CS 173, Running Time Analysis, Counting, and Dynamic Programming

Tandy Warnow

CS 173

September 25, 2018 Tandy Warnow

Today

Topics:

- Results from survey
- Midterm
- Very basic counting
- Analyzing running times
- Dynamic programming algorithms

Midterm 1: October 11, 7:15-8:30 PM

- Two rooms, you'll find out which one you're going to soon
- Deadline to request conflict exam October 2
- Most likely conflict exam October 11 morning in the regularly scheduled course meeting

Midterm 1: October 11, 7:15-8:30 PM

- See http://tandy.cs.illinois.edu/173-2018-midterm1-prep.pdf for practice problems
- One weak induction proof
- One strong induction proof
- One proof by contradiction
- Multiple choice problems (covering all course material)

Results from survey

121 students responded
Responses between 1-5, with 1 meaning too slow/easy, 3 about right, 5 meaning too fast/difficult.

- Pace: median 3 [2%, 12%, 49%, 30%, 8%]
- Difficulty: median 3 [1%, 10%, 48%, 31%, 10%]

Very basic counting

In analyzing the running time of an exhaustive search strategy, you need to be able to *count* the number of objects in a set.

Example: Let $S = \{1, 2, \ldots, n\}$.

- How many subsets are there of S? (Equivalently, what is $|\mathbb{P}(S)|$, where $\mathbb{P}(S)$ denotes the power set of S, i.e., the set of all subsets of S?)
- How many non-empty subsets are there of S?
- How many subsets are there of S that contain 1?
- How many subsets are there of S that do not contain 1?
- How many subsets are there of S that contain 1 and 2?
- How many subsets are there of S that do not contain 1 or 2?
- How many ways can we order the elements of S?

Running time analyses

Often algorithms are described recursively or using divide-and-conquer. We will show how to analyze the running times by:

- writing the running time as a recurrence relation
- solving the running time
- proving the running time correct using induction

Analyzing running times

To analyze the running time of an algorithm, we have to know what we are counting, and what we mean.
First of all, we usually want to analyze the **worst case** running time.
This means an upper bound on the total running time for the operation, usually expressed as a function of its input **size**.

Running time analysis

We are going to count **operations**, with each operation having the same cost:

- I/O (reads and writes)
- Numeric operations (additions and multiplications)
- Comparisons between two values
- Logical operations

Running times of algorithms

Now we begin with algorithms, and analyzing their running times.
Let A be an algorithm which has inputs of size n, for $n \geq n_0$.
Let $t(n)$ describe the worst case largest running time of A on input size n.

Running times of recursively defined algorithms

Algorithms that are described recursively typically have the following structure:

- Solve the problem if $n = n_0$; else
  - Preprocessing
  - Recursion to one or more smaller problems
  - Postprocessing

As a result, their running times can be described by recurrence relations of the form

$$t(n_0) = C \text{ (some positive constant)}$$
$$t(n) = f(n) + \sum_{i \in I} t(i) + g(n) \text{ if } n > n_0$$

For the second bullet,

- $f(n)$ is the preprocessing time
- I is a set of dataset sizes that we run the algorithm on recursively
- $g(n)$ is the time for postprocessing

Example of a running time analysis

Consider a recursive algorithm to compute the maximum element in a list of n elements:

- If $n = 1$, return the single element in the list
- Otherwise (for $n \geq 2$)
  - recursively find the maximum entry in the first $n - 1$ elements,
  - then compare it to the last entry in the list and return whichever is larger.

How do we prove this is linear time?

Example of a running time analysis

Let $t(n)$ denote the number of operations used by this algorithm on an input of n values:

- If $n = 1$, return the single element in the list
- Otherwise (for $n \geq 2$)
  - recursively find the maximum entry in the first $n - 1$ elements,
  - then compare it to the last entry in the list and return whichever is larger.

Then $t(n)$ satisfies the recursion:

- $t(1) = C$ for some positive constant C
- $t(n) = t(n-1) + C'$ if $n \geq 2$ for some positive constant C'

We can prove that $t(n) = C'(n-1) + C$ by induction on n.

Another running time analysis

Let's calculate Fibonacci numbers, defined by

- $F(1) = 1$
- $F(2) = 1$
- $F(n) = F(n-1) + F(n-2)$ if $n \geq 3$

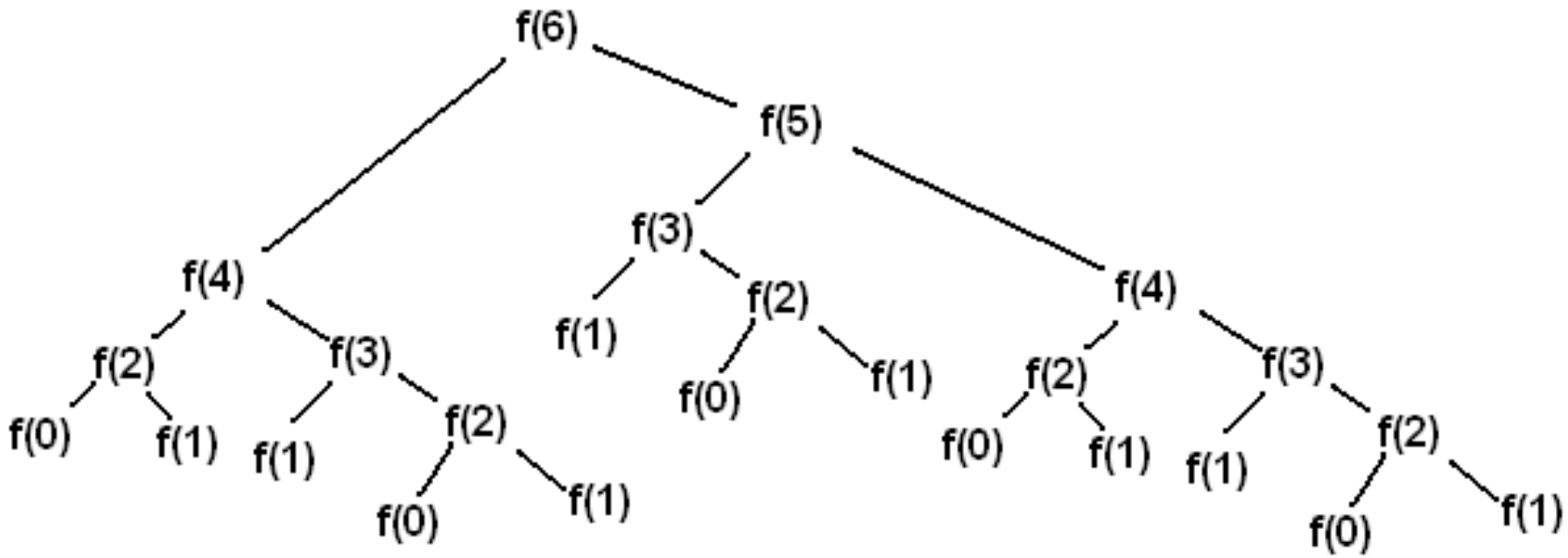
Calculating Fibonacci numbers recursively

Recursive algorithm to compute $F(n)$:

- If ($n = 1$ or $n = 2$) then Return (1)
- Else
  - Recursively compute $F(n-1)$ and store in X
  - Recursively compute $F(n-2)$ and store in Y
  - Return $X + Y$

Class exercise: Try computing $F(10)$ using this approach in 3 minutes.

The recursion tree for the Fibonacci numbers



https://en.wikibooks.org/wiki/Algorithms/Dynamic_Programming

Running time of recursive algorithm

The running time $t_1(n)$ of this algorithm satisfies:

- $t_1(1) = C$
- $t_1(2) = C$
- $t_1(n) = t_1(n-1) + t_1(n-2) + C'$

for some positive integers C, C'.

It's immediately obvious that $t_1(n) \geq F(n)$ for all $n \in \mathbb{Z}^+$ (compare the recurrence relations).

This is a problem, because $F(n)$ grows exponentially (look at http://mathworld.wolfram.com/FibonacciNumber.html), and so $t_1(n)$ grows at least exponentially!

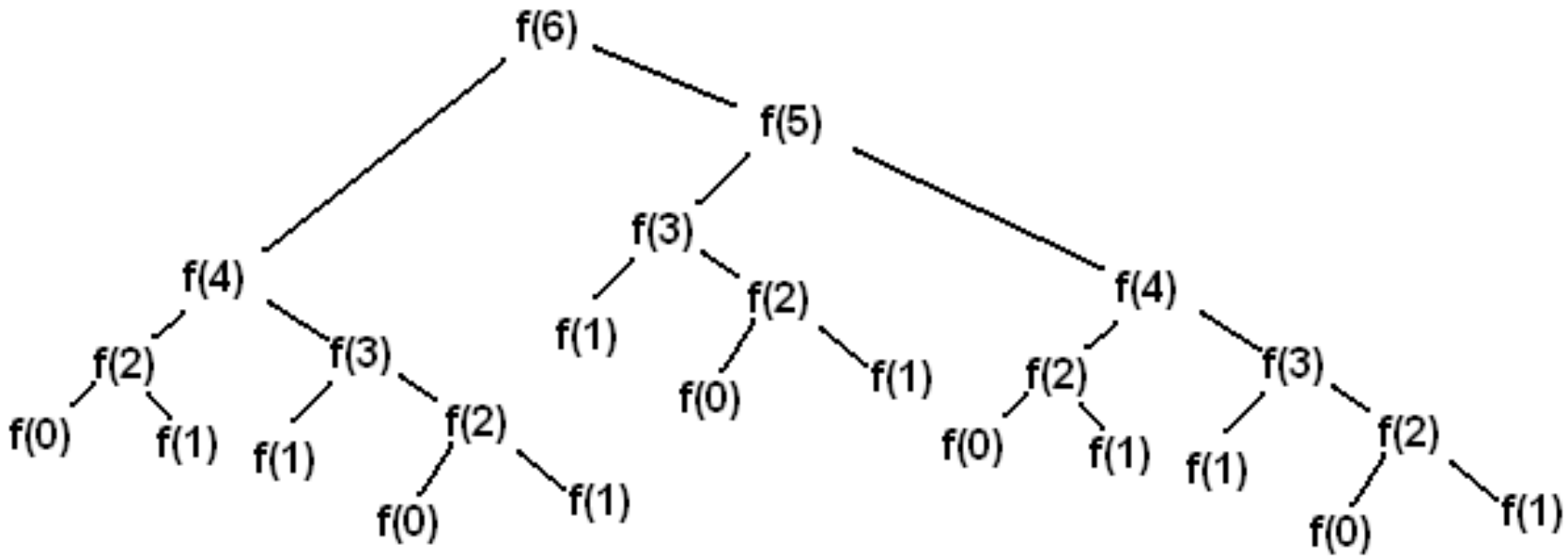
Recursive computation of the Fibonacci numbers

When we compute the Fibonacci numbers recursively, we compute $F(n)$ by independently computing $F(n-1)$ and $F(n-2)$.

Note that $F(n-1)$ also requires that we compute $F(n-2)$, and so $F(n-2)$ is computed twice, rather than once and then re-used.

It would be much better if we had stored the computations for $F(i)$ for smaller values of i (in $\{1, 2, \ldots, n-1\}$) so that they could be re-used.

The recursion tree for the Fibonacci numbers



https://en.wikibooks.org/wiki/Algorithms/Dynamic_Programming

A better way of computing $F(n)$

The simple recursive way of computing $F(n)$ is exponential, but there is a very simple approach that runs in linear time!

Consider computing an array FIB that you fill in from left to right, so the i^{th} entry of FIB is the Fibonacci number $F(i)$.

You return the n^{th} entry of the array!

A better way of computing $F(n)$

Input: $n \in \mathbb{Z}^+$

Output: We will return $F(n)$, the n^{th} Fibonacci number.

- If $n \leq 2$ return 1. Else:
  - $FIB[1] := 1$
  - $FIB[2] := 1$
  - For $i = 3$ up to n, DO
    - $FIB[i] := FIB[i-1] + FIB[i-2]$
  - Return $FIB[n]$

Computing the array $FIB[1...n]$

We fill in the matrix from left to right.

To fill in $FIB[n]$ we have to examine $FIB[n-1]$ and $FIB[n-2]$ and then add them.

Class exercise: Try computing $FIB[10]$ in 3 minutes.

Analyzing the running time

The running time $t_2(n)$ for this algorithm satisfies

- $t_2(1) \leq C_0$
- $t_2(2) \leq C_1$
- $t_2(n) = t_2(n-1) + C_2$ if $n > 2$

for some positive constants C_0, C_1, C_2.
Note the difference in the recursive definition for $t_1(n)$ and $t_2(n)$.

Bounding this recurrence relation

The running time $t_2(n)$ for this algorithm satisfies

- $t_2(1) \leq C_0$
- $t_2(2) \leq C_1$
- $t_2(n) \leq t_2(n-1) + C_2$

for some positive constants C_0, C_1, and C_2.
Let $C' = \max\{C_0, C_1, C_2\}$. It is easy to see that $C' > 0$.

We can prove that $t_2(n) \leq C'n$ for all $n \geq 1$, by induction on n (i.e., linear time).

(Note: even simple induction suffices for this proof.)

In other words, this dynamic programming algorithm is linear time.

In contrast, the recursive algorithm for computing Fibonacci numbres was exponential time!

Dynamic programming vs. Recursion

In other words, this algorithm (filling in the array from left-to-right) uses linear time.

The other algorithm (which used recursion) used exponential time!

The difference is whether you are bottom-up or top-down.

Dynamic programming is bottom-up.

This was a dynamic programming algorithm!

Summary

We

- Showed how to do some basic combinatorial counting.
- Talked about running time analyses, and proving upper bounds on running times using induction.
- Talked about dynamic programming and recursive algorithms.
- Noted that sometimes dynamic programming is faster than recursion!