CS 170: Homework #1

Due on January 26, 2026 at 3:10pm

L. Chen and U. Vazirani, Spring 2026

Zachary Brandt

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, explicitly write "none."

none

2 Course Policies

Please read through the course policies below:

1. The exams will be at the following times. We do not plan on offering alternate exams, so please mark your calendars:
   (a) **Midterm 1**: Thursday, 2/26/2026, 7:00 PM - 9:00 PM
   (b) **Midterm 2**: Thursday, 4/9/2026, 8:00 PM - 10:00 PM
   (c) **Final**: Friday, 5/15/2026, 7:00 PM - 10:00 PM
2. Homework is due Mondays at 10:00pm, with a late deadline at 11:59pm to protect against technical issues. There is no penalty for submitting before the late deadline, but absolutely no submissions will be accepted after 11:59pm.
3. Your lowest homework will be dropped at the end of the semester. By filling out the mid-semester survey and end-of-semester course evaluation, you have the opportunity for two additional drops.
4. Each homework will have one question for you to attempt on your own, labeled "solo question." You will get detailed feedback on your answer to help callibrate you on your learning. On the remaining questions you may collaborate with your fellow students and with AI tools, but you must always list your collaborators and write up the solutions in your own words. (Details in the policies linked below.)
5. The primary source of communication for CS170 will be through Edstem.
6. **Syllabus and Policies:** `https://cs170.org/policies/`
7. **Homework Guidelines:** `https://cs170.org/resources/homework-guidelines/`
8. **Regrade Etiquette:** `https://cs170.org/resources/regrade-etiquette/`
9. **Forum Etiquette:** `https://cs170.org/resources/ed-etiquette/`

Copy and sign the following sentence on your homework submission.
"I have read and understood the course syllabus and policies."

I have read and understood the course syllabus and policies.

3 Asymptotic Complexity Comparisons

Order the following functions so that for all i, j, if f_i comes before f_j in the order then $f_i = O(f_j)$. Do not justify your answers.

- $f_1(n) = 3^n$
- $f_2(n) = n^{\frac{1}{3}}$
- $f_3(n) = 12$
- $f_4(n) = 2^{\log_2 n}$
- $f_5(n) = \sqrt{n}$
- $f_6(n) = 2^n$
- $f_7(n) = \log_2 n$
- $f_8(n) = 2^{\sqrt{n}}$
- $f_9(n) = n^3$
- $f_{10}(n) = \log_3 n$
- $f_{11}(n) = \log^2 n$

As an answer you may just write the functions as a list, e.g. $f_8, f_9, f_1, \ldots$

$f_3, f_{10}, f_7, f_{11}, f_2, f_5, f_4, f_9, f_8, f_6, f_1$

4 Counting Steps (Solo Question)

You can climb a ladder with n rungs by climbing either 1 rung or 2 rungs in each step. How many distinct ways are there to climb to the top?

Part A

Give a simple recursive algorithm to compute the answer.

Solution

Below is a simple recursive algorithm, WAYS(n), to compute how many distinct ways there are to climb to the top of a ladder with n rungs by climbing either 1 rung or 2 rungs in each step:

```
1: function WAYS(n)
2:     if n = 0 then
3:         return 1
4:     else if n < 0 then
5:         return 0
6:     else
7:         return WAYS(n − 1) + WAYS(n − 2)
8:     end if
9: end function
```

Problem 4 continued on next page...

Part B

Prove correctness using induction.

Solution

Below is a proof of correctness using induction on n that the algorithm correctly computes how many distinct ways there are to climb a ladder with n rungs by climbing either 1 rung or 2 rungs in each step.

Proof. **Base cases:**

- If $n = 0$, there is 1 way to climb the ladder, i.e., do nothing, and WAYS(0) returns 1.
- If $n < 0$, there are 0 ways to climb the ladder since it has negative steps, and WAYS(0) returns 0.

Inductive hypothesis: Assume that for all $0 \leq m \leq k$, WAYS(m) outputs the correct result.

Inductive step: For $n = k+1$, the algorithm returns WAYS(k)+WAYS($k-1$). By the inductive hypothesis, WAYS(k) correctly counts ways to reach rung k, from which it is possible to take 1 more step, and WAYS($k-1$) correctly counts ways to reach rung $k-1$, from which we it is possible to take 2 more steps. Their sum then counts all distinct ways there are to reach rung $k+1$, i.e., the top, by climbing either 1 rung or 2 rungs in each step. □

5 Counting Steps Efficiently

Part A

Analyze the running time of your algorithm from the previous question. Is it bounded by a polynomial in n? Assume for simplicity that two integers can be added in one timestep.

Solution

The recurrence relation for my algorithm from the previous question is $T(n) = T(n-1)+T(n-2)$, where $T(n)$ denotes the running time of the algorithm and immediately returns at the base cases $T(0)$ and $T(n)$ for $n < 0$.

Since $T(n-1)$ must be greater than or equal to $T(n-2)$, $T(n)$ is bounded from below by $2T(n-2)$

$$T(n) = T(n-1) + T(n-2) \geq T(n-2) + T(n-2) = 2T(n-2)$$

Through the recurrence relation, the inequality can be "unraveled" to directly compute the running time of the lower bound to $T(n)$

$$T(n) \geq 2T(n-2) \geq 2^2T(n-4) \geq \cdots \geq 2^{\frac{n}{2}}$$

So $T(n)$ grows at least as much as $2^{\frac{n}{2}}$, $T(n) = \Omega(2^{\frac{n}{2}})$. Similarly, since $T(n-2)$ must be less than or equal to $T(n-1)$, $T(n)$ is bounded from below by $2T(n-1)$

$$T(n) = T(n-1) + T(n-2) \leq T(n-1) + T(n-1) = 2T(n-1)$$

This can also be unravled to directly compute the running time of the upper bound to $T(n)$

$$T(n) \leq 2T(n-1) \leq 2^2T(n-2) \geq \cdots \geq 2^n$$

So $T(n)$ grows at least as fast as 2^n, $T(n) = O(2^n)$. These bounds show that the running time for my algorithm from the previous question is not bounded by a polynomial in n, but is rather exponential. The running time grows as fast as an exponential function with a base a between $\sqrt{2}$ (since $2^{\frac{n}{2}} = (2^{\frac{1}{2}})^n = (\sqrt{2})^n$) and 2. Plugging $T(n) = \Theta(a^n)$ into the recurrence relation produces the following

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ a^n &= a^{n-1} + a^{n-2} \\ \frac{a^n}{a^{n-2}} &= \frac{a^{n-1}}{a^{n-2}} + 1 \\ \frac{a^n}{a^n a^{-2}} &= \frac{a^n}{a^n a^{-1}} + 1 \\ a^2 &= a + 1 \end{aligned}$$

Now solving for a will result in the base of the exponential function that determines the running time for my algorithm

$$\begin{aligned} 0 &= a^2 - a - 1 \\ a &= \frac{-(-1) \pm \sqrt{(-1)^2 - 4(1)(-1)}}{2(1)} \\ a &= \frac{1 \pm \sqrt{5}}{2} \end{aligned}$$

where the base then equals $\frac{1+\sqrt{5}}{2}$ which is approximately $a \approx 1.618$ and between $\sqrt{2}$ and 2. Therefore, the running time for my algorithm is

$$T(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

Problem 5 continued on next page...

Part B

Give an efficient algorithm to compute the answer, and show the running time is bounded by $O(n)$ (polynomial in n is also acceptable).

Solution

Below is an efficient algorithm to compute the answer.

```
1: function WAYS(n)
2:     rungs[0] ← 1
3:     rungs[1] ← 1
4:     for i ← 2 to n do
5:         rungs[i] ← rungs[i − 1] + rungs[i − 2]
6:     end for
7:     return rungs[n]
8: end function
```

The above algorithm implements the base cases and recurrence of my previous algorithm iteratively. From the base cases $n = 0$ and $n = 1$, where there is only 1 way to climb the rung in each case, the algorithm builds up the solution for increasing values of n using the recurrence relation $\text{WAYS}(k) = \text{WAYS}(k-1)+\text{WAYS}(k-2)$. The values are maintained in a dictionary for reference in future iterations.

The loop executes $n-2$ times, and each iteration executes accessing, adding, and assigning values in constant time. The initialization of the base cases and returning the result also execute in constant time. So the algorithm achieves linear runtime complexity, $O(1) + (n-2) \times O(1) = O(n)$, by only computing each subproblem once, instead of many times as in my previous algorithm.

Part C

(Extra credit) Give an even more efficient algorithm to compute the answer, and analyze its running time, which should be sub-polynomial, again assuming each integer arithmetic operation takes 1 step. *Hint: can you compute the answer by multiplying* 2×2 *matrices?*

Is this running time a fair assessment of how the algorithm will perform in practice?

Solution

Below is an even more efficient algorithm to compute the answer:

```
1: function WAYS(n)
2:     A ← [ 1  1 ]
           [ 1  0 ]
3:     A ← MATRIXPOWER(A, n + 2)
4:     return A[1, 1]
5: end function
```

The above algorithm produces the correct answer on an arbitrary input of $n \geq 0$. The function leverages matrix multiplication of a 2×2 matrix, A, by computing A^{n+2} and returning the bottom-right entry of the result. This works because A encodes the base cases of the function and computes an update in multiplication

$$\begin{bmatrix} \textsc{Ways}(n+1) & \textsc{Ways}(n) \\ \textsc{Ways}(n) & \textsc{Ways}(n-1) \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} \textsc{Ways}(n+2) & \textsc{Ways}(n+1) \\ \textsc{Ways}(n+1) & \textsc{Ways}(n) \end{bmatrix}$$

and on an arbitrary input of $n \geq 0$, computing A^{n+2} will result in a 2×2 matrix with $\textsc{Ways}(n)$ in its bottom-right corner. Since the base cases $\textsc{Ways}(1)$ and $\textsc{Ways}(0)$ are both equal to 1, A also encodes the base cases of the algorithm.

Each matrix multiplication executes in constant time, with eight multiplications and four additions in the case of 2×2 matrices. However, since MATRIXPOWER uses exponentiation via repeated squaring, it does not execute matrix multiplication $n-2$ times. For example, for an even $n+2$, $\log(n+2)$ matrix multiplications are executed, $A^1, A^2, A^4, A^8 \ldots$, to arrive at the answer. So the running time of this algorithm is logarithmic, $O(\log n)$.

This is not a fair assessment of how the algorithm will perform in practice because for large integers the arithmetic operations will not execute in one step.