Exercise review
Dynamic programming (intro)
CS 2860: Algorithms and Complexity I

Magnus Wahlström, McCrea 113

Magnus.Wahlstrom@rhul.ac.uk

December 3, 2014

Lab 8 review

Question 2: Find longest palindrome

- For a better bound, we need a trick
- Observe: In the "center" of every palindrome is a shorter palindrome (`"bb"` in "abba", or `"e"` in `"racecar"`)
- Conversely: Starting from `"e"` in `"racecar"`, we can grow to `"cec"`, to `"aceca"`, and to `"racecar"`
- By $O(n)$ calls to a palindrome growing function, we can solve the problem

Question 2: Faster solution

- Helper function `expand(text,i,j)`: Find and return the longest palindrome centered in `text[i...j]`
  1. While `text.charAt(i-1) == text.charAt(j+1)`: Let `i=i-1` and `j=j+1`
  2. Afterwards, return substring `text[i...j]` (call `text.subString(i,j+1)`)
- Main function `longestPalindrome(text)`:
  1. For `i=0` to `text.length()-1`:
     1.1 Call `expand(text,i,i)` (odd-length palindromes)
     1.2 If `text.charAt(i) == text.charAt(i+1)`: Also call `expand(text,i,i+1)` (even-length palindromes)
     1.3 Remember the longest palindrome found so far
- Time: $O(n)$ per expand, $O(n)$ expand-calls $\Rightarrow O(n^2)$ in total

Dynamic Programming

Illustration: Fibonacci

- Fibonacci numbers $F(n)$:
  - $F(0) = 0$
  - $F(1) = 1$
  - $F(n) = F(n-1) + F(n-2)$ if $n \geq 2$
- First values: 0, 1, 1, 2, 3, 5, 8, 13...

Recursive computation:

```
long fib(int n) {
  if (n<=0) return 0;
  if (n==1) return 1;
  return f(n-1)+f(n-2);
}
```

Recursive Fibonacci

- Note wasted effort
  - Computing $f(4)$ requires computing $f(2)$ twice
  - Each time calls $f(0)$ and $f(1)$ again
  - Computing $f(5)$ requires computing $f(2)$ three times
  - ...(Exponential growth!)
- Plan 1: Remember value of $f(2)$ after first time computed (in some table)
- Avoid re-computations by lookup in table

Generalising

- Plan 1 (memoization) works – but rediscovers the same table structure each time
- Moderately tricky programming (for "memory")
- Plan 2: Construct table bottom-up
- We already know which computations to perform
- This is dynamic programming

Dynamic programming

General scheme:

1. A recursive procedure can be slow because of repeated subproblems
2. By caching computed answers in a table, we can speed up computation (memoization)
3. By further "understanding" the structure of the table, can compute answer directly (without recursion) (dynamic programming)
4. If there are only few different subproblem, improvement can be drastic! ($F(n)$: From $O(1.62^n)$ to $O(n)$.)
5. More advanced: Design recursive solutions to work with dynamic programming

Fibonacci (final)

For completeness: Simpler and faster fibonacci

```
long fib(int n) {
  long prev1=0, prev2=1, current;
  for (int i=2; i<=n; i++) {
    current = prev1 + prev2;
    prev2 = prev1;
    prev1 = current;
  }
  return current;
```