Announcements

• From Jan 31 (next mon), all lectures and OH will be in person.
• There will be recording (Panopto or zoom).
• If you feel sick, don’t go to the class.
• Midterm is in person.
  • Feb 4 (next Friday)
  • Open book and notes (hard copies only)
  • Coverage: All topics through divide and conquer
• If you cannot attend the midterm, please contact me ASAP.
• HW4 is out!
HW2 Comments

- What are **not** considered as a proof?
  - You just describe what your algorithm does.
  - Bad: We explores edges in alphabetical order, so the output is correct.
  - You can always look at lecture notes to see how things are proved.
  - Techniques:
    - Induction (Key: Come up with a good hypothesis)
    - Contradiction (If the output is wrong, what do we contradicts to?)

- I changed the guideline to
  - Discuss runtime
  - Prove correctness

- Still, the most important thing is to come up with a right algorithm. You are doing a great job here.
Divide and Conquer Approach
Divide and Conquer

We reduce a problem to several subproblems. Typically, each sub-problem is at most a constant fraction of the size of the original problem.

Recursively solve each subproblem
Merge the solutions

Examples:
• Mergesort, Binary Search, Strassen’s Algorithm,
A Classical Example: Merge Sort

A Classical Example: Merge Sort

A

Split to $n/2$

sort recursively

merge
Why Balanced Partitioning?

An alternative "divide & conquer" algorithm:

• Split into n-1 and 1
• Sort each sub problem
• Merge them

Runtime

\[ T(n) = T(n - 1) + T(1) + n \]

Solution:

\[ T(n) = n + T(n - 1) + T(1) \]
\[ = n + n - 1 + T(n - 2) \]
\[ = n + n - 1 + n - 2 + T(n - 3) \]
\[ = n + n - 1 + n - 2 + \cdots + 1 = O(n^2) \]
Reinventing Mergesort

Suppose we've already invented Bubble-Sort, and we know it takes $n^2$

Try just one level of divide & conquer:

- Bubble-Sort (first $n/2$ elements)
- Bubble-Sort (last $n/2$ elements)

Merge results

Time: $2T(n/2) + n = n^2/2 + n \ll n^2$

Almost twice as fast!
Reinventing Mergesort

• “the more dividing and conquering, the better”
  • Two levels of D&C would be almost 4 times faster, 3 levels almost 8, etc., even though overhead is growing.
  • Best is usually full recursion down to a small constant size (balancing "work" vs "overhead").

In the limit: you’ve just rediscovered mergesort!

• Even unbalanced partitioning is good, but less good
  • Bubble-sort improved with a 0.1/0.9 split:
    \[
    .1n^2 + .9n^2 + n = .82n^2 + n
    \]
    The 18% savings compounds significantly if you carry recursion to more levels, actually giving \(O(n \log n)\), but with a bigger constant.

• This is why Quicksort with random splitter is good – badly unbalanced splits are rare, and not instantly fatal.
  In C++, stdlib do quick sort for \(n > 16\) and insertion sort for \(n \leq 16\).
  See https://www.youtube.com/watch?v=FJJTYQYB1JQ
Finding the Root of a Function
Finding the Root of a Function

Given a continuous function \( f \) and two points \( a < b \) such that
\[
\begin{align*}
    f(a) &\leq 0 \\
    f(b) &\geq 0
\end{align*}
\]
Goal: Find a point \( c \) where \( f(c) \) is close to 0.

\( f \) has a root in \([a, b]\) by
intermediate value theorem

Note that roots of \( f \) may be irrational,
So, we want to approximate
the root with an arbitrary precision!
A Naive Approach

Suppose we want $\varepsilon$ approximation to a root.

Divide $[a, b]$ into $n = \frac{b-a}{\varepsilon}$ intervals. For each interval check $f(x) \leq 0, f(x + \varepsilon) \geq 0$

This runs in time $O(n) = O\left(\frac{b-a}{\varepsilon}\right)$

Can we do faster?
Divide & Conquer (Binary Search)

Bisection \((a, b, \varepsilon)\)

\[\text{if } (b - a) < \varepsilon \text{ then}
\]
\[\quad \text{return } a;\]
\[\text{else}
\]
\[\quad m \leftarrow (a + b)/2;\]
\[\quad \text{if } f(m) \leq 0 \text{ then}
\]
\[\quad \quad \text{return Bisection}(m, b, \varepsilon);\]
\[\quad \text{else}
\]
\[\quad \quad \text{return Bisection}(a, m, \varepsilon);\]
Time Analysis

Let $n = \frac{b-a}{\epsilon}$ be the # of intervals and $c = (a + b)/2$

Always half of the intervals lie to the left and half lie to the right of $c$

So,

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

i.e., $T(n) = O(\log n) = O(\log(\frac{b-a}{\epsilon}))$

For $d$ dimension, "Binary search" can be used to minimize convex functions. The current best algorithms take $O(d^3 \log^{O(1)}(d/\epsilon))$ time.
Fast Exponentiation
Fast Exponentiation

• **Power**(*a*, *n*)
  
  **Input:** integer *n* ≥ 0 and number *a*
  
  **Output:** *a*^*n*

• Obvious algorithm
  
  *n* − 1 multiplications

• Observation:
  
  if *n* is even, then *a*^*n* = *a*^*n/2* · *a*^*n/2*.
Divide & Conquer (Repeated Squaring)

\[
\text{Power}(a, n) \begin{cases} 
\text{if } (n = 0) & \text{return 1} \\
\text{else if } (n \text{ is even}) & \text{return Power}(a, n/2) \cdot \text{Power}(a, n/2) \\
\text{else} & \text{return Power}(a, (n - 1)/2) \cdot \text{Power}(a, (n - 1)/2) \cdot a 
\end{cases}
\]

Is there any problem in the program?

\text{Time (# of multiplications):}
\begin{align*}
T(n) & \leq T(\lfloor n/2 \rfloor) + 2 \text{ for } n \geq 1 \\
T(0) & = 0
\end{align*}

Solving it, we have
\begin{align*}
T(n) & \leq T(\lfloor n/2 \rfloor) + 2 \leq T(\lfloor n/4 \rfloor) + 2 + 2 \\
& \leq \cdots \leq T(1) + 2 + \cdots + 2 \leq 2 \log_2 n.
\end{align*}
Problem 4 (20 points).
Given an array of positive numbers $a = [a_1, a_2, \ldots, a_n]$ . Give an $O(n \log n)$ time algorithm that find $i$ and $j$ (with $i \leq j$) that maximize the subarray product $\prod_{k=i}^{j} a_k$. Prove the correctness and the runtime of the algorithm.

For example, in the array $a = [3, 0.2, 5, 7, 0.4, 4, 0.01]$, the sub-array from $i = 3$ to $j = 6$ has the product $5 \times 7 \times 0.4 \times 4 = 56$ and no other sub-array contains elements that product to a value greater than 56. So, the answer for this input is $i = 3, j = 6$.

Hints: Divide and Conquer.
Quiz

Algorithm

\[ \text{function } (i, j) = \text{MAXSUB}(a_1, a_2, \cdots, a_n) \]

- If \( n = 1 \)
  - Output \( i = j = 1 \).

- Else
  - \((i_1, j_1) = \text{MAXSUB}(a_1, \cdots, a_{\lfloor n/2 \rfloor})\).
  - \((i_2, j_2) = \text{MAXSUB}(a_{\lfloor n/2 \rfloor + 1}, \cdots, a_n)\).
  - Find \( i_3 \leq \lfloor n/2 \rfloor \) that maximize \( \prod_{k=i_3}^{\lfloor n/2 \rfloor} a_k \).
  - Find \( j_3 > \lfloor n/2 \rfloor \) that maximize \( \prod_{k=\lfloor n/2 \rfloor + 1}^{j_3} a_k \).
  - Compare the subarray product for \((i_1, j_1), (i_2, j_2)\) and \((i_3, j_3)\) and output the one with the largest subarray product.

Runtime

The runtime satisfies \( T(n) = 2T(n/2) + O(n) \). So, we have \( T(n) = O(n \log n) \).
Correctness

Induction statement: “The algorithm is correct for input size \( \leq n \)”

Base case \( n = 1 \): The algorithm is correct because \( i = j = 1 \) is the only possible output.

Inductive step:
Case 1: \( j \leq \lfloor n/2 \rfloor \).

The algorithm finds the solution from the first sub-problem (due to the induction hypothesis).

Case 2: \( i > \lfloor n/2 \rfloor \).

The algorithm finds the solution from the second sub-problem (due to the induction hypothesis).

Case 3: \( i \leq \lfloor n/2 \rfloor \) and \( j > \lfloor n/2 \rfloor \).

Note that \( \prod_{k=i}^j a_k = \prod_{k=i}^{\lfloor n/2 \rfloor} a_k \times \prod_{k=\lfloor n/2 \rfloor}^j a_k \). Since \( i \) and \( j \) maximize the left hand side, \( i \) must be the maximizer of \( \prod_{k=i}^{\lfloor n/2 \rfloor} a_k \) and \( j \) must be the maximizer of \( \prod_{k=\lfloor n/2 \rfloor}^j a_k \).

Therefore, the algorithm correctly finds it in this case.
Master Theorem
Master Theorem

Suppose \( T(n) = a \cdot T\left(\frac{n}{b}\right) + cn^k \) for all \( n > b \). Then,

- If \( a < b^k \) then \( T(n) = \Theta(n^k) \)

- If \( a = b^k \) then \( T(n) = \Theta(n^k \log n) \)

- If \( a > b^k \) then \( T(n) = \Theta(n^{\log_b a}) \)

Works even if it is \( \left\lfloor \frac{n}{b} \right\rfloor \) instead of \( \frac{n}{b} \).

We also need \( a \geq 1, b > 1, k \geq 0 \).
Master Theorem

Suppose $T(n) = a \, T\left(\frac{n}{b}\right) + cn^k$ for all $n > b$. Then,

- If $a < b^k$ then $T(n) = \Theta(n^k)$
- If $a = b^k$ then $T(n) = \Theta(n^k \log n)$
- If $a > b^k$ then $T(n) = \Theta(n^{\log_b a})$

**Example:** For mergesort algorithm we have

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n).$$

So, $k = 1, a = b^k$ and $T(n) = \Theta(n \log n)$
Proving Master Theorem

Problem size

\[ T(n) = aT\left(\frac{n}{b}\right) + cn^k \]

# probs

| \(a\) | \(a \cdot c\left(\frac{n}{b}\right)^k\) |
| 1   | \(cn^k\) |

| \(a^2\) | \(a^2 \cdot c\left(\frac{n}{b^2}\right)^k\) |
| \(a^d\) | \(a^d \cdot c\left(\frac{n}{b^d}\right)^k\) |

\[ T(n) = \sum_{i=0}^{d=\log_b n} a^i c \left(\frac{n}{b^i}\right)^k \]
Master Theorem

Suppose $T(n) = a\ T\left(\frac{n}{b}\right) + cn^k$ for all $n > b$. Then,

- If $a < b^k$ then $T(n) = \Theta(n^k)$ \hspace{1cm} \# of problems increases \textbf{slower} than the decreases of cost. \textbf{First} term dominates.
- If $a = b^k$ then $T(n) = \Theta(n^k \log n)$
- If $a > b^k$ then $T(n) = \Theta(n^{\log_b a})$ \hspace{1cm} \# of problems increases \textbf{faster} than the decreases of cost \textbf{Last} term dominates.