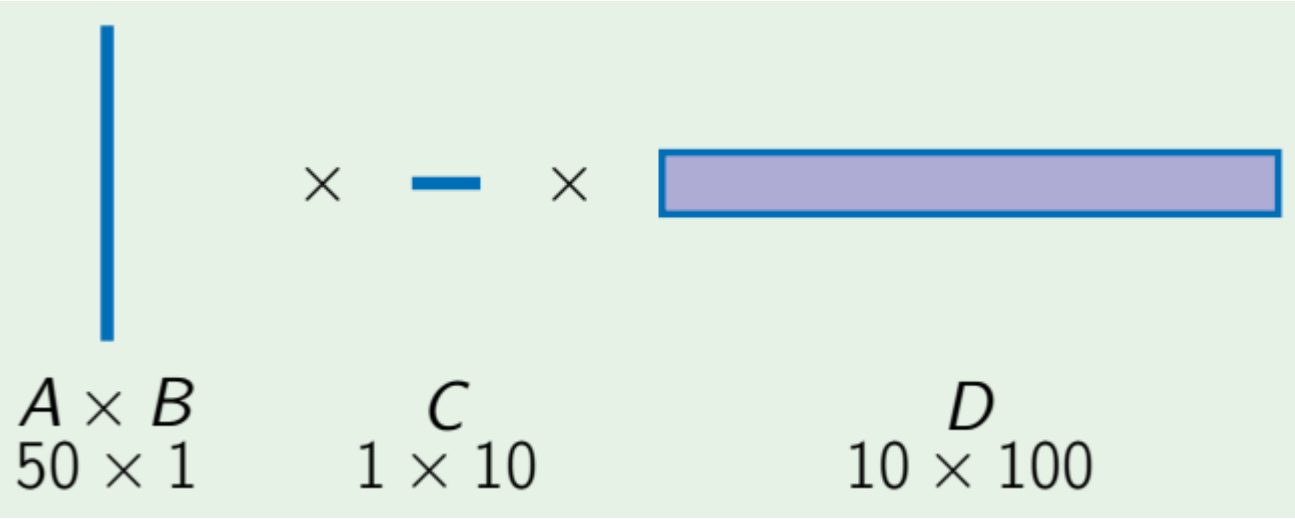
Cost:
20.1.10+20.10.100+50.20.100 = 120200

Cost : 20.1.10

Example: (A x B) x (C x D)



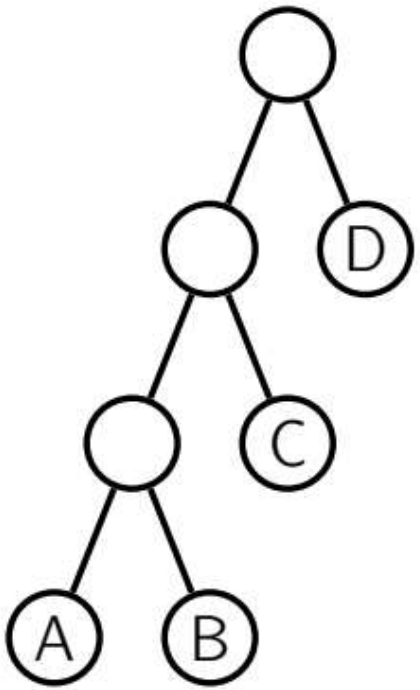




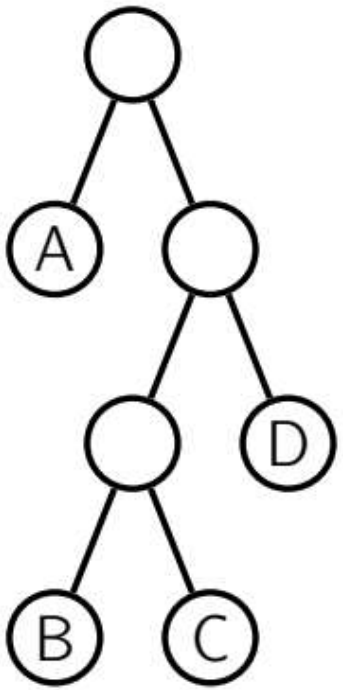
Cost: 50 · 20 · 1 + 1 · 10 · 100 + 50 · 1 · 100 = 7000

Cost : 50.20.1

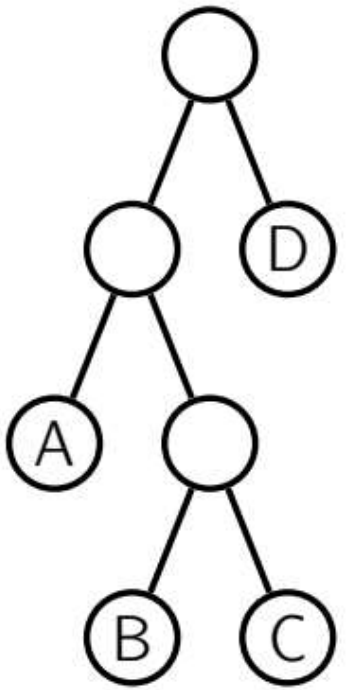
Order as a Full Binary Tree



$((A \times B) \times C) \times D$

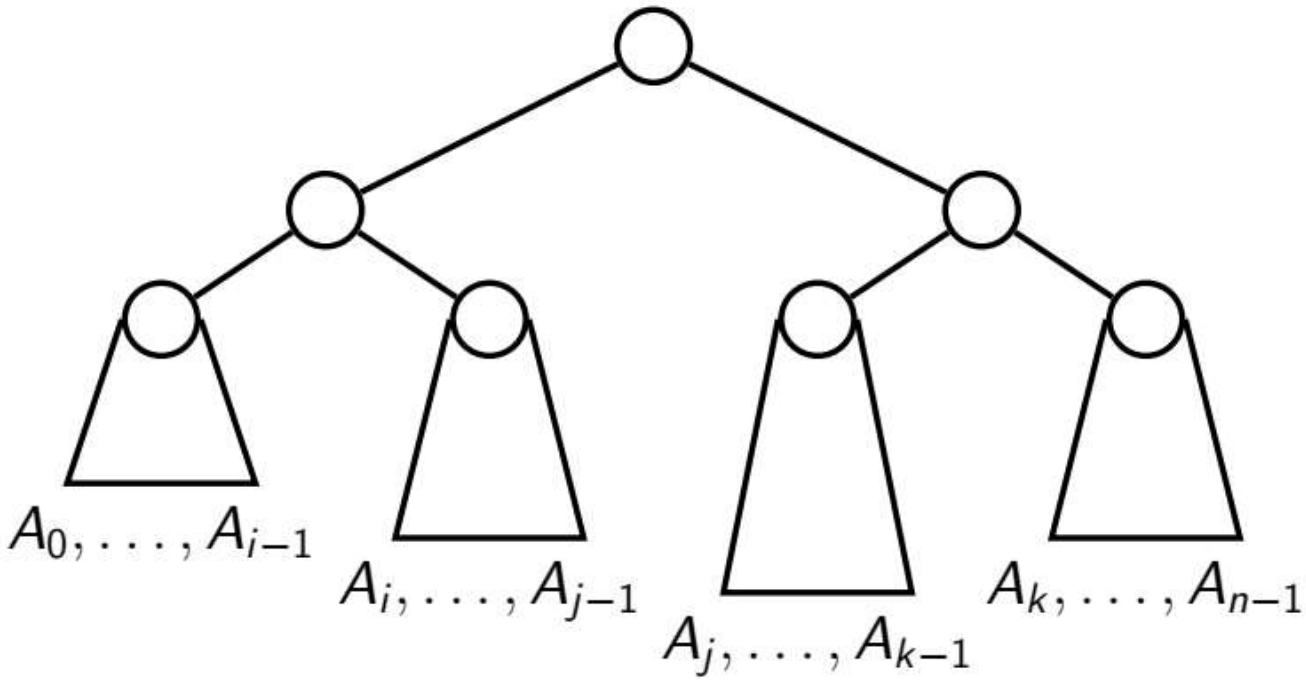


$A \times ((B \times C) \times D)$



$(A \times (B \times C)) \times D$

Analyzing an Optimal Tree



each subtree computes
the product of $A_p, \ldots, A_q$ for some $p \leq q$

Subproblems

- Let $M(i, j)$ be the minimum cost of computing $Ai \times \cdots \times Aj\text{-}1$
- Then

$$M(i,j) = \min_{i<k<j} \{M(i,k) + M(k,j) + m_i \cdot m_k \cdot m_j\}$$

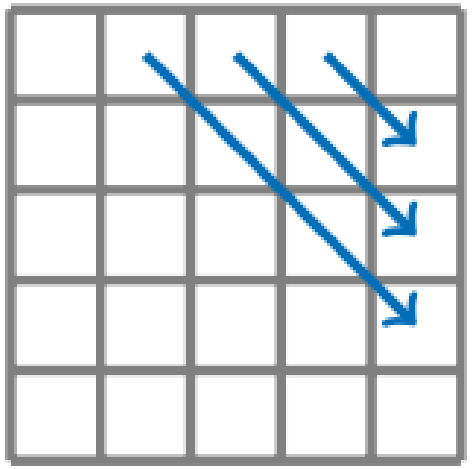
- Base case: $M(i, i + 1) = 0$

Recursive Algorithm

```
T = dict()

def matrix_mult(m, i, j):
  if (i, j) not in T:
    if j == i + 1:
      T[i, j] = 0
    else:
      T[i, j] = float("inf")
      for k in range(i + 1, j):
        T[i, j] = min(T[i, j],
          matrix_mult(m, i, k) +
          matrix_mult(m, k, j) +
          m[i] * m[j] * m[k])

  return T[i, j]

print(matrix_mult(m=[50, 20, 1, 10, 100], i=0, j=4))
```

Converting to an Iterative Algorithm

- We want to solve subproblems going from smaller size subproblems to larger size ones
- The size is the number of matrices needed to be multiplied: $j - I$
- A possible order:



Iterative Algorithm

```
def matrix_mult(m):
  n = len(m) - 1
  T = [[float("inf")] * (n + 1) for _ in range(n + 1)]

  for i in range(n):
    T[i][i + 1] = 0

  for s in range(2, n + 1):
    for i in range(n - s + 1):
      j = i + s
      for k in range(i + 1, j):
        T[i][j] = min(T[i][j],
          T[i][k] + T[k][j] +
          m[i] * m[j] * m[k])

  return T[0][n]

print(matrix_mult(m=[50, 20, 1, 10, 100]))
```

Final Remarks

- Running time: $O(n^3)$
- To unwind a solution, go from the cell (0, n) to a cell (i, $i + 1$)
- Brute force search: recursively enumerate all possible trees

Dynamic Programming - Summary

Step 1 (the most important step)

- Define subproblems and write down a recurrence relation (with a base case)
  - either by analyzing the structure of an optimal solution, or
  - by optimizing a brute force solution

Subproblems: Review

- Longest increasing subsequence: *LIS*(*i*) is the length of longest common subsequence ending at element *A*[*i*]
- Edit distance: *ED*(*i, j*) is the edit distance between prefixes of length *i* and *j*
- Knapsack: *K*(*w*) is the optimal value of a knapsack of total weight *w*
- Chain matrix multiplication *M*(*i, j*) is the optimal cost of multiplying matrices through *i* to *j* - 1

Step 2

- Convert a recurrence relation into a recursive algorithm with memoization:
  - store a solution to each subproblem in a table
  - before solving a subproblem check whether its solution is already stored in the table

Step 3

- Convert a recursive algorithm into an iterative algorithm:
  - initialize the table
  - go from smaller subproblems to larger ones
  - specify an order of subproblems

Step 4

- Prove an upper bound on the running time. Usually the product of the number of subproblems and the time needed to solve a subproblem is a reasonable estimate.

Step 5

- Uncover a solution

Step 6

- Exploit the regular structure of the table to check whether space can be saved

Recursive vs Iterative

- Advantages of iterative approach:
  - No recursion overhead
  - May allow saving space by exploiting a regular structure of the table
- Advantages of recursive approach:
  - May be faster if not all the subproblems need to be solved
  - An order on subproblems is implicit

Assignment

- **Robotruck** UVA
- **Balloons in a box** UVA
- **Copying books** UVA
- 10181 – Puzzle Problem UVA
- 704 – Color Hash UVA
- 10001 – Garden of Eden UVA
- 861- Little Bishops UVA
- 10261 – Ferry Loading UVA
- 116 – Unidirectional TSP UVA
- 10154 – Weights and Measures UVA
- 10131 – Is bigger smarter? UVA

Assignment

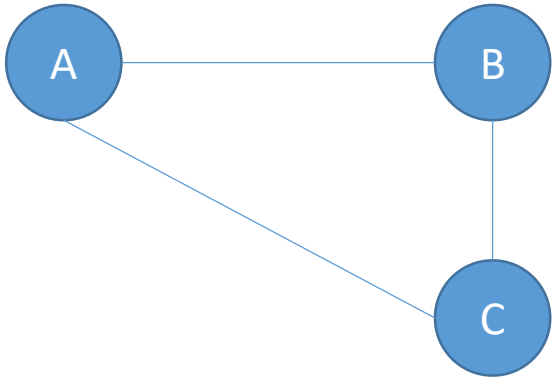
- Quiz6-DynProg2
- assignment-DP2

Graph traversal and paths

Introduction

You all know graphs:

- Set of nodes *N*
- Set of edges $E \subseteq N \times N$
- Edges can be undirected or directed, i.e., $(a; b) \neq (b; a)$
  - N= {A,B, C}
  - E= {(,B), (A,C), (B,C)}



Data Structures

Several options to represent graphs:

- Adjacency matrix:
  - bool G[MAXN][MAXN];
  - G[x][y] is true if an edge between node x and y exists
  - Replace bool by int to represent weighted edges
- Adjacency list:
  - vector<int> Adj[MAXN];
  - y is in Adj[x] if an edge between node x and y exists
  - Pairs to represent weights
- Edge list:
  - vector<pair<int, int> > Edges;
  - Edges contains a pair of nodes if an edge exists between them
- Nodes and edges may also be custom structs or classes

Traversals – Depth-First search (DFS)

Visit each node in the graph once:

- Recursive implementation below
- Manage stack yourself for iterative version
- First visit child nodes then siblings

```
bool Visited[MAXN] = {};
void DFS(int u) {
        if (Visited[u])
                return;
        Visited[u] = true;
        // maybe do something with u (pre-order) ...
        for (auto v : Adj[u])
                DFS(v);
        // or do something here (post-order)
}
```

Application of DFS

- Determine a topological order of nodes
  - Only works for graphs without cycles (i.e., x ! y ! z ! x)
  - Add visited node at the head of an ordered list (at the end of DFS: ordering.push_front(u))
- Detect if a cycle exists:
  - Check if the currently visited node is on the stack
  - A) Use three states for Visited array: UNVISITED, ONSTACK, VISITED
  - B) Explicitly search in the stack of the iterative algorithm
- **Examples: https://visualgo.net/dfsbfs**

Traversals – Breadth-First search (BFS)

Visit each node in the graph once:

- Similar to DFS, but replaces stack by queue

```
queue<int> Q;
bool Visited[MAXN] = {};
void BFS(int root) {
        Q.push(root);
        while (!Q.empty()) {
                int u = Q.front();
                Q.pop();
                if (Visited[u])
                        continue;
                Visited[u] = true;
                for (auto v : Adj[u])
                        Q.push(v); // usually do something with v
                }
        }
```

Application of BFS

Shortest path search

- Stop processing when a given node d was found
- Minimizes number of hops, i.e., all edges have same weight
- Generalization follows next

Examples: https://visualgo.net/dfsbfs

Finding paths - Dijkstra

BFS can only be used if edge weights are uniform

- Dijkstra's algorithm generalizes this
- Constraint: all edges need to have non-negative weights
- Use a priority queue to choose which node to examine next
  - Would require a function to update the priority of an element
    - Would need to update order in the priority queue
  - We'll use the standard priority_queue of STL
    - Simply insert a new element in the queue (no update)
    - Ok since priorities decrease monotonically
    - This slightly diverges from Dijkstra's algorithm
- May revisit nodes several times
- **Example: https://visualgo.net/sssp**

Finding paths – Dijkstra Algorithm

```
unsigned int Dist[MAXN];
typedef pair<unsigned int, int> WeightNode; // weight goes first
priority_queue<WeightNode, std::vector<WeightNode>, std::greater<WeightNode> > Q; //queue with smallest weight at the top
void Dijkstra(int root) {
    fill_n(Dist, MAXN, MAXLEN);
    Dist[root] = 0;
    Q.push(make_pair(0, root));
    while (!Q.empty()) {
        int u = Q.top().second; // get node with least priority
        Q.pop();
        for (auto tmp : Adj[u]) {
            int v = tmp.second;
            unsigned int weight = tmp.first;
            if (Dist[v] > Dist[u] + weight) { // shorter path found?
                Dist[v] = Dist[u] + weight;
                Q.push(make_pair(Dist[v], v)); // simply push, no update here
            }
        }
    }
}
```

Finding paths – Bellman-Ford

- Dijkstra’s algorithm is limited to non-negative edge weights
- Bellman-Ford extends this to general edge weights
- Constraint: no cycle with negative total costs
- May again revisit nodes several times
- **Example: https://visualgo.net/sssp**

Finding paths – Bellman-Ford Algorithm

```
unsigned int Dist[MAXN];
void BellmanFord(int root) {
        fill_n(Dist, MAXN, MAXLEN);
        Dist[root] = 0;
        for(int u=0; u < N - 1; u++) {
// just iterate N - 1 times
                for (auto tmp : Adj[u])
                {
                        int v =
tmp.second; // same as Dijkstra
before
                                unsigned int
weight = tmp.first;
                                Dist[v] =
min(Dist[v], Dist[u] + weight);
                        }
                }
}
```

Finding paths – Floyd-Warshall

Dijkstra and Bellman-Ford compute shortest paths

- From a single source (root)
- To all other (reachable) nodes
- This is known as: single-source shortest path problem

Floyd-Warshall extends this to compute the shortest paths between all pairs of nodes

- This is known as: all-pairs shortest path problem

Finding paths – Floyd-Warshall Algorithm

```
int Dist[MAXN][MAXN];
void FloydWarshall() {
    fill_n((int*)Dist, MAXN*MAXN, MAXLEN);
    for(int u=0; u < N; u++) {
        Dist[u][u] = 0;
        for (auto tmp : Adj[u])
            Dist[u][tmp.second] = tmp.first;
    }
    for(int k=0; k < N; k++) // check sub-path combinations
        for(int i=0; i < N; i++)
        for(int j=0; j < N; j++) // concatenate paths
            Dist[i][j] = min(Dist[i][j], Dist[i][k] + Dist[k][j]);
}
```

Keeping track of the path

We only considered the length of the path so far:

- All of the above algorithms can track the actual shortest path
- This allows to print each edge/node along the path
- Basic idea:
  - Introduce an array int Predecessor[MAXN]
  - (Use two-dimensional array for Floyd-Warshall)
  - Updated whenever Dist[v] changes
  - Simply set to the new predecessor u

Heuristics – A* Search

- Heuristics may speed-up the path search
  - Bellman-Ford and Floyd-Warshall equally explore all possibilities
  - Dijkstra prefers nodes with shorter distance
  - Basic idea behind A* Search:
    - Extension to Dijkstra's algorithm
    - Introduce an estimation of the remaining distance
    - Prefer nodes with minimal estimated remaining distance
  - Advantages
    - May converge faster than Dijkstra
    - Can be used to compute approximate solutions (trading speed for precision)

Eulerian Circuits

We study undirected graphs and assume they are connected:

- Eulerian path:

  Use every edge of a graph exactly once. Start and end may differ

- Eulerian circuit:

  Use every edge exactly once. Start and end at the same node

- Conditions to find Eulerian path:
  - All nodes have even degree or
  - Precisely two nodes have odd degree
- For Eulerian circuit, all nodes must have even degree
- https://www-m9.ma.tum.de/graph-algorithms/hierholzer/index_en.html

Hierholzer's Algorthm for Eulerian paths (assuming they exist)

```
set<int> Adj[MAXN]; vector<int> Circuit;
void Hierholzer() {
    int v = 0; // find node with odd degree, else start with node 0
    for (int u=0; u < N && v == 0; u++)
        if (Adj[u].size() & 1)
            v = u; // node with odd degree
    stack<int> Stack;
    Stack.push(v);
    while (!Stack.empty()) {
        if (!Adj[v].empty()) { // follow edges until stuck
            Stack.push(v);
            int tmp = *Adj[v].begin();
            Adj[v].erase(tmp); // remove edge, modifying graph
            Adj[tmp].erase(v);
            v = tmp;
        } else { // got stuck: stack contains a circuit
            Circuit.push_back(v); // append node at the end of circuit
            v = Stack.top(); // backtrack using stack, find larger circuit
            Stack.pop();
        }
    }
}
```

Assignment

- 10067 Playing with Wheels UVA
- 10099 The Tourist Guide UVA
- 10051 Tower of Cubes UVA
- 10187 From Dusk Till Dawn UVA
- **1056 - Degrees of separation** UVA
- **1130- Men at work** UVA
- **12144 - Almost Shortest Path** UVA

Advanced graph algorithms

Union-Find

A data structure to track equivalence relations between elements:

- Elements are partitioned into non-overlapping sets
  - Initially only pairwise relations are known
    - (i.e., X and Y are in the same set)
  - From pairwise relations, deduce the global partitioning step-wise
- Basic idea:
  - Represent partitions as trees
  - Merge trees when a new pairwise relation is discovered

Union-Find

Two operations to update/query the union and data structure:

- Union(x,y):
  - Add a new pairwise relation between x and y and update the Union-Find structure to put them in the same set
- Find(x):
  - Get the (current) representative of the set for element x

https://visualgo.net/ufds

Union-Find using Ranks and Path Compression

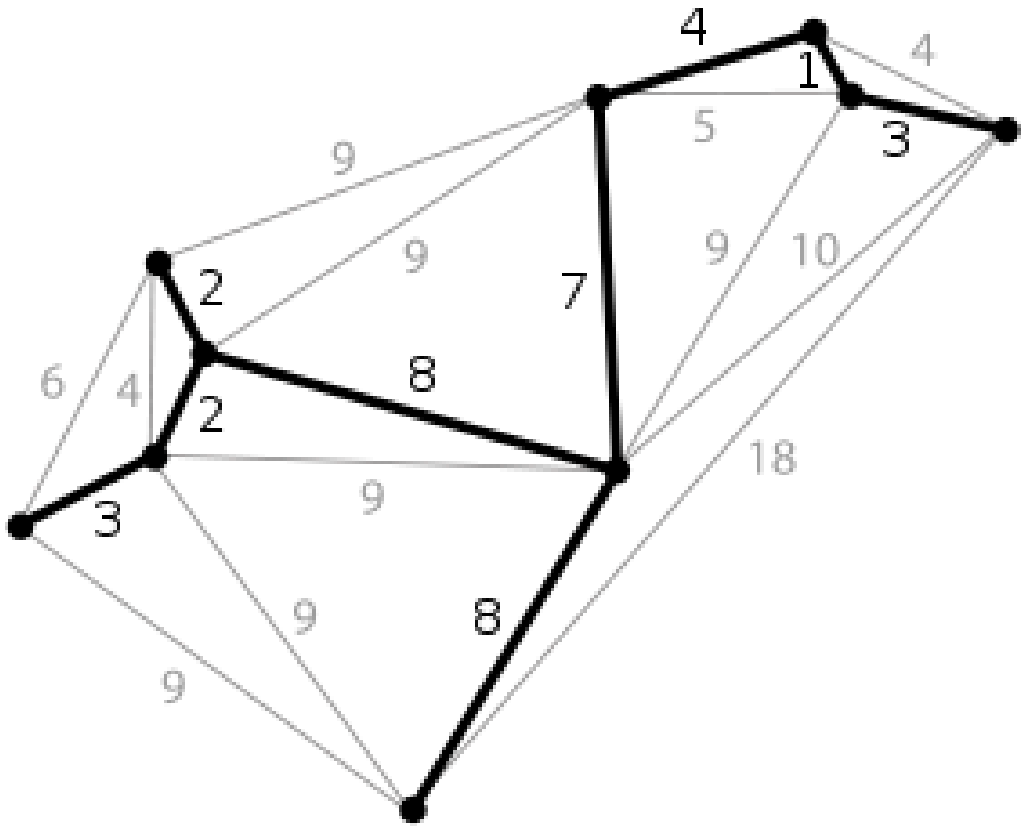
```
map<int, pair<int, unsigned int> > Sets; // map to parent & rank
void MakeSet(int x) {
    Sets.insert(make_pair(x, make_pair(x, 0)));
}
int Find(int x) {
    if (Sets[x].first == x) return x; // Parent == x ?
    else return Sets[x].first = Find(Sets[x].first); // Get Parent
}
```

```
void Union(int x, int y) {
    int parentX = Find(x), parentY = Find(y);
    int rankX = Sets[parentX].second, rankY = Sets[parentY].second;
    if (parentX == parentY) return;
    else if (rankX < rankY)
        Sets[parentX].first = parentY;
    else
        Sets[parentY].first = parentX;
    if (rankX == rankY)
        Sets[parentX].second++;
}
```

Spanning Trees

- Subgraph of undirected connected graph that forms a tree
- The subgraph contains each node of the original graph
- Computing spanning trees
  - Many possible spanning trees exist
  - Any depth-first traversal gives a spanning tree
- In weighted graphs one might need a minimum spanning tree
  - A spanning tree as defined above
  - Require that the total sum of edge weights is minimal
    - (i.e., no other spanning tree with a lower total sum exists)
- Example: https://visualgo.net/mst

Minimum Spanning Trees



https://commons.wikimedia.org/wiki/File:Minimum_spanning_tree.svg

Kruskal's Algorithm

```
vector<pair<int, pair<int, int> > > Edges;

set<pair<int,int> > A; // Final minimum spanning tree

void Kruskal() {
    for(int u=0; u < N; u++)
    MakeSet(u); // Initialize Union-Find
    sort(Edges.begin(), Edges.end()); // Sort edges by weight
    for(auto tmp : Edges) {
        auto edge = tmp.second;
        if (Find(edge.first) != Find(edge.second))
        {
            Union(edge.first, edge.second); // update Union-Find
            A.insert(edge); // include edge in MST
        }
    }
}
```

Flow Networks

- Flow networks are weighted directed graphs
- Edge weights denote the capacity of edges
  - Current in an electric circuit
  - Water in pipes
  - Trains on a railroad
- Question: What is the maximum flow between nodes s and t?
  - Assign flow to each edge respecting the edge's capacity
  - For each node, the combined in/out flows must be equal

Maximum Flow / Minimum Cut

Maximum flow is limited by a cut separating s and t

- Basic idea behind cuts:
  - Partition the network's nodes into two sets S and T
  - S contains s while T contains t
  - Edges (u; v) with u ∈ S and v ∈ T are in the cut between S and T
  - The edges in the cut completely separate the two sets
  - => Removing those edges gives a maximum flow of zero
- Link between cuts and flows:
  - The combined edge weights of a cut bound the flow from s to t
  - The **maximum flow** is thus limited by a **minimum cut**
- **Example:** https://visualgo.net/maxflow

Ford-Fulkerson Algorithm

```
// find path from s to t in G, return true if such a path exists
bool DFS(int G[MAXN][MAXN], int s, int t, int Predecessor[MAXN]);
int FordFulkerson(int G[MAXN][MAXN], int s, int t) {
    int GRes[MAXN][MAXN]; // residual graph
    copy_n((int*)G, MAXN*MAXN, (int*)GRes); // copy original graph
    int Predecessor[MAXN];
    int Maxflow = 0;
    while (DFS(GRes, s, t, Predecessor)) { // find residual path
        int Bottleneck = MAXFLOW; // get minimal flow of residual path
        for (int v = t; v != s; v = Predecessor[v])
            Bottleneck = min(Bottleneck, GRes[Predecessor[v]][v]);
        for (int v = t; v != s; v = Predecessor[v]) {
            // decrease capacity along residual path
            GRes[Predecessor[v]][v] -= Bottleneck;
            GRes[v][Predecessor[v]] += Bottleneck;
        }
        Maxflow += Bottleneck;
    }
    return Maxflow;
}
```

Assignment Problems and Matchings

- Represented as bipartite graphs:
  - Nodes are partitioned into two disjoint sets X and Y
  - Edges always connect nodes from both sets ($G = (X \cup Y, E)$, where $E \subseteq X \times Y$)
- Basic idea:
  - Search the best assignment of elements in X to elements in Y
  - Each element may appear only in one assignment
- Problem variants
  - Maximize matching cardinality (Hopcroft-Karp – on next slide)
  - Maximize matching cost in weighted graphs

Hopcroft-Karp (data structures)

```
// Artificial node (unused otherwise) -- end of augmenting path
#define NIL 0

// "Infinity", i.e., value larger than min(|X|, |Y|)
#define INF numeric_limits<unsigned int>::max()

// Partitions X and Y
vector<int> X, Y;

// Neighbors in Y of nodes in X
vector<int> Adj[MAXX];

// Matching X-Y and Y-X
int PairX[MAXX];
int PairY[MAXY];

// Augmenting path lengths
unsigned int Dist[MAXX];
```

Hopcroft-Karp (main)

```
int HopcroftKarp() {
    fill_n(PairX, X.size(), NIL); // initialize: empty matching
    fill_n(PairY, Y.size(), NIL);
    int Matching = 0; // count number of edges in matching
    while (BFS()) { // find all shortest augmenting paths
    for(auto x : X) // update matching cardinality
    if (PairX[x] == NIL && DFS(x)) //  node
// not yet in matching? && does an
// augmenting path start at x?
        Matching++;
}
return Matching;
}
```

Hopcroft-Karp (BFS)

```
bool BFS() {
    queue<int> Q;
    Dist[NIL] = INF;
    for(auto x : X) { // start from nodes that are not yet matched
        Dist[x] = (PairX[x] == NIL) ? 0 : INF;
        if (PairX[x] == NIL)
            Q.push(x);
    }
    while (!Q.empty()) { // find all shortest paths to NIL
        int x = Q.front(); Q.pop();
        if (Dist[x] < Dist[NIL]) // can this become a shorter path?
        for (auto y : Adj[x])
            if (Dist[PairY[y]] == INF) {
                Dist[PairY[y]] = Dist[x] + 1; // update path length
                Q.push(PairY[y]);
            }
    }
    return Dist[NIL] != INF; // any shortest path to NIL found?
}
```

Hopcroft-Karp (DFS)

```
bool DFS(int x) {
    if (x == NIL)
        return true; // reached NIL
    for (auto y : Adj[x])
        if (Dist[PairY[y]] == Dist[x] + 1 &&
        DFS(PairY[y])) { // follow trace of BFS
            PairX[x] = y; // add edge from x to y to
            matching
            PairY[y] = x;
            return true;
        }
    Dist[x] = INF;
    return false; // no augmenting path
    found
}
```

Assignment

- **The Necklace** UVA
- **Buy or Build** UVA
- **Taxi Cab Scheme** UVA
- 10034 Freckles UVA
- 10199 Tourist Guide UVA
- 10249 The Grand Dinner UVA

String manipulation

String representation

- *Character codes* are mappings between numbers and the symbols which make up a particular alphabet.
- ASCII : $2^7 = 128$ characters on eight bit. First bit is always 0

0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT	12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	DEL

String representation

- All non-printable characters have either the first three bits as zero or all seven lowest bits as one. This makes it very easy to eliminate them before displaying junk, although somehow very few programs seem to do so.
- Both the upper- and lowercase letters and the numerical digits appear sequentially. Thus we can iterate through all the letters/digits simply by looping from the value of the first symbol (say, “a”) to value of the last symbol (say, “z”).
- Another consequence of this sequential placement is that we can convert a character (say, “I”) to its rank in the collating sequence (eighth, if “A” is the zeroth character) simply by subtracting off the first symbol (“A”).
- We can convert a character (say “C”) from upper- to lowercase by adding the difference of the upper and lowercase starting character (“C”-“A”+“a”). Similarly, a character x is uppercase if and only if it lies between “A” and “Z”.
- Given the character code, we can predict what will happen when naively sorting text files. Which of “x” or “3” or “C” appears first in alphabetical order? Sorting alphabetically means sorting by character code. Using a different collating sequence requires more complicated comparison functions.

String representation

Strings can be represented as:

- Null terminated Arrays
  - C\C++ representation.
  - Allocate enough memory for the null character
  - Strings can be treated as arrays
- Array pus length
  - Java representation
  - First location in the array stores the length of the string
- Linked lists of characters
  - High overhead
  - Can be useful if insertion or deletion of substrings are frequent

Substring/pattern search

Simple algorithm:

```
// s is the string,  p is the pattern
for (int i=0, j; i < s.size() - p.size() + 1; ++i) {
        for (j = 0; j < p.size() && s[i+j] == p[j]; ++j)
                ;
        if (j == p.size())
                printf("Match at position %d\n", i);
}
```

- Similar to the native implementation of the language (strstr or str::find)
- O(|s|x|p|)

Substring/pattern search

- **Knuth–Morris–Pratt**
  - String “ABBA” prefixes: {A, AB, ABB, ABBA}, Proper prefixes: {A, AB, ABB}, Suffixes {ABBA, BBA, BA, A}
  - Largest proper prefix of "ABBA" which is also a suffix of "ABBA" is A with length 1.
  - For a pattern p of length m, build lps[i] for i=0 to m-1 where lps[i] is the largest prefix of p[0..i] which is also a proper suffix of p[0..i] (or largest proper prefix of p[0..i] which is also a suffix of p[0..i] or largest prefix of p which is also a proper suffix of p[0..i])

For the pattern “AAAA”,	For the pattern “ABCDE”,	For the pattern “AABAACAABAA”	For the pattern “AAACAAAAAC”	For the pattern “AAABAAA”,
lps[] is [0, 1, 2, 3]	lps[] is [0, 0, 0, 0, 0]	lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]	lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]	lps[] is [0, 1, 2, 0, 1, 2, 3]

Substring/pattern search

- **Knuth–Morris–Pratt - Build LPS**

```
char p[MAXN]; //the pattern
int lps[MAXN+1]; // we use an extra slot in lps to reset it and lps is shifted by  i.e. for a pattern
// p of length m, build lps[i] for i=1 to m where lps[i] is the largest proper prefix of p[0..i-1] which is also
//a suffix of  p[0..i-1]
int np = strlen(p);
lps[0] = -1;
int cnd = 0;// the zero-based index in p of the next character of the current candidate substring
for (int i = 1; i <= np; i++) {// i is the current position we are computing in lps
    lps[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = lps[cnd];//when there is a mismatch, fall back on the previously known prefix
        // match suffix
    cnd++; // increment as the prefix matches the suffix
}
```

Substring/pattern search

- **Knuth–Morris–Pratt – Search the substring**

```
char p[MAXN]; int lps[MAXN+1];
int np = strlen(p), ns = strlen(s);
//p pattern, s string being searched
[...] // Build the lps table here
int cnd = 0; // current position in pattern p
for (int i = 0; i <= ns; i++) { // while you do not read
    while (cnd >= 0 && p[cnd] != s[i]) // the next char in p
        cnd = T[cnd]; // you go back in p
    cnd++; // now move forward as the pattern char matches the string char
    if (cnd == np) {
    // We reached the end of p. A match is found
    printf("match at %d\n", i - np + 1);
    cnd = T[cnd]; // we go back in p in case the next match overlaps
    }
}
```

Substring/pattern search

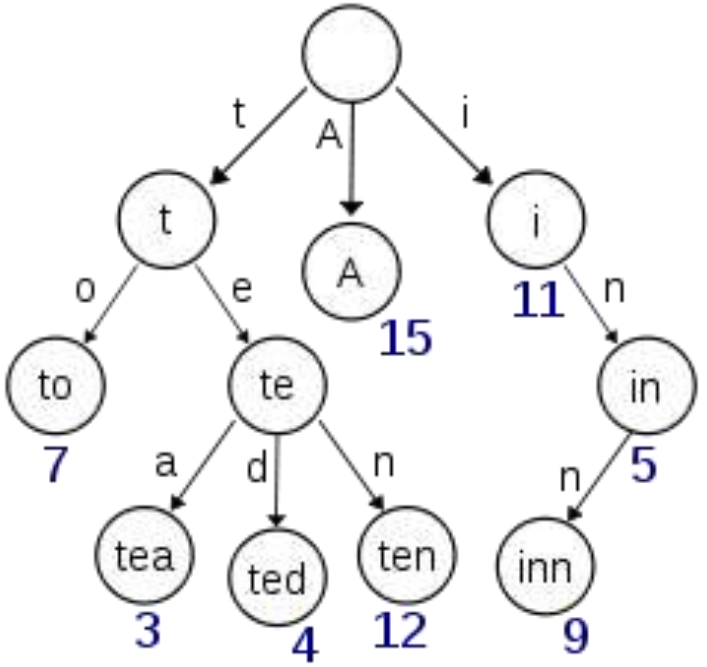
- **Knuth–Morris–Pratt**
  - Build LPS O(np)
  - Search O(ns)
  - O(np+ns) in total

Finding a word in a set of word

- Dictionary of n words
- To find if a word belons to the dictionary
  - Vector or forward list: O(n)
  - Set (binary balanced tree): O(log(n))
  - Hashmap (unordered_set): O(1) but O(n) in the worst case
  - Trie or (**digital tree**, **radix tree** or **prefix tree**): O(1)

Trie

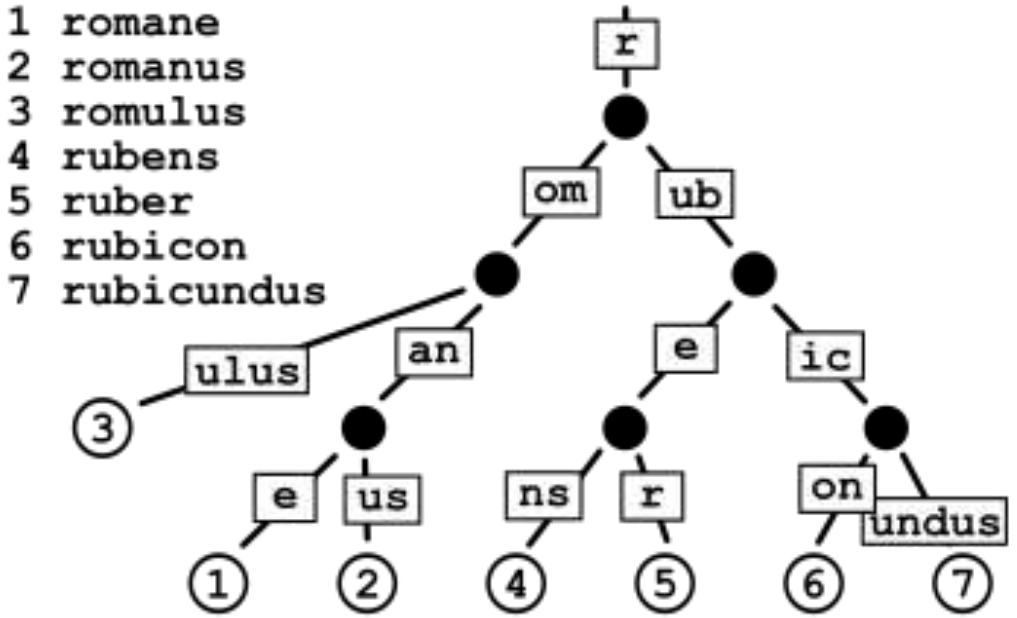
- Tree of all prefixes of words in Dictionary D
- The edges are labelled by letters
- On each node, you indicate the ID in D of the word at that node.
- Can find all words in D which have m as prefix



A trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn". Note that this example does not have all the children alphabetically sorted from left to right as it should be (the root and node 't').

Radix Tree

- space-optimized trie (prefix tree) in which each node that is the only child is merged with its parent.

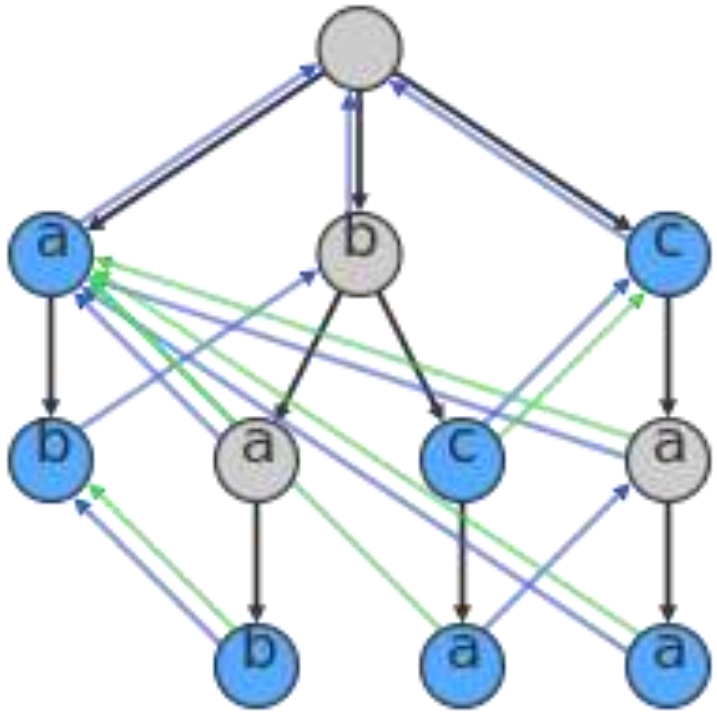


Find all elements of a finite set of strings (the "dictionary") within an input text

Aho-Corasick

- Build a trie of the dictionarythat has one node for every prefix of every string in the dictionary. So if (bca) is in the dictionary, then there will be nodes for (bca), (bc), (b), and (). If a node is in the dictionary then it is a blue node. Otherwise it is a grey node.
- There is a black directed "child" arc from each node to a node whose name is found by appending one character.
- There is a blue directed "suffix" arc from each node to the node that is the longest possible strict suffix of it in the graph
- There is a green "dictionary suffix" arc from each node to the next node in the dictionary that can be reached by following blue arcs
- Generalize Knuth–Morris–Pratt to an arbitrary trie (not a sequence of char): $O(n * k + l + m)$

Find all elements of a finite set of strings (the "dictionary") within an input text



Dictionary {a, ab, bab, bc, bca, c, caa}

Path	In dictionary (blue vs grey)	Suffix link (blue)	Dict suffix link (green)
()	–		
(a)	+	()	
(ab)	+	(b)	
(b)	–	()	
(ba)	–	(a)	(a)
(bab)	+	(ab)	(ab)
(bc)	+	(c)	(c)
(bca)	+	(ca)	(a)
(c)	+	()	
(ca)	–	(a)	(a)
(caa)	+	(a)	(a)

Aho-Corasick – Build the trie

```
int nw; // number of words in the dictionary
char w[MAXW][MAXL]; // words in the dictionary – word i of content w[i]
int trie[MAXW*MAXL+2][ALPHA]; // trie of the words in the dictionary
int fs; // next free index in the trie
int endw[MAXW*MAXL+2]; // word which ends at this index of the trie
// insert the word number i of content s in the trie
// starting from the node of index n in the trie
void insert_trie(int i, char *s, int n)
{
    unsigned char x = s[0];
    if (!x) { endw[n] = i; return; } // reached the end f the word
    if (!trie[n][x]) // if there is no edge labeled x from n, ...
        trie[n][x] = fs++; // ... Create a new target node
    return insert_trie(i, s+1, trie[n][x]);
}
```

Aho-Corasick – Beginning

```
void aho_corasick() {
    // Index 0 in the trie indicates non existing nodes
    // the root of the trie is 1
    // Therefore, initially the next free index is 2
    fs = 2;
    // Words are numbered from 1
    // 0 in endw indicates that no word terminates
    for (int i = 1; i <= nw; i++)
    insert_trie(i, w[i], 1); // insert each word in the
    trie
    [...] // here, compute pointers and shortcuts
}
```

Aho-Corasick – Computing pointers and shortcuts – part 1

```
int pointer[MAXW*MAXL+2]; // pointers (in blue), cf lps ofKMP
int shortcut[MAXW*MAXL+2]; // shortcuts (in green)
queue<int> q;
q.push(1); // explore in BFS from the root of the trie
while (!q.empty()) {
    int n = q.front();
    q.pop();
    for (unsigned char x = 0; x < ALPHA; x++) {
        int n2 = trie[n][x];
        if (!n2)
        continue; // no edge for this letter from n
        [...] // process n2 – see next slide
        q.push(n2); // continue BFS with n2
    }
}
```

Aho-Corasick – Computing pointers and shortcuts – part 2

```
// node n2 with parent n and label x, i.e., n -x-> n2

// == compute the pointer ==

int p = pointer[n]; // p is the pointer to n

while (p && !trie[p][x]) // while p has no edge x...
    p = pointer[p]; // ... Backtrack p via the pointer

if (!p) // if p = 0 we are out of the trie ...
    pointer[n2] = 1; // ... So we set pointer=root

else // otherwise the pointer is ...
    pointer[n2] = trie[p][x]; // ...the x-successor of p

// == compute the shortcut==

if (endw[pointer[n2]]) // if a word ends in n2...
    shortcut[n2] = pointer[n2]; // ... Shortcut = pointer

else // otherwise the shortcut is the pointer's one
    shortcut[n2] = shortcut[pointer[n2]];
```

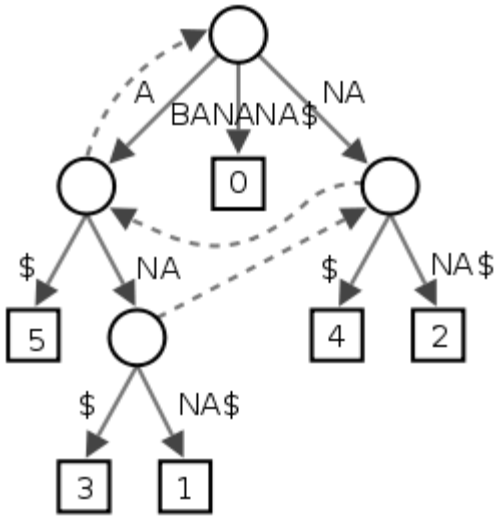
Aho-Corasick – Usage

```
int pos = 1; // current position in the trie
char s[MAXS]; int ns = strlen(s); // string to be searched
for (int i = 0; i < ns; i++) {
    unsigned char x = s[i]; // x = new char read
    while (pos && !trie[pos][x]) // while there is no edge x from p...
        pos = pointer[pos]; // ... Backtrack pos along the pointer
    pos = trie[pos][x]; // now try to move foward via x
    if (!pos) // if we are out of the trie...
        pos = 1; // ... Come back to the root
    int posb = pos; // we are going to enumerate the possibles matches from pos...
    do { // end of word possible in I
        if (endw[posb]) // we indeed have a word end at posb (in i):
            printf("match of %s at %d\n", // display
                w[endw[posb]], i - strlen(endw[posb]) + 1); // the match
        posb = shortcut[posb]; // follow the shortcut from posb
    } while (endw[posb]); // ... While words are ending.
}
```

Suffix tree

- A suffix tree (also called PAT tree or, in an earlier form, position tree) is a compressed trie containing all the suffixes of a given text as their keys and positions in the text as their values.
- Construction for S takes the length of S.
- Usages
  - Locating a substring in S
  - Locating a substring if a certain number of mistakes are allowed
  - Locating matches for a regular expression pattern
  - Longest common substring problem

Suffix tree



Suffix tree for the text BANANA. Each substring is terminated with special character $. The six paths from the root to the leaves (shown as boxes) correspond to the six suffixes A$, NA$, ANA$, NANA$, ANANA$ and BANANA$. The numbers in the leaves give the start position of the corresponding suffix. Suffix links, drawn dashed, are used during construction.

Regular Expressions

- Regular expressions describe patterns to be searched in a string
  - For example: (a|b)*#(a|b)*(#(a|b|#)*)?
- Std::regex in C++11

Huffman Algorithm

- Objective
  - Given A set of symbols s_i and their weights w_i (usually proportional to probabilities).
  - Find a prefix-free binary code (a set of codewords C_i coded binarily) with minimum expected codeword length (equivalently, a tree with minimum weighted path length from the root) i.e. minimize $\sum_i |C_i| * w_i$.
    - Intuitively more probable words should have smaller codes
- Principle of solution
  - Use a decision tree to represent the prefix-free codewords
  - Create a binary tree of nodes. Store the nodes in a regular array, the size of which depends on the number of symbols, n. A node can be either a leaf node or an internal node.

Huffman Algorithm - Principle

- Initially, all nodes are leaf nodes, which contain the symbol itself, the weight (frequency of appearance) of the symbol and optionally, a link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node.
- Internal nodes contain a weight, links to two child nodes and an optional link to a parent node.
- As a common convention, bit '0' represents following the left child and bit '1' represents following the right child.
- The process begins with the leaf nodes containing the probabilities of the symbol they represent.
- Then, the process takes the two nodes with smallest probability, and creates a new internal node having these two nodes as children. The weight of the new node is set to the sum of the weight of the children.
- We then apply the process again, on the new internal node and on the remaining nodes (i.e., we exclude the two leaf nodes), we repeat this process until only one node remains, which is the root of the Huffman tree.

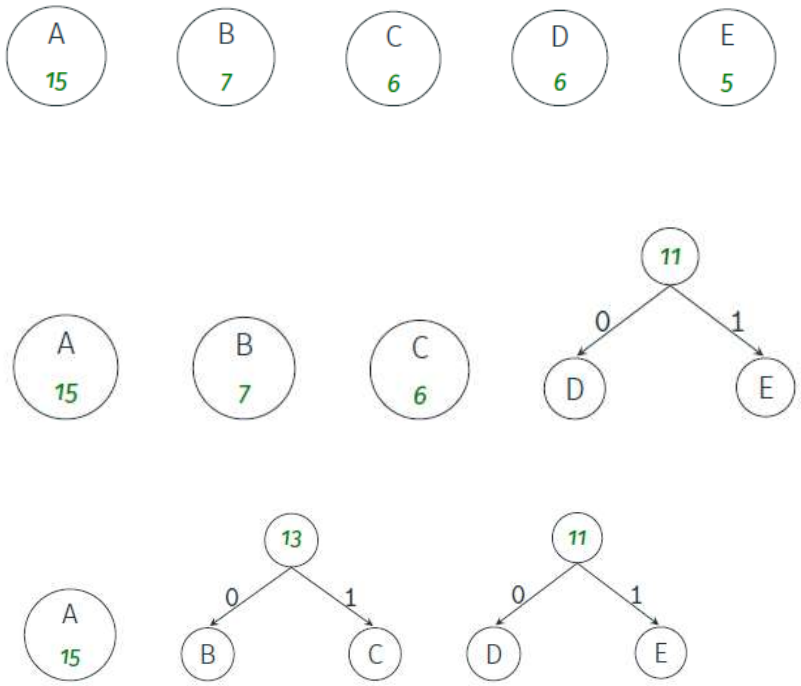
Huffman Algorithm - Principle

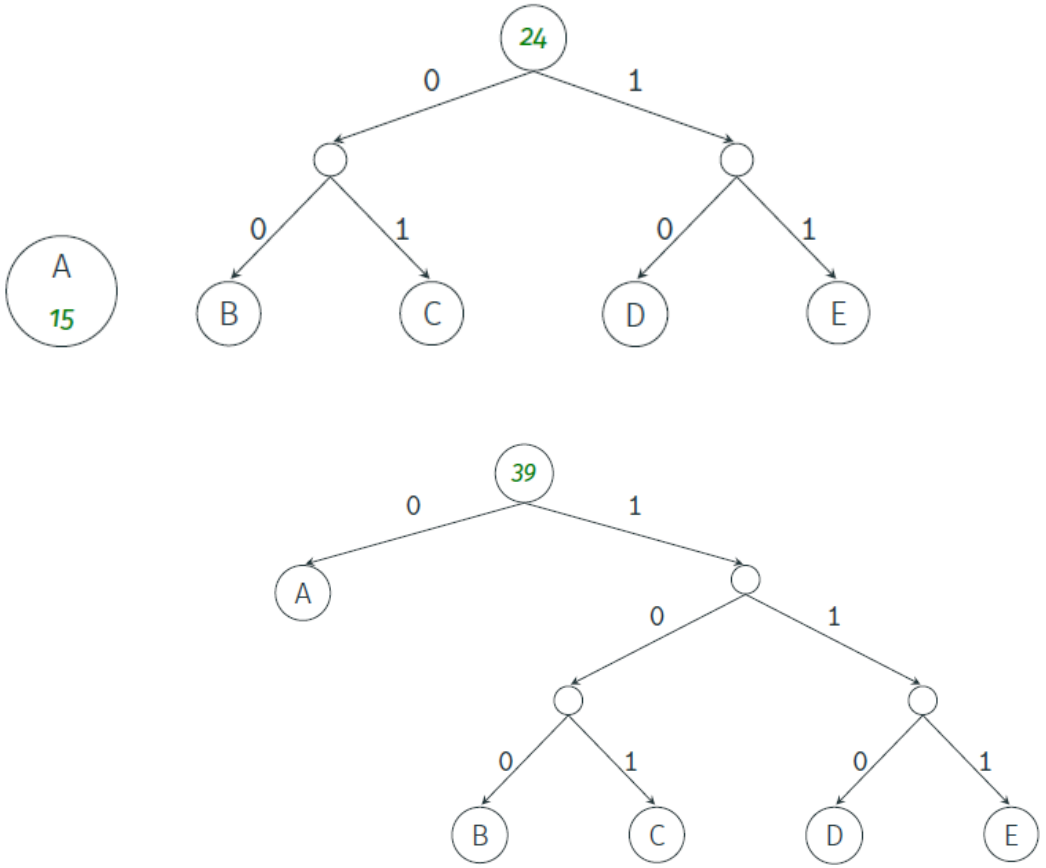
The simplest construction algorithm uses a priority queue where the node with lowest probability is given the highest priority:

- Create a leaf node for each symbol and add it to the priority queue.
- While there is more than one node in the queue:
- Remove the two nodes of highest priority (lowest probability) from the queue
- Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
- Add the new node to the queue.
- The remaining node is the root node and the tree is complete.

O(n log n), where n is the number of symbols.

Huffman algorithm - example





C++ String Library Functions

- string::size() /* string length */
- string::empty() /* is it empty */
- string::c_str() /* return a pointer to a C style string */
- string::operator [](size_type i) /* access the ith character */
- string::append(s) /* append to string */
- string::erase(n,m) /* delete a run of characters */
- string::insert(size_type n, const string&s) /* insert string s at n */
- string::find(s)
- string::rfind(s) /* search left or right for the given string */
- string::first()
- string::last() /* get characters, also there are iterators */

Strategies summaries

Find pattern/substring in a string:

- Hashmap O(1)

Find all the strings in a dictionnaire starting with a given prefix

- Trie, radix tree

Find a substring in a string

- Use the native implementation of the language

Find a long substring in a string

- Use Knuth-Morris-Pratt or Boyer-Moore O(np+ns)

Find a disctionnary of substrings in a string

- Aho-Corasick $O(n* k + l + m)$
- Suffix tree

Find a complex pattern in a string

- Use a regexp or an automate compiled from the pattern

Bit manipulation

Introduction

- Data represented as bits
- C++ allows direct manipulation
- Sometimes faster to execute and/or to write

Bitwise operation

- **bitwise NOT**, or **complement**, is a unary operation that performs logical negation on each bit
  - ~ 00110101 = 11001010
- **bitwise AND** takes **two equal-length binary representations** and performs the logical AND operation on each pair of the corresponding bits
  - 01011101 & 00110101 = 00010101
- **bitwise OR** takes two bit patterns of equal length and performs the logical inclusive OR operation on each pair of corresponding bits.
  - 01011101 | 00110101 = 01111101
- A **bitwise XOR** takes two bit patterns of equal length and performs the logical exclusive OR operation on each pair of corresponding bits.
  - 01011101 ^ 00110101 = 01101000

Bit shift operation

- **Left shift**
  - Shift to left and add zeros
  - Keep sign bit
  - Can truncate
  - 01011101 << 2 = 01110100
- **Right shift**
  - Shift to right and add zeros
  - Keep sign bit
  - Can truncate
  - 01011101 >> 2 = 00010111

Attention

- The result of shifting by a bit count greater than or equal to the word's size is undefined behavior in C and C++.
  - a << 1337 is undefined !
- Right-shifting a negative value is implementation-defined and not recommended by good coding practice;
- The result of left-shifting a signed value is undefined if the result cannot be represented in the result type.
- Be mindful of the priority
  - a & b == 1 -> a & (b == 1)

Sets

- Store a small set in the bits of an unsigned integer
- Bit i is 1 iff element i is in the set
  - You can enumerate the sets by enumerating the integers
  - More compact than set and sometimes needed to reach the memory/time target
- unsigned long long: 64 bits, unsigned long: 32 bits, unsigned: 16 bits
- Also : uint64_t, uint32_t, uint16_t, uint8_t (with #include <cstdint>)
- Be mindful of casting
  - uint64_t x = 1 << 33; // bug (undefined)
  - uint64_t x = ((uint64_t) 1) << 33; // OK

Sets

- s & t intersection
- s | t union
- s ˆ t (Elements in s only or t only)
- s & (1 << i) teste si l'élément *i* est dans l'ensemble
- s | (1 << i) add element *i*
- s & ˜(1 << i) remove element *i*
- s ˆ (1 << i) toggle element *i*
- s & (s-1) remove the smallest element le plus petit élément

Sets

- When dealing with sets that cannot fit in integers
  - Bitset<N> - fixed size
  - Vector<bool> - dynamic size
    - Overhead of vector
- Enumerate all subsets of s
  - *n* is the current subset
  - Approach :
  - | ~s to set to 1 all unused bits
  - +1 to propagate a carry over
  - & s to set to 0 all unused bits

    -> Next subset: ((n | ~s) + 1) & s

Numbers

- Two’s complement
  - The two's complement of an *N*-bit number is defined as its complement with respect to 2^N. For instance, for the **three-bit** number 010, the two's complement is 110, because 010 + 110 = 1000 = 2^3.
  - Used to represent signed integers
    - Positive from 0 to 2^{N-1} (excluded)
    - Negative from -1 to -2^{N-1} (included)

Three-bit signed integers		
Decimal value	Binary (two's-complement representation)	Two's complement $(2^3 - n)_2$
0	000	000
1	001	111
2	010	110
3	011	101
-4	100	100
-3	101	011
-2	110	010
-1	111	001

Numbers

- Power of twos
  - Test if a number x is even : ! (x&1)
  - Compute 2^i : (1<<i)
  - Test if a number x is an exact power of 2: x && !(x & x-1)

Gray code

- Problem: Enumerate the values from 0 to 2^n (excluded) while changing only one bit when going from one enumeration to the next
- Exemple
  - n = 1 – Solution : C_1= 0, 1
  - n = 2 – Solution : C_2= 00, 01, 11, 10
- Recursive Algorithm
  - Build the code C_{n-1}
  - Build the mirror code $\overline{C_{n-1}}$. This is also a code
  - Build C^1_n by prefixing C_{n-1} with 0
  - Build C^2_n by prefixing $\overline{C_{n-1}}$ with 1
  - C_n is a concatenation of C^1_n followed by C^2_n

Sixth Assignment

- **Jumping Monkey** UVA
- **Counting substhreengs** UVA
- **The big painting** UVA
- 10010 Where's Waldorf? UVA
- 850 Crypt Kicker II UVA
- 10132 File Fragmentation UVA
- 10150 Doublets UVA

Algorithmic Geometry

See pdf

Assignment

- **Regular Convex Polygon** UVA
- **Trash Removal** UVA
- **Bribing Eve** UVA
- 10310 Dog and Gopher UVA
- 10180 Rope Crisis in Ropeland! UVA
- 10167 Birthday Cake (1) UVA
- 10012 How Big Is It? UVA
- 10135 Herding Frosh UVA
- 10043 Chainsaw Massacre UVA
- 10084 Hotter Colder UVA
- 10117 Nice Milk UVA