Lecture 16: Dynamic Programming

Agenda:

- Decision tree sorting lower bound: review
- Midterm coverage
- Dynamic programming
  - concepts
  - characteristics

Reading:

- Textbook pages 323 – 324

Sorting lower bound:

- Comparison-based sort:

  keys can be (2-way) compared only !

- This lower bound argument considers only the comparison-based sorting algorithms. For example,
  - Insertionsort, Mergesort, Heapsort, Quicksort
  - *Selectionsort*, *Bubblesort*

- Binary tree facts:
  - Suppose there are t leaves and k levels. Then,
  - $t \leq 2^{k-1}$
  - So, $\lg t \leq (k-1)$
  - Equivalently, $k \geq 1 + \lg t$
    — binary tree with t leaves has *at least* $(1 + \lg t)$ levels

- Comparison-based sorting algorithm facts:
  - Look at its *Decision Tree*. We have,
  - It's a binary tree.
  - It should contain every possible permutation of the positions $\{1, 2, \ldots, n\}$.
  - So, it contains at least $n!$ leaves ...
  - Equivalently, it has at least $1 + \lg(n!)$ levels.
  - A longest root-to-leaf path of length at least $\lg(n!)$.
  - The worst case number of KC is at least $\lg(n!)$.
  - $\lg(n!) \in \Theta(n \log n)$

Sorting lower bound (cont'd):

- Key ideas in deriving the lower bound:
  - Decision tree
  - It's binary
  - Length of longest root-to-leaf path $\longleftrightarrow$ WC KC
  - The number of possible permutations $\longleftrightarrow$ number of leaves
- It doesn't hold for non-comparison-based sorting algorithm ...

  Check Chapter 8 for extra reading

Announcements:

- Midterm (Mar 5) coverage up to this lower bound analysis
- Some important subjects so far:
  - Loop invariant (& math induction)
    - design
    - proof
  - Sorting algorithms
    - key ideas
    - execution
    - analysis (running time, space, computational models)
    - decision tree lower bound analysis
  - Asymptotic notations
    - proof by definition
  - Recurrence
    - deriving
    - closed form guessing, iterated substitution
    - proof by induction
    - recursion tree
    - Master Theorems

Dynamic programming introduction:

- An algorithm design technique

- Key idea:
  - *Avoiding re-computation*
  - of repeated subproblems by storing subproblem answers in tables/arrays

- 1st example problem — Fibonacci numbers

$$f(n) = \begin{cases} n, & \text{when } n = 0, 1 \\ f(n-1) + f(n-2), & \text{when } n \geq 2 \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9
$f(n)$	0	1	1	2	3	5	8	13	21	34

Question: how do we compute $f(n)$?

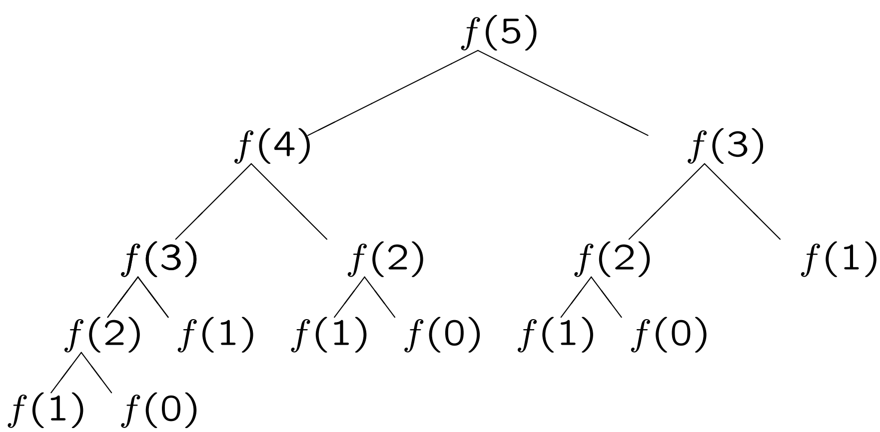
1^{st} Fibonacci implementation — recursion

- Pseudocode:

```
procedure f(n)

if n < 2 then
   return n
else
   return f(n − 1) + f(n − 2)
```

- Recursion tree:



- Notice that there are a lot of repeated function calls

- Running time recurrence

$$T(n) = \begin{cases} c_1, & \text{when } n = 0, 1 \\ c_2 + T(n-1) + T(n-2), & \text{when } n \geq 2 \end{cases}$$

- Conclusion: $T(n) > f(n) \longrightarrow T(n) \in \Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$

2^{nd} Fibonacci implementation — memoization

- Problem with the 1^{st} implementation — repeated function calls
- Improvement idea:

  Keep recursion, avoid re-computation ...

  - store $f(\cdot)$ values in array $F[\cdot]$
  - if $F[j]$ not yet initialized, compute it
  - if $F[j]$ is initialized, access it
  - $F[j]$ computed only once
- **memoization** — recursion with Dynamic Programming

```
procedure dpFib(n)

for j ← 1 to n do
    F[j] ←???          **un-initialized
F[0] ← 0
F[1] ← 1
dpf(n)

procedure dpf(n)

if F[n] =??? then
    F[n] ← dpf(n − 1) + dpf(n − 2)
return F[n]
```

- Since each $F[k]$ known after the first `dpf`(k) call,

  `dpf`(k) called $\leq$ twice
- So, running time $T(n) \in \Theta(n)$

3rd Fibonacci implementation — dynamic programming

- Pseudocode:

```
procedure dpf(n)

F[0] ← 0
F[1] ← 1
for j ← 2 to n do
    F[j] ← F[j − 1] + F[j − 2]
return F[n]
```

- Running time

$$T(n) \in \Theta(n)$$

Have you understood the lecture contents?

well	ok	not-at-all	topic
☐	☐	☐	decision tree lower bound
☐	☐	☐	deriving recurrence
☐	☐	☐	avoiding re-computation
☐	☐	☐	(top-down) memoization
☐	☐	☐	bottom-up — dynamic programming