GCPC 2012

GCPC 2012 Jury

gcpc@gcpc.nwerc.eu

30.06.2012

Battleship

- Solve by Simulation
- Read problem statement carefully
- Ending the game and draw may be tricky cases

BrainfuckVM

- Problem: decide whether a program terminates

BrainfuckVM

- Problem: decide whether a program terminates
- Solution: simulate program while keeping track of states
- simulate concurrently at normal and at double speed (Floyd/Brent's cycle trick)
- if both simulations have equal state: loop!
- needs clever state comparison to be fast enough

BrainfuckVM

- $\Rightarrow$ simpler solution: just simulate 50 000 000 steps
- if not terminated $\rightarrow$ loop
- simulate another 50 000 000 steps to get loop instructions
- no state comparison needed

Candy Distribution

- K kids, C candies in one bag
- to compute: #bags b so that $b \cdot C = (K \cdot X) + 1$ (X is a positive integer)

Candy Distribution

- K kids, C candies in one bag
- to compute: #bags b so that $b \cdot C = (K \cdot X) + 1$ (X is a positive integer)
- $\Rightarrow b \cdot C \equiv 1 \bmod K$

Candy Distribution

- K kids, C candies in one bag
- to compute: $\#$bags b so that $b \cdot C = (K \cdot X) + 1$ (X is a positive integer)
- $\Rightarrow b \cdot C \equiv 1 \bmod K$
- compute modular inverse with extended euclid
- be careful with special cases like $K = 1$, $C = 1$, or both

Candy Distribution

- K kids, C candies in one bag
- to compute: $\#$bags b so that $b \cdot C = (K \cdot X) + 1$ (X is a positive integer)
- $\Rightarrow b \cdot C \equiv 1 \bmod K$
- compute modular inverse with extended euclid
- be careful with special cases like $K = 1$, $C = 1$, or both
- equation solvable if K and C coprime

Outsourcing

- Underlying problem: Check two DFAs for equivalence

Outsourcing

- Underlying problem: Check two DFAs for equivalence
- No bruteforcing possible (how?)

Outsourcing

- Underlying problem: Check two DFAs for equivalence
- No bruteforcing possible (how?)
- Possible solution:

Outsourcing

- Underlying problem: Check two DFAs for equivalence
- No bruteforcing possible (how?)
- Possible solution:
  - Minimize both DFAs (see Hopcroft/Ullman)
  - Check for equivalence by for example DFS:
  - Simultaneous DFS on the two minimized automatas numbering the states in preorder
  - Check if the state reached by an input character has same DFS number or is unvisited
  - Check if the final states have the same DFS number
  - Runs in $O(|states|^2 \cdot |\Sigma|)$

Outsourcing

- Smart algorithm by Hopcroft and Karp:

Outsourcing

- Smart algorithm by Hopcroft and Karp:
  - Join the automata
  - Union states that are reached by the same inputs by DFS
  - After all, check if there is a union of exactly the two final states
  - Really fast (nearly $O(|states| + |transitions|)$) with a good union-find implementation

Pizza Hawaii

- Given the ingredients of Pizzas in two languages
- For each word determine which words could have the same meaning in the other language

Pizza Hawaii

- Given the ingredients of Pizzas in two languages
- For each word determine which words could have the same meaning in the other language
- Solution: Match words which appear as ingredients on the same set of Pizzas

Pizza Hawaii

- Given the ingredients of Pizzas in two languages
- For each word determine which words could have the same meaning in the other language
- Solution: Match words which appear as ingredients on the same set of Pizzas
- Use bitmasks to specify for each ingredient the subset of Pizzas on which this ingredient occurs.

Pizza Hawaii

- Given the ingredients of Pizzas in two languages
- For each word determine which words could have the same meaning in the other language
- Solution: Match words which appear as ingredients on the same set of Pizzas
- Use bitmasks to specify for each ingredient the subset of Pizzas on which this ingredient occurs.
- Brute force over all pairs of words and check if their corresponding bitmasks are equal

Roller coaster fun

Normal Knapsack problems can be solved by either of the two possible recursion equations:

Roller coaster fun

Normal Knapsack problems can be solved by either of the two possible recursion equations:

- *Unbounded Knapsack*:

$$dp[id][size] = \max \begin{cases} dp[id-1][size] \\ dp[id][size - weight[id]] + profit[id] \end{cases}$$

- 0/1-*Knapsack*:

$$dp[id][size] = \max \begin{cases} dp[id-1][size] \\ dp[id-1][size - weight[id]] + profit[id] \end{cases}$$

Roller coaster fun

For this problem we need both recursion types. An item (roller coaster) is given by the three values a_i, b_i and t_i

Roller coaster fun

For this problem we need both recursion types. An item (roller coaster) is given by the three values a_i, b_i and t_i

- Case $b_i = 0$ (*unbounded knapsack*):
  $dp[i][T] = \max\{dp[i-1][T], dp[i][T-t_i] + a_i\}$

Roller coaster fun

For this problem we need both recursion types. An item (roller coaster) is given by the three values a_i, b_i and t_i

- Case $b_i = 0$ (*unbounded knapsack*):
  $dp[i][T] = \max\{dp[i-1][T], dp[i][T-t_i] + a_i\}$
- Case $b_i \neq 0$ (0/1-*knapsack*?)

Roller coaster fun

For this problem we need both recursion types. An item (roller coaster) is given by the three values a_i, b_i and t_i

- Case $b_i = 0$ (*unbounded knapsack*):
  $dp[i][T] = \max\{dp[i-1][T], dp[i][T-t_i] + a_i\}$
- Case $b_i \neq 0$ ($0/1$-*knapsack*?)
  No!

Roller coaster fun

For this problem we need both recursion types. An item (roller coaster) is given by the three values a_i, b_i and t_i

- Case $b_i = 0$ (*unbounded knapsack*):
  $dp[i][T] = \max\{dp[i-1][T], dp[i][T-t_i] + a_i\}$
- Case $b_i \neq 0$ (0/1-*knapsack*?)
  No!
  $\Rightarrow$ split into J_i items, where the k-th item has a profit (fun) of $a_i - (k-1)^2 \cdot b_i$ and weight (time) t_i.
  J_i is the largest index where the profit is positive.

Roller coaster fun

For this problem we need both recursion types. An item (roller coaster) is given by the three values a_i, b_i and t_i

- Case $b_i = 0$ (*unbounded knapsack*):
  $dp[i][T] = \max\{dp[i-1][T], dp[i][T-t_i] + a_i\}$
- Case $b_i \neq 0$ ($0/1$-*knapsack*?)
  No!
  $\Rightarrow$ split into J_i items, where the k-th item has a profit (fun) of $a_i - (k-1)^2 \cdot b_i$ and weight (time) t_i.
  J_i is the largest index where the profit is positive.
  Use $0/1$-*knapsack* on those items!

Roller coaster fun

Time complexity:

- Build table: $O(N \cdot J_{max} \cdot T_{max})$
- Query table entries: $O(Q)$

Only one testcase, $J_{max} \leq 32$
$\Rightarrow$ time complexity is sufficient

Roller coaster fun

Memory complexity:
Size of table: $N \cdot J_{max} \cdot T_{max} \approx 305\text{MB}$

Roller coaster fun

Memory complexity:
Size of table: $N \cdot J_{max} \cdot T_{max} \approx 305\text{MB}$
Idea:

- Calculate table line by line
- Only previous line necessary for calculation
- Only last line is needed for the queries

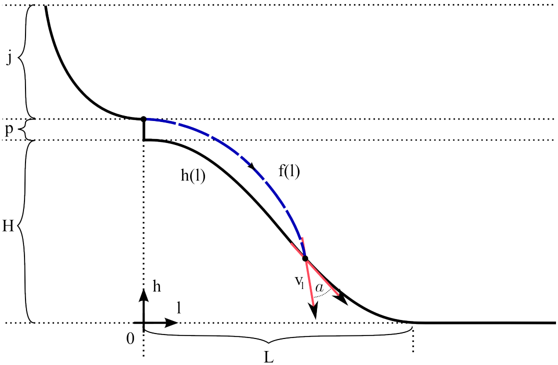
Roller coaster fun

Memory complexity:
Size of table: $N \cdot J_{max} \cdot T_{max} \approx 305\text{MB}$
Idea:

- Calculate table line by line
- Only previous line necessary for calculation
- Only last line is needed for the queries

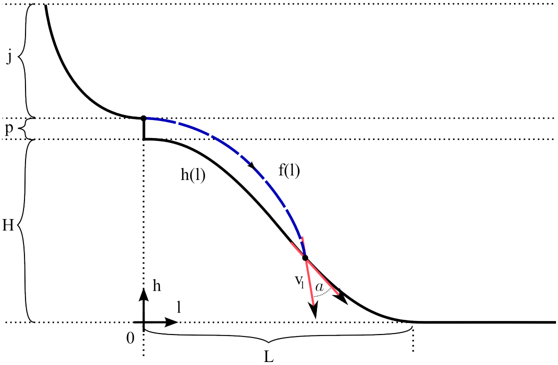
$\rightsquigarrow$ Memory: $O(2 \cdot T_{max}) < 1\text{MB}$

Ski Jumping



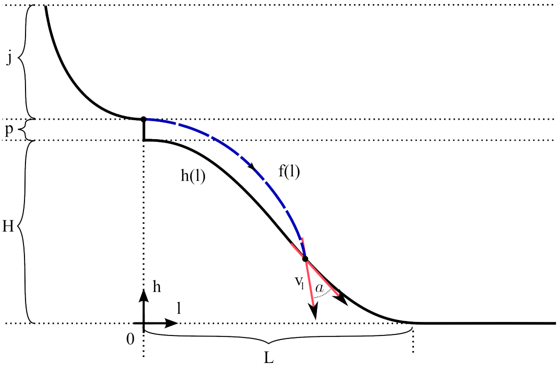
- Find l
  1. by solving for $l \rightarrow$ much math, paper and pencil approach, $\mathcal{O}(1)$
  2. by binary search on $l \rightarrow$ easy too implement, $\mathcal{O}(\log N)$

Ski Jumping



- Get landing speed $|v|$
  - $v_x =$ speed gained in approach, not changed during flight
  - $v_y =$ speed gained during flight (drop since approach)

Ski Jumping



- Get landing angle
  1. First derivatives of f and h yield slopes
  2. Obtain slopes at landing point
  3. Write slopes as vector and apply given equation
  4. Convert rad to degree

Suffix Array RE-construction

- Problem: Reconstruct a full string from a partial set of suffixes

- Problem: Reconstruct a full string from a partial set of suffixes
- Straight forward task, special character '*' occurs at most once per suffix
- Fill in characters into output string, print `IMPOSSIBLE` whenever a conflict occurs or characters are missing

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	S	t	r	i	n	g	W	i	t	h	C	o	n	f	l	i	c	t	s

1 AStringW*Conflicts → 1 AStringW, 12 Conflicts

8 WithC*Licts → 8 WithC, 16 Licts

Suffix Array RE-construction

- Problem: Reconstruct a full string from a partial set of suffixes
- Straight forward task, special character '*' occurs at most once per suffix
- Fill in characters into output string, print `IMPOSSIBLE` whenever a conflict occurs or characters are missing

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	S	t	r	i	n	g	W	i	t	h	C	o	n	f	l	i	c	t	s

1 AStringW*Conflicts → 1 AStringW
→ 12 Conflicts

8 WithC*Licts → 8 WithC
→ 16 Licts

Suffix Array RE-construction

- Problem: Reconstruct a full string from a partial set of suffixes
- Straight forward task, special character '*' occurs at most once per suffix
- Fill in characters into output string, print `IMPOSSIBLE` whenever a conflict occurs or characters are missing

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	S	t	r	i	n	g	W	i	t	h	C	o	n	f	l	i	c	t	s

1 AStringW*Conflicts → 1 AStringW
→ 12 Conflicts

8 WithC*Licts → 8 WithC
→ 16 Licts

Suffix Array RE-construction

- Problem: Reconstruct a full string from a partial set of suffixes
- Straight forward task, special character '*' occurs at most once per suffix
- Fill in characters into output string, print `IMPOSSIBLE` whenever a conflict occurs or characters are missing

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	S	t	r	i	n	g	W	i	t	h	C	o	n	f	l	i	c	t	s

1 AStringW*Conflicts → 1 AStringW
→ 12 Conflicts

8 WithC*Licts → 8 WithC
→ 16 Licts

Touchscreen Keyboard

- intended to be the easiest problem

Touchscreen Keyboard

- intended to be the easiest problem
- calculate distance between letters:

```
string keys[3] = {"qwertyuiop","asdfghjkl","zxcvbnm"};
for(i,0..3) for(j,0..keys[i].size()) {
    x[keys[i][j]] = i;
    y[keys[i][j]] = j;
}
```

Touchscreen Keyboard

- intended to be the easiest problem
- calculate distance between letters:

```
string keys[3] = {"qwertyuiop","asdfghjkl","zxcvbnm"};
for(i,0..3) for(j,0..keys[i].size()) {
    x[keys[i][j]] = i;
    y[keys[i][j]] = j;
}
```

- compute distance sums:

```
for(i,0..n) sum += abs(x[ref[i]]-x[cur[i]])
                 + abs(y[ref[i]]-y[cur[i]]);
```

Touchscreen Keyboard

- intended to be the easiest problem
- calculate distance between letters:

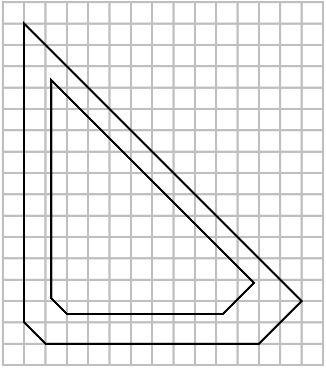
```
string keys[3] = {"qwertyuiop","asdfghjkl","zxcvbnm"};
for(i,0..3) for(j,0..keys[i].size()) {
    x[keys[i][j]] = i;
    y[keys[i][j]] = j;
}
```

- compute distance sums:

```
for(i,0..n) sum += abs(x[ref[i]]-x[cur[i]])
                 + abs(y[ref[i]]-y[cur[i]]);
```

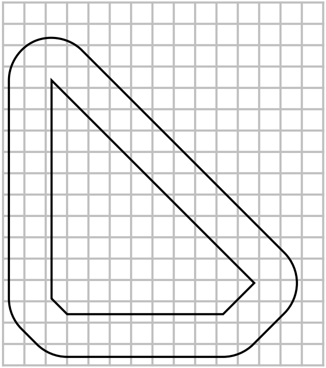
- sort and print

scale with f



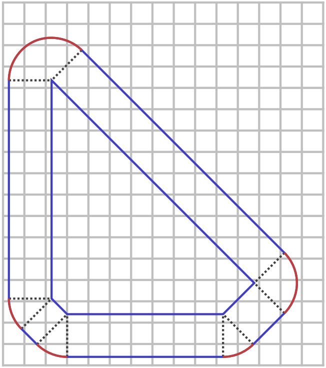
Track Smoothing

border with distance r



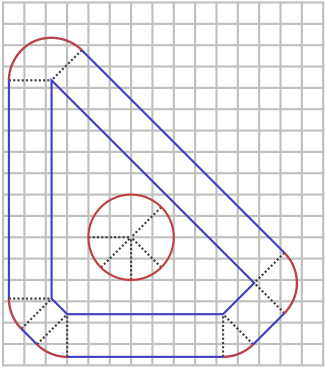
Track Smoothing

$f \cdot$ track_length



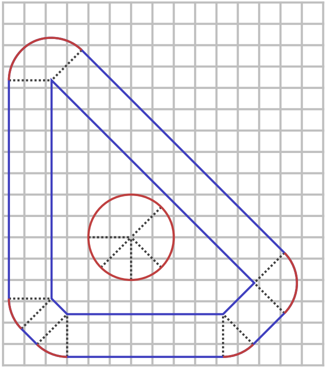
Track Smoothing

$f \cdot \text{track\_length} + 2r\pi$



Track Smoothing

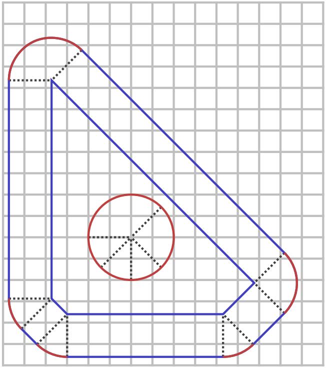
$$f \cdot \text{track\_length} + 2r\pi = \text{track\_length}$$



- $f = \frac{\text{track\_length} - 2r\pi}{\text{track\_length}}$

Track Smoothing

$$f \cdot \text{track\_length} + 2r\pi = \text{track\_length}$$



- $f = \frac{\text{track\_length} - 2r\pi}{\text{track\_length}}$
- negative $\Rightarrow$ "Not possible"

Treasure Diving

- Problem: decide how many Treasures a diver can rescue from a cave network using a limited air budget

Treasure Diving

- Problem: decide how many Treasures a diver can rescue from a cave network using a limited air budget
- Classical TSP instance, with a minor twist
- The diver does not have to collect all treasures, only maximal number possible
- Two step approach
  - calculate distance table
    (at most 8 Treasures + exit → 9x9 Table)
  - perform backtracking on table, recursing only if air sufficient for the return

Award Ceremony