ACM International Collegiate Programming Contest — Training Session II

C. Maria Keet

Department of Computer Science, University of Cape Town, South Africa
mkeet@cs.uct.ac.za

August 13, 2016

Outline

Today

- CS ‘problem-solving paradigm’
- Identifying a CS ‘problem-solving paradigm’ from the description
- Algorithms: Strings
- Another one, fitting one of the PSPs (hints later)
- If you don’t finish a problem, try at home and make sure you’ve implemented it, can submit to the ICPC and UVa sites anytime

Outline

1. CS PSP
2. Classify problems
3. Some problems in detail
   - Student IDs
   - Legal Pascal Real Constants
   - Similarity
   - More strings: data structures and algorithms
   - Durban Prawns

Problem-solving paradigms in computing[1]

- **CS** Complete search (brute force)
- **D&C** Divide & Conquer
- **Gr** Greedy
- **DP** Dynamic Programming

[1] content in this section based on "Competitive programming 1", by Steven and Felix Halim

Problem-solving paradigms in computing[1]

CS Complete search (brute force)

- solve problem searching the entire search space

D&C Divide & Conquer

Gr Greedy

DP Dynamic Programming

[1] content in this section based on "Competitive programming 1", by Steven and Felix Halim

Problem-solving paradigms in computing[1]

- **CS** Complete search (brute force)
  - solve problem searching the entire search space
- **D&C** Divide & Conquer
  - make problem 'simpler' by dividing into sub-problems (usually half the size)
- **Gr** Greedy
- **DP** Dynamic Programming

[1] content in this section based on "Competitive programming 1", by Steven and Felix Halim

Problem-solving paradigms in computing[1]

- **CS** Complete search (brute force)
  - solve problem searching the entire search space
- **D&C** Divide & Conquer
  - make problem 'simpler' by dividing into sub-problems (usually half the size)
- **Gr** Greedy
  - make locally optimal choice at each step
- **DP** Dynamic Programming

[1] content in this section based on "Competitive programming 1", by Steven and Felix Halim

Problem-solving paradigms in computing[1]

- CS Complete search (brute force)
  - solve problem searching the entire search space
- D&C Divide & Conquer
  - make problem ‘simpler’ by dividing into sub-problems (usually half the size)
- Gr Greedy
  - make locally optimal choice at each step
- DP Dynamic Programming
  - problem that has overlapping subproblems and optimal substructure (more on Sept 10)

[1] content in this section based on “Competitive programming 1”, by Steven and Felix Halim

Notes on Complete Search

- Bug-free code never gives WA (wrong answer)
- Useful for ‘small values’, but inefficient for larger spaces (resulting in TLE (time limit exceeded)) (recall algorithm complexity)
- Can run Complete Search on small instances of a hard problem to get some patterns from its output
- Can serve as verifier for faster non-trivial algorithms

Some tips on Complete Search

- Filtering the right answer from a set of values vs. generating only the right values (latter more efficient)
- Think about the data space: remove infeasible space upfront
- Utilise symmetries
- Pre-computation: make better data structure, such that time making it outweighs the ‘loss’ in time searching
- Solve it ‘backwards’ from the data/results space fitting to the problem

Notes on D&C

- Divide the original problem into sub-problems — usually by half or nearly half
- Find (sub-)solutions for each of these sub-problems – which are now easier
- If needed, combine the sub-solutions to produce a complete solution for the main problem

Notes on D&C

- Divide the original problem into sub-problems — usually by half or nearly half
- Find (sub-)solutions for each of these sub-problems – which are now easier
- If needed, combine the sub-solutions to produce a complete solution for the main problem
- Binary search seems easy, but there are some creative options there

Notes on Greedy

- Problem must exhibit two things in order for a greedy algorithm to work for it:
  1. It has optimal sub-structures. (Optimal solution to the problem contains optimal solutions to the sub-problems.)
  2. It has a greedy property. (If we make a choice that seems best at the moment and solve the remaining subproblems later, we still reach optimal solution. We never have to reconsider our previous choices)

Notes on Greedy

- Problem must exhibit two things in order for a greedy algorithm to work for it:
  1. It has optimal sub-structures. (Optimal solution to the problem contains optimal solutions to the sub-problems.)
  2. It has a greedy property. (If we make a choice that seems best at the moment and solve the remaining subproblems later, we still reach optimal solution. We never have to reconsider our previous choices)
- Risky: if you get WA and code is correct, the problem may not be greedy after all
- Use when input size is too large for CS or DP

Outline

Problem sets

- See printout (there are three sets printed, of varying difficulty)
- Most problems are taken from, or based on, the UVa problem set (includes ICPC problem set)
- Each problem falls in one PSP or ad hoc/simulation
- Read them, classify them
- Discuss afterward, and solve them

Solution

- Work reduction – Greedy
- Trainsorting – Dynamic Programming
- Wine trading in Gergovia – Greedy
- The jackpot – Dynamic Programming
- Durban prawns – Complete Search
- Mobile Phone Coverage – Complete Search + Geometry
- Blowing fuses – ad hoc (simulation)
- Dog lineup – Divide & Conquer (binary search)

Outline

Student IDs

- 2013 regionals problem (see printout), many teams solved it
- type: algorithmically, some coding, relatively easy (in the grand scheme of things)
- As exercise: use last week's methodological steps
- Solve at least the 'what' and 'how'-parts in 30 minutes

Student IDs

- 2013 regionals problem (see printout), many teams solved it
- type: algorithmically, some coding, relatively easy (in the grand scheme of things)
- As exercise: use last week's methodological steps
- Solve at least the 'what' and 'how'-parts in 30 minutes

⇒ automata are helpful. design important to make sure you don't miss anything.

Legal Pascal Real Constants

- UVa 325: Identifying Legal Pascal Real Constants
- type: string processing, for pattern matching

Legal Pascal Real Constants

- UVa 325: Identifying Legal Pascal Real Constants
- type: string processing, for pattern matching

$\Rightarrow$ need to know about regular expressions

Regular expressions

- Given an alphabet Σ, regular expressions are strings over the alphabet $\Sigma \cup \{+, *, (,), \cdot, \varepsilon, \emptyset\}$ defined inductively as follows:
  - Basis: ε, $\emptyset$, and each $\mathtt{a} \in \Sigma$ is a RE
  - Inductive step: if $\mathtt{E}$ and $\mathtt{F}$ are REs, then so are:
    - $\mathtt{E+F}$ (i.e., the union: strings are either in E or on F or in both)
    - $\mathtt{E} \cdot \mathtt{F}$ (concatenation, symbol is often omitted in many notations and implied)
    - $\mathtt{E}^*$ (closure, interpret as 'zero or more times')
    - $\mathtt{(E)}$ (the usual parentheses)
- e.g., $\mathtt{a} \cdot (\mathtt{a} + \mathtt{b})^*\mathtt{b}^*\mathtt{a}$ is a regular expression.

Regular expressions

- as is, e.g., $\mathtt{a}(\mathtt{a}+\mathtt{b})^*\mathtt{b}^*\mathtt{a}$. do these 'match' (/are they in the language / can the RE find them)?
  - aa
  - aaba
  - abab
  - aabbbbaba
- `www.regular-expressions.info` on how it's done in many programming languages

Regular expressions

- as is, e.g., $\mathtt{a}(\mathtt{a}+\mathtt{b})^*\mathtt{b}^*\mathtt{a}$. do these 'match' (/are they in the language / can the RE find them)?
  - aa yes, shortest string possible
  - aaba
  - abab
  - aabbbbaba
- `www.regular-expressions.info` on how it's done in many programming languages

Regular expressions

- as is, e.g., $\mathtt{a}(\mathtt{a}+\mathtt{b})^*\mathtt{b}^*\mathtt{a}$. do these ‘match’ (/are they in the language / can the RE find them)?
    - aa yes, shortest string possible
    - aaba yes (check this)
    - abab
    - aabbbbaba
- `www.regular-expressions.info` on how it’s done in many programming languages

Regular expressions

- as is, e.g., $\mathtt{a}(\mathtt{a}+\mathtt{b})^*\mathtt{b}^*\mathtt{a}$. do these 'match' (/are they in the language / can the RE find them)?
  - aa yes, shortest string possible
  - aaba yes (check this)
  - abab no
  - aabbbbaba
- `www.regular-expressions.info` on how it's done in many programming languages

Regular expressions

- as is, e.g., $\mathtt{a}(\mathtt{a}+\mathtt{b})^*\mathtt{b}^*\mathtt{a}$. do these ‘match’ (/are they in the language / can the RE find them)?
    - aa yes, shortest string possible
    - aaba yes (check this)
    - abab no
    - aabbbbaba yes (check this)
- `www.regular-expressions.info` on how it’s done in many programming languages

Legal Pascal Real Constants

- UVa 325: Identifying Legal Pascal Real Constants

Legal Pascal Real Constants

- UVa 325: Identifying Legal Pascal Real Constants

⇒ with line stored in String `s`, then the 1-line solution in java is:

```
s.matches("[-+]?\\d+(\\.\\d+([eE][-+]?\\d+)?|[eE][-+]?\\d+)")
```

Similarity problem

- also a 2013 regionals problem (see printout), hardly anyone solved it
- type: algorithmically, difficulty depends on prior knowledge
- First exercise: understand the problem
  - map the problem space (what is asked for, examples, input space, output, ...)
  - what is needed?

Solution (direction)

- Distance `word_A` to `word_B`
- Changes: insert (cost 2), delete (cost 2), replace (just case cost 1, else 2)
- Compare 'words' (strings), letter by letter (but just that would have misalignment issues)

Solution (direction)

- Distance `word_A` to `word_B`
- Changes: insert (cost 2), delete (cost 2), replace (just case cost 1, else 2)
- Compare ‘words’ (strings), letter by letter (but just that would have misalignment issues)
- My association: 1) DNA and RNA alignments, 2) natural languages and computational linguistics
- This was solved a while ago, so there must be an existing algorithm

Solution (direction)

- Indeed: Levenshtein distance!
  - How exactly does it do that?
  - Modify the algorithm for the costs that deviate from the standard algorithm
  - Implement
  - Test

Solution (direction)

- Indeed: Levenshtein distance!
  - How exactly does it do that?
  - Modify the algorithm for the costs that deviate from the standard algorithm
  - Implement
  - Test
- Solved by recognising the problem to be basically the same one that was already solved

$\Rightarrow$ know and understand your algorithms

Levenshtein distance[2]

- Measure of the similarity between two strings, being the source string (s) and the target string (t)
- Distance is the number of deletions, insertions, or substitutions required to transform s into t
- Named after the Russian scientist Vladimir Levenshtein, who devised the algorithm in 1965, used in, a.o.:
  - Spell checking
  - DNA analysis
  - Plagiarism detection

[2] based on info from http://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Fall2006/Assignments/editdistance/Levenshtein%20Distance.htm

Main steps (1/2)

1. Step 1
   - Set n to be the length of s
   - Set m to be the length of t
   - If $n = 0$, return m and exit
   - If $m = 0$, return n and exit
   - Construct a matrix containing $0..m$ rows and $0..n$ columns

[3] easier: shortest string as column, fill columns fltr

Main steps (1/2)

1. Step 1
   - Set n to be the length of s
   - Set m to be the length of t
   - If $n = 0$, return m and exit
   - If $m = 0$, return n and exit
   - Construct a matrix containing $0..m$ rows and $0..n$ columns
2. Step 2
   - Initialize the first row to $0..n$
   - Initialize the first column to $0..m$

[3] easier: shortest string as column, fill columns fltr

Main steps (1/2)

1. Step 1
   - Set n to be the length of s
   - Set m to be the length of t
   - If $n = 0$, return m and exit
   - If $m = 0$, return n and exit
   - Construct a matrix containing $0..m$ rows and $0..n$ columns
2. Step 2
   - Initialize the first row to $0..n$
   - Initialize the first column to $0..m$
3. Step 3
   - Examine each character of s (i from 1 to n)

[3] easier: shortest string as column, fill columns fltr

Main steps (1/2)

1. Step 1
   - Set n to be the length of s
   - Set m to be the length of t
   - If $n = 0$, return m and exit
   - If $m = 0$, return n and exit
   - Construct a matrix containing $0..m$ rows and $0..n$ columns
2. Step 2
   - Initialize the first row to $0..n$
   - Initialize the first column to $0..m$
3. Step 3
   - Examine each character of s (i from 1 to n)
4. Step 4[3]
   - Examine each character of t (j from 1 to m)

[3] easier: shortest string as column, fill columns fltr

Main steps (2/2)

5. Step 5
   - If $s[i]$ equals $t[j]$, the cost is 0
   - If $s[i]$ doesn't equal $t[j]$, the cost is 1

Main steps (2/2)

5. Step 5
   - If $s[i]$ equals $t[j]$, the cost is 0
   - If $s[i]$ doesn't equal $t[j]$, the cost is 1
6. Step 6
   - Set cell $d[i,j]$ of the matrix equal to the minimum of:
     - The cell immediately above plus 1: $d[i-1,j]+1$
     - The cell immediately to the left plus 1: $d[i,j-1]+1$
     - The cell diagonally above and to the left plus the cost: $d[i-1,j-1]+\text{cost}$

Main steps (2/2)

5. Step 5
   - If $s[i]$ equals $t[j]$, the cost is 0
   - If $s[i]$ doesn't equal $t[j]$, the cost is 1
6. Step 6
   - Set cell $d[i, j]$ of the matrix equal to the minimum of:
     - The cell immediately above plus 1: $d[i-1, j]+1$
     - The cell immediately to the left plus 1: $d[i, j-1]+1$
     - The cell diagonally above and to the left plus the cost: $d[i-1, j-1] + \text{cost}$
7. Step 7
   - After the iteration steps (3, 4, 5, 6) are complete, the distance is found in cell $d[n, m]$

Example—Step 1 and 2

		t	a	f	e	l
	0	1	2	3	4	5
i	1					
t	2					
a	3					
f	4					
u	5					
l	6					
a	7					

Example—Steps 3-6 when $i = 1$

		t	a	f	e	l
	0	1	2	3	4	5
i	1	1				
t	2	1				
a	3	2				
f	4	3				
u	5	4				
l	6	5				
a	7	6				

Example—Steps 3-6 when $i = 2$

		t	a	f	e	l
	0	1	2	3	4	5
i	1	1	2			
t	2	1	2			
a	3	2	1			
f	4	3	2			
u	5	4	3			
l	6	5	4			
a	7	6	5			

Example—Steps 3-6 when $i = 3$

		t	a	f	e	l
	0	1	2	3	4	5
i	1	1	2	3		
t	2	1	2	3		
a	3	2	1	2		
f	4	3	2	1		
u	5	4	3	2		
l	6	5	4	3		
a	7	6	5	4		

Example—Steps 3-6 when $i = 4$

		t	a	f	e	l
	0	1	2	3	4	5
i	1	1	2	3	4	
t	2	1	2	3	4	
a	3	2	1	2	3	
f	4	3	2	1	2	
u	5	4	3	2	2	
l	6	5	4	3	3	
a	7	6	5	4	4	

Example—Steps 3-6 when $i = 5$

		t	a	f	e	l
	0	1	2	3	4	5
i	1	1	2	3	4	5
t	2	1	2	3	4	5
a	3	2	1	2	3	4
f	4	3	2	1	2	3
u	5	4	3	2	2	3
l	6	5	4	3	3	2
a	7	6	5	4	4	3

Example—Final

		t	a	f	e	l
	0	1	2	3	4	5
i	1	1	2	3	4	5
t	2	1	2	3	4	5
a	3	2	1	2	3	4
f	4	3	2	1	2	3
u	5	4	3	2	2	3
l	6	5	4	3	3	2
a	7	6	5	4	4	3

Finalise the similarity problem

- implement the algorithm
- modify algorithm for different costs, according to specs
- check any loose ends; e.g.:
  - what if either m or n is empty?
  - what if $n > m$ (i.e., first string is longer than the second), or vv?
  - does it work with numbers? (e.g.: `gr8` and `great`)

Other string problems

- Text messages: with a fancy data structure (UVa 6133)
- Top 10 (also in the combinedSet.pdf): UVa 1254 and Asia/Jakarta 2009 (also with a ‘fancy; data structure)
- Palindromes: algorithmic. sources differ on emphasis of solution (UVa 11151); some typical tasks:
  - Determine whether a string is a palindrome
  - Longest palindromic substring

Other string problems

- Text messages: with a fancy data structure (UVa 6133)
- Top 10 (also in the combinedSet.pdf): UVa 1254 and Asia/Jakarta 2009 (also with a ‘fancy; data structure)
- Palindromes: algorithmic. sources differ on emphasis of solution (UVa 11151); some typical tasks:
  - Determine whether a string is a palindrome (recursive algorithm)
  - Longest palindromic substring (Manacher's algorithm; nice explanation at `http://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-sub`

Durban Prawns: toward a solution–first try

- 1024×1024 array
- $n \leq 20000$ cockroaches
- Determine which cell has to be bombed so that a square box from (x-d, y-d) to (x+d, y+d) is maximised, with d the power of the gas bomb (and $d \leq 50$)

Durban Prawns: toward a solution–first try

- 1024×1024 array
- $n \leq 20000$ cockroaches
- Determine which cell has to be bombed so that a square box from (x-d, y-d) to (x+d, y+d) is maximised, with d the power of the gas bomb (and $d \leq 50$)
- First option: bomb each of the 1024^2 cells, select most effective location using an $O(d^2)$ scan

Durban Prawns: toward a solution–first try

- 1024×1024 array
- $n \leq 20000$ cockroaches
- Determine which cell has to be bombed so that a square box from (x-d, y-d) to (x+d, y+d) is maximised, with d the power of the gas bomb (and $d \leq 50$)
- First option: bomb each of the 1024^2 cells, select most effective location using an $O(d^2)$ scan
- Probably not fast enough (TLE danger): $1024^2 \times 50^2 = 2621M$ operation

Toward a solution–second try

- Look again at the sample input and output: while the grid indeed could be fully populated with cockroaches (*worst case*), this clearly need not be the case

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4					10				
5									
6							20		
7									
8									

Toward a solution–second try

$\Rightarrow$ Solve it going 'backward' (recall slide 11)

- Instead of computing everything and then looking at the number of killed cockroaches, we make an array `int killed[1024][1024]`, then
  - For each rat population at coordinate (x, y), add it to `killed[i][j]`, where $|i - x| \leq d$ and $|j - y| \leq d$ (this costs us $O(n \times d^2)$ operations)
  - Find optimal bombing position: find highest entry in `killed` array (can be done in 1024^2 operations)

Grid with the sample input

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4					10				
5									
6							20		
7									
8									

killed array with the sample input + first population

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3				10	10	10			
4				10	10	10			
5				10	10	10			
6							20		
7									
8									

`killed` array with the sample input + second population

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3				10	10	10			
4				10	10	10			
5				10	10	30	20	20	
6						20	20	20	
7						20	20	20	
8									

`killed` array with the sample inputs—highest value

	0	1	2	3	4	5	6	7	8
0									
1									
2									
3				10	10	10			
4				10	10	10			
5				10	10	30	20	20	
6						20	20	20	
7						20	20	20	
8									

Scheduled training dates

- Aug 6: 10:00-16:00
- Aug 13: 10:00-16:00
- **Aug 27: 10:00-16:00**
- Sept 10: 10:00-16:00 → Ashraf Moolla on DP
- Sept 17: 10:00-16:00
- Sept 24: 10:00-16:00 or Oct 1: 10:00-16:00
- Date of the regionals: TBD ("some Saturday between mid Sept and end Oct")