Midterm Review

CS16: Introduction to Data Structures & Algorithms
Spring 2018
Outline

- Algorithm Analysis
  - Big-O, amortized, expected
- Selection & Medians
- Dynamic Programming
- Induction
Algorithm Analysis

- What is the purpose of runtime analysis?
  - quantify/measure how “good” an algorithm is
  - compare algorithms
  - allows us to pick the best algorithm for the job
- In CS16 “good” means efficient
What is an “Efficient” Algorithm

› Possible efficiency measures
  › Total amount of time on a stopwatch?
  › Low memory usage?
  › Low power consumption?
  › Network usage?

› The analysis of algorithms helps us quantify this
Algorithm Analysis

‣ “Count” # of operations as function of input size
  ‣ If input has size \( n \), how many operations will Alg. take?
  ‣ results in a function \( T(n) \) which is Alg.’s runtime

‣ Problem 1
  ‣ What if # of operations changes depending on input?
  ‣ Then each input has a different runtime...
  ‣ …which is the “right” one?
Algorithm Analysis: Worst-Case

- Assignment to index (Step 4)
  - executed depending on the contents of A
- Worst-case analysis: max runtime (over all inputs)
Algorithm Analysis

- “Count” # of operations as function of input size
  - If input has size $n$, how many operations will Alg. take?
  - results in a function $T(n)$ which is Alg.’s runtime

Problem 1

- What if # of operations changes depending on input?
  - use the worst-case runtime (over all the inputs)

Problem 2

- what if runtime $T(n)$ is very complicated?
  - use Big-O to simplify it
Definition (Big-O): \( f(n) \) is \( O(g(n)) \) if there exists positive constants \( c \) and \( n_0 \) such that:

\[
f(n) \leq c \cdot g(n)
\]

for all \( n \geq n_0 \)

- Example: \( 2n+10 \) is \( O(n) \)
  - for example, choose \( c=3 \) and \( n_0=10 \)
- Why? because
  - \( 2n+10 \leq 3 \cdot n \) when \( n \geq 10 \)
  - for example, \( 2 \cdot 10+10 \leq 3 \cdot 10 \)
Summary of Big-O Rules

- If $f(n)$ is a polynomial of degree $d$ then
  - $f(n)$ is $O(n^d)$
- In other words you can ignore
  - lower-order terms
  - constant factors
- Use the term with the smallest possible degree
  - $2n$ is $O(n^{50})$ but that’s not helpful
  - instead it is better to say it is $O(n)$
- **Discard constant factors & use smallest possible degree**
Big-O & Growth Rate

- Big-O gives upper bound on
  - growth rate of function when input is large
- An algorithm is $O(g(n))$ if growth its rate is
  - no more than growth rate of $g(n)$
- Examples
  - $n^2$ is not $O(n)$
  - $n$ is $O(n^2)$
  - $n^2$ is $O(n^3)$
Big-O

- Another example
  - $n^2$ is not $O(n)$
  - Why? To prove that $n^2$ is $O(n)$ we have to show that there exists constants $c$ and $n_0$ such that
    - $n^2 \leq c \cdot n \iff n \leq c$ for all $n \geq n_0$
  - This is not possible!
    - for example set $c = 10$
Big-Omega

Definition (Big-$\Omega$): $f(n)$ is $\Omega(g(n))$ if there exists positive constants $c$ and $n_0$ such that:

$$f(n) \leq c \cdot g(n)$$

for all $n \geq n_0$

- $f(n)$’s growth rate is upper bounded by $g(n)$’s growth rate
- But what if we need to express a lower bound?
  - we use Big-$\Omega$ notation!
Big-Omega

Definition (Big-$\Omega$): $f(n)$ is $\Omega(g(n))$ if there exists positive constants $c$ and $n_0$ such that:

$$f(n) \geq c \cdot g(n)$$

for all $n \geq n_0$

- $f(n)$’s growth rate is lower bounded by $g(n)$’s growth rate
- What about an upper and a lower bound?
- We use Big-$\Omega$ notation
Big-Theta

Definition (Big-$\Theta$): $f(n)$ is $\Theta(g(n))$ if it is $O(g(n))$ and $\Omega(g(n))$.

- $f(n)$’s growth rate is the same as $g(n)$’s
## More Examples

<table>
<thead>
<tr>
<th>( f(n) )</th>
<th>Big-O</th>
<th>Big-Ω</th>
<th>Big-P</th>
</tr>
</thead>
<tbody>
<tr>
<td>( an + b )</td>
<td>?</td>
<td>?</td>
<td>( P(n) )</td>
</tr>
<tr>
<td>( an^2 + bn + c )</td>
<td>?</td>
<td>?</td>
<td>( P(n^2) )</td>
</tr>
<tr>
<td>( a )</td>
<td>?</td>
<td>?</td>
<td>( P(1) )</td>
</tr>
<tr>
<td>( 3^n + an^{40} )</td>
<td>?</td>
<td>?</td>
<td>( P(3^n) )</td>
</tr>
<tr>
<td>( an + b \log n )</td>
<td>?</td>
<td>?</td>
<td>( P(n) )</td>
</tr>
</tbody>
</table>
Running Times

- $O(1)$: independent of input size
- $O(n)$: depends on input size
- $O(n^2)$: depends on square of input size
- $O(n^3)$: depends on cube of input size
- $O(n^{70})$: depends on 70th power of input size
- $O(2^n)$: exponential in input size
Expandable Stack

- Capped-capacity stack is fast
  - but not useful in practice
- How can we design an uncapped Stack?
- Strategy: **Doubling**
  - double size of array when full
Expandable Stack

Stack( ):
   data = array of size 20
   count = 20
   capacity = 20

function push(object):
   data[count] = object
   count++
   if count == capacity
      new_capacity = capacity * 2 /* doubling */
      new_data = array of size new_capacity
      for i = 0 to capacity - 1
         new_data[i] = data[i]
      capacity = new_capacity
      data = new_data

Run time depends on count/history
Doubling

- What is the running time of doubling?
  - $O(1)$ or $O(n)$?
- It depends on the value of `count`…
- …the value of `count` depends on # of previous pushes

Measure cost on sequence of pushes not a single push!
Towards Amortized Analysis

- For certain algorithms better to measure
  - total running time on sequence of operations
  - instead of running time on single operation
  - \( T(n) \): total cost on sequence of \( n \) operations
  - **Not running time on a single input**
- Usually the case for data structure operations
- ex: Stack
  - \( T(n) \): cost push \#1 + cost push \#2 + \ldots + cost push \#n
Amortized Analysis

- Instead of reporting **total** cost of sequence
  - report cost of sequence **per operation**

\[
\frac{T(n)}{n}
\]
Amortized Analysis of Doubling

- ex: doubling stack with initial capacity 5?
  - pushes are $O(1)$ until 5th push
  - then $O(n)$

\[
\frac{T(n)}{n} : \frac{T(5)}{5} = \frac{5 + 5}{5} = 2
\]

\[
\frac{T(n)}{n} : \frac{T(10)}{10} = \frac{10 + 5 + 10}{10} = 2.5
\]

\[
\frac{T(n)}{n} : \frac{T(20)}{20} = \frac{20 + 5 + 10 + 20}{20} = 2.75
\]
Amortized Analysis of Doubling

\[ T(n) = n + n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^{k-1}} \]

\[ = n + n \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{k-1}}\right) \]

\[ < n + n \cdot 2 \]

\[ = 3n \]

\[ \frac{T(n)}{n} = O(1) \]
Amortized Analysis

- Summary for Doubling
  - Total cost of $n$ pushes: $T(n) = O(n)$
  - Amortized cost of $n$ pushes: $T(n)/n = O(1)$
Analysis

- When do we use worst-case analysis?
- When do we use amortized analysis?
Randomized Algorithms

- Some algorithms depend on a random “object”
  - The setup algorithm of hash tables with UHF
    - choose tuple of 4 numbers \((a_1, a_2, a_3, a_4)\)
  - Quicksort & quickselect
    - choose pivot at random

- Runtime changes depending on random object
  - “bad” random objects: random objects that lead to slow runtime
  - “good” random objects: random objects that lead to fast runtime

- What does a worst-case runtime tell us?
  - runtime for the worst input and the worst random object
Randomized Algorithms

- What if “bad” random objects are very rare…
  - …and “good” random objects are very common?

- Then worst-case runtime is misleading…
  - …because worst-case runtime is rare

- Better idea
  - execute algorithm a large number of times and report the average of the runtimes
    - let’s do that “analytically”!

- This is the expected runtime
Analysis

- When do we use worst-case analysis?
- When do we use amortized analysis?
- When do we use expected runtime?
Outline

- Algorithm Analysis
  - Big-O, amortized, expected
- Selection & Medians
- Dynamic Programming
- Induction
Selection

- Selection
  - given an unsorted set $L$ and a rank $k$
  - output the $k$th smallest element in set
  - set can be represented as list, array, …
Quickselect (Hoare’s Selection)

- Divide and conquer
  - divide: pick random element (called pivot) and partition set into
    - **L**: elements less than x
    - **E**: elements equal to x
    - **G**: elements larger than x
  - recur:
    - if \( k \leq |L| \): call `quickselect(L,k)`
    - if \( |L| < k \leq |L| + |E| \): return x
    - if \( k > |L| + |E| \): call `quickselect(G, k - (|L| + |E|))`
  - conquer: return
Quickselect Pseudo-code

```python
quickselect(list, k):
    if list has 1 element return it
    pivot = list[rand(0, list.size)]
    L = []   E = []   G = []
    for x in list:
        if x < pivot: L.append(x)
        if x == pivot: E.append(x)
        if x > pivot: G.append(x)
    if k <= L.size:
        return quickselect(L, k)
    else if k <= (L.size + E.size):
        return pivot
    else
        return quickselect(G, k - (L.size + E.size))
```
Quickselect Analysis

- How fast is Quickselect?
  - same as Quicksort
  - $O(n^2)$ run time if we pick the wrong pivots

- We use expected runtime since algorithm is randomized

- We assume all elements are distinct (worst-case)
  - if list has more than one copy of pivot,
  - it would shrink the sub-lists and improve runtime
Quickselect Analysis

- Each pivot has equal probability of being chosen
- Each pivot splits list in two lists
  - one of size $i$
  - and one of size $n-1-i$
- Recurrence relation now has form
  \[
  \mathbb{E}[T(n)] = (n - 1) + \frac{1}{n} \sum_{i=1}^{n-1} T(i)
  \]
- which is expected $O(n)$
Median-of-Medians Select

- Similar to Quickselect but
  - pick pivot that is “always good”
  - i.e., between 25th and 75th percentile
- Do this by picking the median of medians (mom)
  - partition list into $n/5$ lists of size 5
  - sort each list in $O(1)$ time
  - choose the median of each list
  - call mom-select recursively on list of $n/5$ medians
- Use the mom as pivot and continue w/ selection algorithm
Median-of-Medians Select

\[
\text{momSelect}(\text{list, } k)
\]

if \text{list.size} \leq 5:
    \text{sort5(list)} \quad \text{// in } O(1) \text{ b/c list always size 5}
    \text{return kth element of list}

\text{miniLists} = \text{divide list into } n/5 \text{ lists of size 5}
\text{medians} = []
for \text{miniList} in \text{miniLists}:
    \text{sort5(miniList)}
    \text{medians.append(miniList[2])}
\text{pivot} = \text{momSelect(medians, medians.size/2)}

\text{L} = [] \quad \text{E} = [] \quad \text{G} = []
for \text{x} in \text{list}:
    if \text{x} < \text{pivot}: \text{L.append(x)}
    if \text{x} == \text{pivot}: \text{E.append(x)}
    if \text{x} > \text{pivot}: \text{R.append(x)}
if \text{k} \leq \text{L.size}:
    \text{return momSelect(L, } k\text{)}
else if \text{k} \leq (\text{L.size + E.size})
    \text{return pivot}
else
    \text{return momSelect(G, } k - (\text{L.size + E.size})\text{)}
Median-of-Medians Select

- Sorting a list of numbers from 1 to 50

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>12</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
</tr>
<tr>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
</tr>
</tbody>
</table>

- Guaranteed to be larger than red numbers and smaller than blue numbers which is between 25th and 75th percentiles. **Great pivot choice!**

- How many elements will pivot eliminate? What is area of blue or red region?
  - **height:** 3
  - **width:** \((n/5)/2\)
  - **area:** \(3n/10\)

leaves problem of size at most \(7n/10\)
Median-of-Medians Select

- With median-of-medians as pivot...
- ...the selection recurrence relation is:

\[ T(n) = T \left( \frac{n}{5} \right) + T \left( \frac{7n}{10} \right) + O(n) \]

- From recurring on list of \( n/5 \) medians to find median of median
- From recurring on list guaranteed to be at most \( 7/10 \) size of original

- This is \( O(n) \)
Summary

- We can perform selection in worst-case $O(n)$
  - ...which means we can find medians in worst-case $O(n)$
  - $\text{median}(L) = \text{mom\_select}(L, n/2)$

- Quicksort
  - If we choose pivot at random then expected $O(n \log n)$
  - If we choose pivot with $\text{mom\_select}$
    - then worst-case $O(n \log n)$
Outline

- Algorithm Analysis
  - Big-O, amortized, expected
- Selection & Medians
- Dynamic Programming
- Induction
Fibonacci

\[ F(n) = F(n - 1) + F(n - 2) \]

Base cases:

\[ F(0) = 1 \quad \& \quad F(1) = 1 \]

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
Fibonacci (Recursive)

- Defined by the recursive relation
  - \( F_0 = 0, \ F_1 = 1 \)
  - \( F_n = F_{n-1} + F_{n-2} \)
- We can implement this with a recursive function

```python
function fib(n):
    if n = 0:
        return 0
    if n = 1:
        return 1
    return fib(n-1) + fib(n-2)
```
Fibonacci (Recursive)

```python
function fib(n):
    if n = 0:
        return 0
    if n = 1:
        return 1
    return fib(n-1) + fib(n-2)
```

- Each node of tree is a recursive call of Fib( )
- Each level of the tree is a level of the recursion
Fibonacci (Recursive)

function fib(n):
    if n = 0:
        return 0
    if n = 1:
        return 1
    return fib(n-1) + fib(n-2)

- How many times does fib get called for fib(4)?
  - 8 times
- At each level it makes twice as many recursive calls as last
  - For fib(n) it makes approximately $2^n$ recursive calls
  - Algorithm is $O(2^n)$
Fibonacci (Recursive)

- How many times does \texttt{fib(1)} get computed?
- Instead of recomputing Fibonacci numbers over and over again
- Compute them \texttt{once} and store them for later

\begin{verbatim}
function fib(n):
    if n = 0:
        return 0
    if n = 1:
        return 1
    return fib(n-1) + fib(n-2)
\end{verbatim}
Fibonacci (Dynamic Programming)

```python
function dynamicFib(n):
    fibs = [] // make an array of size n
    fibs[0] = 0
    fibs[1] = 1

    for i from 2 to n:
        fibs[i] = fibs[i-1] + fibs[i-2]

    return fibs[n]
```
Outline

- Algorithm Analysis
  - Big-O, amortized, expected
- Selection & Medians
- Dynamic Programming
- Induction
Induction Example

\[ P(n) : \sum_{i=1}^{n} = \frac{n \cdot (n + 1)}{2}, \text{ for } n \geq 1 \]

- Prove base case: \( n=1 \)
  - \( \sum_{i=1}^{1} = 1 \) and \( \frac{1 \cdot (1 + 1)}{2} = 1 \) so \( P(1) : \sum_{i=1}^{1} = \frac{1 \cdot (1 + 1)}{2} \) is true

- Induction assumption: \( n=k \)
  - assume \( P(k) : \sum_{i=1}^{k} = \frac{k \cdot (k + 1)}{2} \) is true

- Induction step
  - prove that \( P(k+1) \) is true if \( P(k) \) is true
Induction Example

- Prove induction step

\[ P(k+1) : \sum_{i=1}^{k+1} i = 1 + 2 + \cdots + k + (k+1) \]

\[
= \sum_{i=1}^{k} i + (k + 1) \\
= \frac{k \cdot (k+1)}{2} + (k + 1) \\
= \frac{k \cdot (k+1)}{2} + \frac{2 \cdot (k+1)}{2} \\
= \frac{(k+1) \cdot (k+2)}{2}
\]

- Induction assumption

\[
\sum_{i=1}^{k} i = \frac{k \cdot (k+1)}{2}
\]

- Factor out \((k+1)\)