CSC236 Intro. to the Theory of Computation
Lecture 7: Master Theorem; more D&C; correctness

Amir H. Chinaei, Fall 2016
Office Hours: W 2–4 BA4222
ahchinaei@cs.toronto.edu
http://www.cs.toronto.edu/~ahchinaei/
Course page:
http://www.cdf.toronto.edu/~csc236h/fall/index.html
Section page:
http://www.cdf.toronto.edu/~csc236h/fall/amir_lectures.html

Recurrences

Review

Last week
• application of recurrence relations to complexity of d&c algorithms
  • in particular mergeSort

This week
• master theorem
• closestPair
• recursive correctness

Example 63: mergeSort
Keep in mind: $T(n) \leq T(\frac{n}{2}) \leq T(\widehat{n})$
and $T(\widehat{n}) = a \log \widehat{n} + b \widehat{n} - 1$

Calculating a lower bound

$T(n) \geq c \cdot n \log n$

$T(n) \geq T\left(\frac{n}{2}\right)\ 
\quad = \frac{a}{2} \cdot \log \frac{n}{2} + \frac{b}{2} \cdot \frac{n}{2} - 1
\quad = \frac{a}{2} (\log n - \log 2) + \frac{b}{2} - 1
\quad \geq \frac{a}{2} \log n + \frac{b}{2} - 1
\quad \geq \frac{a}{2} \log n
\quad c = \frac{1}{2}, n \geq 2
\quad \square$

Example 63: mergeSort
Keep in mind: $T\left(\frac{n}{2}\right) \leq T(n) \leq T(\widehat{n})$
and $T(\widehat{n}) = a \log \widehat{n} + b \widehat{n} - 1$

Calculating an upper bound

$T(n) \leq T(\widehat{n})$

$T(n) = a \log \hat{n} + 2\hat{n} - 1
\leq 2n \log 2n + 2.2n - 1
= 2n (\log 2 + \log n) + 6n - 1
\leq 2n \log n + 6n
\leq 8n \log n
\quad c = 8, n \geq 2
\quad \square$

general d&c and master theorem

D&C algorithms normally divide a problem of size $n$ to a smaller
problems of size $\frac{n}{a}$ where $a > 0, b > 1 \in \mathbb{N}$

Let $g(x)$ denote the recombining cost (conquer), such that the
corresponding recurrence relation is

$T(n) = \begin{cases} 
1 & n \leq B \in \mathbb{N} \\
\alpha T\left(\frac{n}{a}\right) + g(x) & n > B \in \mathbb{N}
\end{cases}$

When $g(x) = cn^d$, $c > 0 \ d \geq 0 \in \mathbb{R}$

$T(n) \in \begin{cases} 
\theta(n^d) & \text{if } a < b^d \\
\theta(n^d \log n) & \text{if } a = b^d \\
\theta(n^{\log a d}) & \text{if } a > b^d
\end{cases}$

Notes:

Recurrences and D&C 7-1

Recurrences and D&C 7-2

Recurrences and D&C 7-3

Recurrences and D&C 7-4

Recurrences and D&C 7-5

Recurrences and D&C 7-6
**Example 64:** closestPair

### Brute Force

```python
s = [set of points]
def closest_pair1(frm, to):
    min_d = d(s[0], s[1])
    for i in range(to - frm):
        for j in range(i + 1, to - frm + 1):
            dist = d(s[i], s[j])
            if dist < min_d:
                min_d = dist
    return min_d
```

### Divide & Conquer

```python
s = [set of points]
def closest_pairDC0(frm, to):
    if to - frm > 1:
        mid = (frm + to) // 2
        half1 = closest_pairDC0(frm, mid)
        half2 = closest_pairDC0(mid + 1, to)
        c = min(half1, half2)
        border_min = border_bf(mid, c)
        return min(c, border_min)
    else:
        return d(s[frm], s[to])
```

### Analysis:

In this example: 2.8
Example 64: closestPair

\[ d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \]

s = (set of points, sorted by x-coordinates)

```python
def closest_pairDC1(frm, to):
    if to - frm > 1:
        mid = (frm+to)//2
        half1 = closest_pairDC1(frm, mid)
        half2 = closest_pairDC1(mid+1, to)
        c = min(half1, half2)
        mergeSort(s, on y-coordinates)
        border_min = border_n(mid, c)
        return min(closest, border_min)
    else:
        return d(s[frm], s[to])
```

Example 64: closestPair

s = (set of points, sorted by x-coordinates)

```python
def closest_pairDC(frm, to):
    if to - frm > 1:
        mid = (frm+to)//2
        half1 = closest_pairDC(frm, mid)
        half2 = closest_pairDC(mid+1, to)
        c = min(half1, half2)
        merge(half1, half2, y-coordinates)
        border_min = border_n(mid, c)
        return min(closest, border_min)
    else:
        return d(s[frm], s[to])
```

programs correctness

- Recently, we saw the application of induction in asymptotic analysis
  * in particular, in the worst case time complexity of recursive algorithms
- Let’s move on to see application of induction in a more important topic: correctness of recursive algorithms
  * followed by correctness of iterative algorithms, next week.

```
def binSearch(x, A, b, e):
    if b == e:
        if x == A[b]:
            return b
        else:
            return -1
    else:
        m = (b + e) // 2  # midpoint
        if x == A[m]:
            return m
        else:
            return binSearch(x, A, m, e)
```

Example 73: correctness of binSearch (from Example 61)

```
def binSearch(x, A, b, e):
    if b == e:
        if x == A[b]:
            return b
        else:
            return -1
    else:
        m = (b + e) // 2  # midpoint
        if x == A[m]:
            return m
        else:
            return binSearch(x, A, m, e)
```
how to devise the correctness proof:

- define the pre- & post- conditions for the algorithm
  - preconditions:
    - conditions that the algorithm's input should satisfy
  - postconditions:
    - conditions that should be satisfied after the algorithm has run
- then, show:
  \[ \text{preconditions} \Rightarrow \text{postconditions} \]

Example 73: correctness of binSearch:

- binSearch(x, A, b, e):
  - preconditions:
    - elements of A (from b to e) are sorted non-decreasingly
    - elements of A and x are comparable
    - \( 0 \leq b \leq e \)
    - \( \text{Len}(A) = n = e - b + 1 \)
  - postconditions:
    - binSearch(x, A, b, e) terminates and returns \( p \) such that \( b \leq p \leq e \) and \( x = A[p] \) if such a \( p \) exists; otherwise returns -1.

proof by induction:

- We want to show: \( \text{preconditions} \Rightarrow \text{postconditions} \)
  \[ P(n): \text{if } 0 \leq b \leq e \text{ where } A[b..e] \text{ is non-decreasing and } \text{Len}(A) = n = e - b + 1, \text{ and } x \text{ is comparable to elements of } A, \text{ the call to } \text{binSearch}(x, A, b, e) \text{ terminates and returns } p \text{ such that } b \leq p \leq e \text{ and } x = A[p] \text{ if such a } p \text{ exists; otherwise returns } -1. \]
- Proof by complete induction.
  - Basis step:
  - Inductive step:

notes: