Announcements

C++ Bootcamp next week
   Watch all videos on edX

PS #1 is out
   Gradescope
   Office Hours

Textbook
Previously ...

Analysis of Algorithms

running times (empirical analysis)

examples
Today …

Analysis of Algorithms
running times (mathematical models)
big O notation
examples
Math Review

Summations

Powers

Logarithms

Proof Techniques

Basic Probability and Combinatorics
Math Review

Summations

Powers

Logarithms

Proof Techniques

Basic Probability and Combinatorics
Time Complexity
Analyzing Running Time

**Empirical Analysis**
- Run algorithm
- Measure actual time

**Mathematical Model**
- Analyze Algorithm
- Develop Model
Mathematical Model

High-level analysis — no need to implement running time $T(n)$ is a function of the input size $n$
Mathematical Model

High-level analysis — no need to implement running time $T(n)$ is a function of the input size $n$ Independent of HW/SW
Mathematical Model

High-level analysis — no need to implement running time $T(n)$ is a function of the input size $n$

Independent of HW/SW

Based on counts of elementary operations
additions, multiplications, comparisons, etc.

exact definition not important
must be ‘relevant’ to the problem
Growth Rate

Changes in HW/SW affect $T(n)$ by a constant factor do not alter the growth rate of $T(n)$.
Growth Rate

Changes in HW/SW
affect $T(n)$ by a constant factor
do not alter the growth rate of $T(n)$

Interest on execution times with large inputs
Growth Rate

Changes in HW/SW
affect $T(n)$ by a constant factor
do not alter the growth rate of $T(n)$

Interest on execution times with large inputs

growth of $T(n)$ as $n \rightarrow \infty$
Order of Growth

\[
T(n) = \Theta(1)
\]
\[
T(n) = \Theta(\log n)
\]
\[
T(n) = \Theta(n)
\]
\[
T(n) = \Theta(n \log n)
\]
\[
T(n) = \Theta(n^2)
\]
\[
T(n) = \Theta(n^3)
\]
\[
T(n) = \Theta(2^n)
\]
Asymptotic Performance

For large values of $n$, an $O(n^2)$ algorithm always beats an $O(n^3)$ algorithm.
Asymptotic Performance

For large values of $n$, an $O(n^2)$ algorithm always beats an $O(n^3)$ algorithm.

In practice, we shouldn’t completely ignore asymptotically slower algorithms.
Big O Notation
\( T(n) = O(f(n)) \) if there are positive constants \( c \) and \( n_0 \) such that 
\[ T(n) \leq c \cdot f(n) \] 
when \( n \geq n_0 \)
$T(n) = O(f(n))$ if there are positive constants $c$ and $n_0$ such that $T(n) \leq c \cdot f(n)$ when $n \geq n_0$
$T(n) = O(f(n))$ if there are positive constants $c$ and $n_0$ such that $T(n) \leq c \cdot f(n)$ when $n \geq n_0$
$T(n) = O(f(n))$ if there are positive constants $c$ and $n_0$ such that $T(n) \leq c \cdot f(n)$ when $n \geq n_0$
Example 1  $7n - 2$ is $O(n)$

Proof 1  A possible choice is $c = 7$ and $n_0 = 1$.

Example 2  $20n^3 + 10n \log n + 5$ is $O(n^3)$

Proof 2  $c = 35$ and $n_0 = 1$.

Example 3  $3 \log n + \log \log n$ is $O(\log n)$

Proof 3  $c = 4$ and $n_0 = 2$.

Example 4  $2^{100}$ is $O(1)$

Proof 4  $c = 2^{100}$ and $n_0 = 1$. 
Example 1 \(7n - 2 \text{ is } O(n)\)

Proof 1 A possible choice is \(c = 7\) and \(n_0 = 1\).

Example 2 \(20n^3 + 10n \log n + 5 \text{ is } O(n^3)\)

Proof 2 \(c = 35\) and \(n_0 = 1\).

Example 3 \(3 \log n + \log \log n \text{ is } O(\log n)\)

Proof 3 \(c = 4\) and \(n_0 = 2\).

Example 4 \(2^{100} \text{ is } O(1)\)

Proof 4 \(c = 2^{100}\) and \(n_0 = 1\).

7n-2 is also \(O(n^2), O(n^3), O(2^n), \ldots\), but tighter bound is preferred.
Example 1 \[7n - 2 \text{ is } O(n)\]

Proof 1 A possible choice is \(c = 7\) and \(n_0 = 1\).

Example 2 \[20n^3 + 10n \log n + 5 \text{ is } O(n^3)\]

Proof 2 \(c = 35\) and \(n_0 = 1\).

Example 3 \[3 \log n + \log \log n \text{ is } O(\log n)\]

Proof 3 \(c = 4\) and \(n_0 = 2\).

Example 4 \[2^{100} \text{ is } O(1)\]

Proof 4 \(c = 2^{100}\) and \(n_0 = 1\).

7n-2 is also \(O(n^2)\), \(O(n^3)\), \(O(2^n)\), …, but tighter bound is preferred

“ignore constants and drop lower order terms”
Big Omega

\[ T(n) = \Omega(f(n)) \text{ if there are positive constants } c \text{ and } n_0 \text{ such that } T(n) \geq c \cdot f(n) \text{ when } n \geq n_0 \]
$T(n) = \Omega(f(n))$ if there are positive constants $c$ and $n_0$ such that $T(n) \geq c \cdot f(n)$ when $n \geq n_0$
Big Omega

\[ T(n) = \Omega(f(n)) \] if there are positive constants \( c \) and \( n_0 \) such that
\[ T(n) \geq c \cdot f(n) \text{ when } n \geq n_0 \]
Big Omega

$T(n) = \Omega(f(n))$ if there are positive constants $c$ and $n_0$ such that $T(n) \geq c \cdot f(n)$ when $n \geq n_0$
Big Theta

\[ T(n) = \Theta(f(n)) \text{ if and only if } T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)) \]
Big Theta

\[ T(n) = \Theta(f(n)) \text{ if and only if } T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)) \]
Big Theta

\[ T(n) = \Theta(f(n)) \text{ if and only if } T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)) \]
Big Theta

\[ T(n) = \Theta(f(n)) \text{ if and only if } T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n)) \]
Big Theta

\[ T(n) = \Theta(f(n)) \text{ if and only if } \begin{align*}
T(n) &= \mathcal{O}(f(n)) \text{ and } T(n) = \Omega(f(n))
\end{align*} \]
Example 1 \[3 \log n + \log \log n \text{ is } \Omega(\log n)\]

Proof 1 \[3 \log n + \log \log n \geq 3 \log n, \text{ for } n_0 = 2\]

Example 2 \[3 \log n + \log \log n \text{ is } \Theta(\log n)\]

Proof 2 \[3 \log n + \log \log n \text{ is } \Omega(\log n) \text{ and } O(\log n)\]
Examples

$10^2 + 3000n + 10$

$21 \log n$

$500 \log n + n^4$

$\sqrt{n} + \log n^{50}$

$4^n + n^{5000}$

$3000n^3 + 3n^{3.5}$

$2^5 + n!$

Big O?

Big Omega?

Big Theta?
for (int i = 0 ; i < n ; i ++) {
    // O(1) operations
}

for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < n ; j ++) {
        // O(1) operations
    }
}

for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < n*n ; j ++) {
        // O(1) operations
    }
}

T(n)=


for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < i ; j ++) {
        // O(1) operations
    }
}

for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < n ; j ++) {
        for (int k = 0 ; k < n ; k ++) {
            // O(1) operations
        }
    }
}

for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < i*i ; j ++) {
        for (int k = 0 ; k < j ; k ++) {
            // O(1) operations
        }
    }
}
Case Study
Maximum Subarray Problem

Given an array of \( n \) integers, find the subarray \( A[j:k] \) that maximizes the sum

\[
s_{j,k} = a_j + a_{j+1} + \cdots + a_k = \sum_{i=j}^{k} a_i.
\]
Brute Force

Algorithm MaxsubSlow\((A)\):

**Input:** An \(n\)-element array \(A\) of numbers, indexed from 1 to \(n\).

**Output:** The maximum subarray sum of array \(A\).

\(m \leftarrow 0 \) // the maximum found so far

for \(j \leftarrow 1\) to \(n\) do

  for \(k \leftarrow j\) to \(n\) do
    \(s \leftarrow 0\) // the next partial sum we are computing

    for \(i \leftarrow j\) to \(k\) do
      \(s \leftarrow s + A[i]\)

    if \(s > m\) then
      \(m \leftarrow s\)

return \(m\)
Brute Force

Algorithm MaxsubSlow(A):

\textbf{Input:} An $n$-element array $A$ of numbers, indexed from 1 to $n$.
\textbf{Output:} The maximum subarray sum of array $A$.

$m \leftarrow 0$ \hspace{1em} // the maximum found so far

\textbf{for} $j \leftarrow 1$ \textbf{to} $n$ \textbf{do}

\hspace{1em} \textbf{for} $k \leftarrow j$ \textbf{to} $n$ \textbf{do}

\hspace{2em} $s \leftarrow 0$ \hspace{1em} // the next partial sum we are computing

\hspace{2em} \textbf{for} $i \leftarrow j$ \textbf{to} $k$ \textbf{do}

\hspace{3em} $s \leftarrow s + A[i]$

\hspace{2em} \textbf{if} $s > m$ \textbf{then}

\hspace{3em} $m \leftarrow s$

\textbf{return} $m$

$O(n^3)$

From Algorithm Design and Applications, Goodrich & Tamassia
Using Prefix Sums

Algorithm MaxsubFaster(A):

Input: An n-element array A of numbers, indexed from 1 to n.
Output: The maximum subarray sum of array A.

\[ S_0 \leftarrow 0 \quad \text{// the initial prefix sum} \]

for \( i \leftarrow 1 \) to \( n \) do

\[ S_i \leftarrow S_{i-1} + A[i] \]

\( m \leftarrow 0 \quad \text{// the maximum found so far} \]

for \( j \leftarrow 1 \) to \( n \) do

for \( k \leftarrow j \) to \( n \) do

\[ s = S_k - S_{j-1} \]

if \( s > m \) then

\[ m \leftarrow s \]

return \( m \)
Using Prefix Sums

Algorithm MaxsubFaster(A):

*Input:* An \( n \)-element array \( A \) of numbers, indexed from 1 to \( n \).

*Output:* The maximum subarray sum of array \( A \).

\[
S_0 \leftarrow 0 \quad \text{// the initial prefix sum}
\]

for \( i \leftarrow 1 \) to \( n \) do

\[
S_i \leftarrow S_{i-1} + A[i]
\]

\( m \leftarrow 0 \quad \text{// the maximum found so far} \)

for \( j \leftarrow 1 \) to \( n \) do

for \( k \leftarrow j \) to \( n \) do

\[
s = S_k - S_{j-1}
\]

if \( s > m \) then

\[
m \leftarrow s
\]

return \( m \)

Any summation \( S_{j,k} \) can be computed in constant time.
Using Prefix Sums

Algorithm MaxsubFaster(A):

**Input:** An $n$-element array $A$ of numbers, indexed from 1 to $n$.

**Output:** The maximum subarray sum of array $A$.

$S_0 \leftarrow 0$  // the initial prefix sum

for $i \leftarrow 1$ to $n$ do
    $S_i \leftarrow S_{i-1} + A[i]$

$m \leftarrow 0$  // the maximum found so far

for $j \leftarrow 1$ to $n$ do
    for $k \leftarrow j$ to $n$ do
        $s = S_k - S_{j-1}$
        if $s > m$ then
            $m \leftarrow s$

return $m$

Any summation $s_{j,k}$ can be computed in constant time

$$s_{j,k} = S_k - S_{j-1}$$
Using Prefix Sums

Algorithm MaxsubFaster(A):

Input: An \( n \)-element array \( A \) of numbers, indexed from 1 to \( n \).
Output: The maximum subarray sum of array \( A \).

\[ S_0 \leftarrow 0 \quad \text{// the initial prefix sum} \]

for \( i \leftarrow 1 \) to \( n \) do

\[ S_i \leftarrow S_{i-1} + A[i] \]

\[ m \leftarrow 0 \quad \text{// the maximum found so far} \]

for \( j \leftarrow 1 \) to \( n \) do

\[ s = S_k - S_{j-1} \]

if \( s > m \) then

\[ m \leftarrow s \]

return \( m \)

Any summation \( s_{j,k} \) can be computed in constant time

\[ s_{j,k} = S_k - S_{j-1} \]

\[ O(1+2+\ldots+n) \]

\[ O(n^2) \]
Using Maximum Suffix Sums

**Algorithm** MaxsubFastest($A$):

- **Input:** An $n$-element array $A$ of numbers, indexed from 1 to $n$.
- **Output:** The maximum subarray sum of array $A$.

$M_0 \leftarrow 0$ // the initial prefix maximum

for $t \leftarrow 1$ to $n$ do

\[ M_t \leftarrow \max\{0, M_{t-1} + A[t]\} \]

$m \leftarrow 0$ // the maximum found so far

for $t \leftarrow 1$ to $n$ do

\[ m \leftarrow \max\{m, M_t\} \]

return $m$
Using Maximum Suffix Sums

Mt is the summation value of a maximum subarray that ends at t

Algorithm MaxsubFastest(A):

Input: An n-element array A of numbers, indexed from 1 to n.
Output: The maximum subarray sum of array A.

M₀ ← 0 // the initial prefix maximum
for t ← 1 to n do
    Mₜ ← max{0, Mₜ₋₁ + A[t]}
m ← 0 // the maximum found so far
for t ← 1 to n do
    m ← max{m, Mₜ}
return m

From Algorithm Design and Applications, Goodrich & Tamassia
Using Maximum Suffix Sums

Algorithm MaxsubFastest(A):

\textbf{Input:} An \( n \)-element array \( A \) of numbers, indexed from 1 to \( n \).

\textbf{Output:} The maximum subarray sum of array \( A \).

\( M_0 \leftarrow 0 \)  // the initial prefix maximum

\textbf{for} \( t \leftarrow 1 \) \textbf{to} \( n \) \textbf{do}

\hspace{1em} \( M_t \leftarrow \max\{0, M_{t-1} + A[t]\} \)

\( m \leftarrow 0 \)  // the maximum found so far

\textbf{for} \( t \leftarrow 1 \) \textbf{to} \( n \) \textbf{do}

\hspace{1em} \( m \leftarrow \max\{m, M_t\} \)

\textbf{return} \( m \)

\( M_t \) is the summation value of a maximum subarray that ends at \( t \)

\( M_t = \max\{0, \max_{j=1,\ldots,t} \{ s_{j,t} \} \} \)

From Algorithm Design and Applications, Goodrich & Tamassia
Using Maximum Suffix Sums

Algorithm MaxsubFastest(A):

*Input:* An $n$-element array $A$ of numbers, indexed from 1 to $n$.

*Output:* The maximum subarray sum of array $A$.

1. $M_0 \leftarrow 0$  // the initial prefix maximum
2. for $t \leftarrow 1$ to $n$ do
   1. $M_t \leftarrow \max\{0, M_{t-1} + A[t]\}$
   2. $m \leftarrow 0$  // the maximum found so far
3. for $t \leftarrow 1$ to $n$ do
   1. $m \leftarrow \max\{m, M_t\}$
4. return $m$

$M_t$ is the summation value of a maximum subarray that ends at $t$

$$M_t = \max\{0, \max_{j=1,\ldots,t}\{s_{j,t}\}\}$$

$$M_t = \max\{0, M_{t-1} + A[t]\}$$

From Algorithm Design and Applications, Goodrich & Tamassia
Using Maximum Suffix Sums

**Algorithm MaxsubFastest(A):**

- **Input:** An $n$-element array $A$ of numbers, indexed from 1 to $n$.
- **Output:** The maximum subarray sum of array $A$.

1. $M_0 \leftarrow 0$ // the initial prefix maximum
2. for $t \leftarrow 1$ to $n$ do
   - $M_t \leftarrow \max\{0, M_{t-1} + A[t]\}$
   - $m \leftarrow 0$ // the maximum found so far
3. for $t \leftarrow 1$ to $n$ do
   - $m \leftarrow \max\{m, M_t\}$
4. return $m$

$M_t$ is the summation value of a maximum subarray that ends at $t$

$$M_t = \max\{0, \max_{j=1,\ldots,t} \{s_{j,t}\}\}$$

$$M_t = \max\{0, M_{t-1} + A[t]\}$$

$O(n)$