Recurrence Relation Analysis
Divide-and-Conquer
• A recurrence relation is an equation that recursively defines a sequence: each term of the sequence is defined as a function of the preceding term(s)

• For instance

\[ f(n) = \begin{cases} 
2 & n=1 \\
f(n-1)+n & n>1
\end{cases} \]
General form

\[ T(n) = \begin{cases} \ c & \text{if } n = n_0 \\ a \cdot T(f(n)) + g(n) & \text{otherwise} \end{cases} \]

- **Base of recursion**
- **Running time for base**
- **Number of times recursive call is made**
- **Size of problem solved by recursive call**
- **All other processing not counting recursive calls**
Example

```python
def bugs(n):
    if n <= 1:
        do_something()
    else:
        bugs(n-1)
        bugs(n-2)
        for i in range(n):
            do_something_else()
```

\[ T(n) = \begin{cases} 
    c_1 & \text{if } n \leq 1 \\
    T(n-1) + T(n-2) + nc_2 & \text{otherwise} 
\end{cases} \]
def yosemite(n):
    if n == 1:
        do_something()
    else:
        for i in range(1,n-1):
            yosemite(i)
            do_something_different()

$T(n) = \begin{cases} 
    c_1 & \text{if } n = 1 \\
    \sum_{i=1}^{n-1} (T(i) + c_2) & \text{otherwise}
\end{cases}$
MergeSort

• MergeSort is a divide & conquer algorithm
  – *Divide*: divide an $n$-element sequence into two subsequences of approx $n/2$ elements
  – *Conquer*: sort the subsequences recursively
  – *Combine*: merge the two sorted subsequences to produce the final sorted sequence
def mergesort(A):
    if len(A) < 2:
        return A
    else:
        m = len(A)/2
        l = mergesort(A[:m])
        r = mergesort(A[m:])
        return merge(l, r)
Example

Figure 4.2: Merge-sort tree $T$ for an execution of the merge-sort algorithm on a sequence with 8 elements: (a) input sequences processed at each node of $T$; (b) output sequences generated at each node of $T$. 
def merge(l, r):
    result, i, j = [], 0, 0
    while i < len(l) and j < len(r):
        if l[i] <= r[j]:
            result.append(l[i])
            i += 1
        else:
            result.append(r[j])
            j += 1
    result += l[i:]
    result += r[j:]
    return result
MergeSort Analysis

- Divide: Just computes the middle of the subsequence, thus takes constant time:
  \[ D(n) = \Theta (1) \]
- Conquer: We solve 2 subproblems of size approximately \( n/2 \):
  \[ a = 2, \quad b = 2 \]
- Combine: Merge takes \( \Theta (n) \):
  \[ C(n) = \Theta (n) \]
- Noting that \( \Theta (n) + \Theta (1) \) is still \( \Theta (n) \), we get:
  \[ T(n) = \Theta (1) \quad \text{if } n = 1 \]
  \[ 2 \ T(n/2) + \Theta (n) \quad \text{if } n > 1 \]
- Later we will see that:
  \[ T(n) = \Theta (n \ lg \ n) \]
“Visual” Analysis

Figure 4.4: A visual analysis of the running time of merge-sort. Each node of the merge-sort tree is labeled with the size of its subproblem.
Solving Recurrence Relation
Methods

Two methods for solving recurrences

– Iterative substitution method
– Master method

Will be on the quiz
Iterative substitution

• Assume $n$ large enough
• Substitute $T$ on the right-hand side of the recurrence relation
• Iterate the substitution until we see a pattern which can be converted into a general closed-form formula
MergeSort recurrence relation

\[ T(N) = 2T\left(\frac{N}{2}\right) + N \quad \text{for} \quad N \geq 2 \]

\[ T(1) = 1 \]
\[ T(N) = 2 \left( 2 T \left( \frac{N}{4} \right) + \frac{N}{2} \right) + N \]

\[ = 4 T \left( \frac{N}{4} \right) + 2N \]

\[ = 4 \left( 2 T \left( \frac{N}{8} \right) + \frac{N}{4} \right) + 2N \]

\[ = 8 T \left( \frac{N}{8} \right) + 3N \]

\[ \vdots \]

\[ = 2^i T \left( \frac{N}{2^i} \right) + iN \]

\[ T(1) = 1 \]

The expansion stops for \( i = \log_2 N \), so that

\[ T(N) = N + N \log_2 N \]
Verify the correctness

• How to verify the solution is correct?

• Use proof by induction!

• Important: make sure the constant \( c \) works for both the base case and the induction step
Proof by induction

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + n & \text{otherwise} 
\end{cases} \]

Fact: \( T(n) \in O(n \log_2 n) \)

Proof. Base case: \( T(2) = 2T(1) + 2 = 4 \leq c(2 \log_2 2) = 2c \).

Hence, \( c \geq 2 \).

Induction hypothesis: \( T(n/2) \leq c \frac{n}{2} \log_2 \frac{n}{2} \)

Induction: \( T(n) = 2T(n/2) + n \)

\[ \leq 2c \frac{n}{2} \log_2 \frac{n}{2} + n \]

\[ = cn \log_2 \frac{n}{2} + n = cn \log_2 n \quad cn \log_2 2 + n \]

\[ = cn \log_2 n + n(1 - c) \leq cn \log_2 n \text{ when } c \geq 1 \]

Choose \( c = 2 \).
Wrong proof by induction

\[
T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases}
\]

Fact (wrong): \( T(n) \in O(n) \)

Proof. Base case: \( T(1) = 1 \leq c, \text{ hence } c \leq 1 \)
Induction hypothesis: \( T(n/2) \leq c(n/2) \)
Induction: \( T(n) = 2T(n/2) + n \leq 2c(n/2) + n \)
\[= cn + n \in O(n)\]

proof is WRONG, but where is the mistake?
Towers of Hanoi
Towers of Hanoi

**Goal:** transfer all \( N \) disks from peg A to peg C

**Rules:**
- move one disk at a time
- never place larger disk above smaller one

**Recursive solution:**
- transfer \( N - 1 \) disks from A to B
- move largest disk from A to C
- transfer \( N - 1 \) disks from B to C

**Total number of moves:**
- \( T(N) = 2 \cdot T(N - 1) + 1 \)
def hanoi(n, a='A', b='B', c='C'):
    if n == 0:
        return
    hanoi(n-1, a, c, b)
    print a, '->', c
    hanoi(n-1, b, a, c)
Towers of Hanoi: Recurrence Relation

Solve

\[ T(N) = \begin{cases} 
2T(N - 1) + 1 & N > 1 \\
1 & N = 1 
\end{cases} \]
Towers of Hanoi: Unfolding the relation

\[ T(N) = 2 \left( 2 \ T(N - 2) + 1 \right) + 1 = \]
\[ = 4 \ T(N - 2) + 2 + 1 = \]
\[ = 4 \left( 2 \ T(N - 3) + 1 \right) + 2 + 1 = \]
\[ = 8 \ T(N - 3) + 4 + 2 + 1 = \]
\[ \ldots \]
\[ = 2^i \ T(N - i) + 2^{i-1} + 2^{i-2} + \ldots + 2^1 + 2^0 \]

the expansion stops when \( i = N - 1 \)

\[ T(N) = 2^{N-1} + 2^{N-2} + 2^{N-3} + \ldots + 2^1 + 2^0 \]

This is a geometric sum, so that we have:

\[ T(N) = 2^N - 1 \in \Theta\left(2^N\right) \]
Problem: Solve exactly (by iterative substitution)

\[ T(n) = \begin{cases} 
 4 & n = 1 \\
 4T(n - 1) + 3 & n > 1 
\end{cases} \]
Problem

Problem: Solve exactly (by iterative substitution)

\[
T(n) = \begin{cases} 
4 & n = 1 \\
4T(n - 1) + 3 & n > 1 
\end{cases}
\]

Solution: \( T(n) = 4^n + 4^{n-1} - 1 \)

Proof?
Another example

\[ T(N) = 2T(\sqrt{N}) + 1 \quad T(2) = 0 \]

\[
\begin{align*}
2T(N^{1/2}) &+ 1 \\
2(2T(N^{1/4}) + 1) &+ 1 \\
4T(N^{1/4}) &+ 1 + 2 \\
8T(N^{1/8}) &+ 1 + 2 + 4 \\
\cdots
\end{align*}
\]
Another example

\[ 2^i T \left( \frac{1}{N^{2^i}} \right) + 2^0 + 2^1 + \ldots + 2^i - 1 \]

The expansion stops for \( N^{2^i} = 2 \)
i.e., \( i = \log\log N \)

\[ T(N) = 2^0 + 2^1 + \ldots + 2^{\log\log N} - 1 = \log N \cdot 1 \]
Divide-and-Conquer
Master Theorem

Size $n$

Size $n/b$

Size $n/b^2$

Branching factor $a$

Depth $\log_b n$

Width $a^{\log_b n} = n^{\log_b a}$

Algorithms, S. Dasgupta
Let’s solve the following recurrence in general:

\[
T(n) = aT(n/b) + O(n^d)
\]

where \(a > 0, b > 1, T(1) = 1\)

Do repeated substitutions:

\[
T(n) = aT(n/b) + cn^d \\
= a[aT(n/b^2) + c(n/b)^d] + cn^d \\
= a^2T(n/b^2) + c(a/b^d)n^d + cn^d \\
\ldots
\]

\[
= a^jT(n/b^j) + cn^d \left[ (a/b^d)^{j-1} + \ldots + (a/b^d)^2 + (a/b^d) + 1 \right]
\]

\[
\ldots
\]

\[
= a^{\log_b n}T(1) + cn^d \cdot \sum_{i=0}^{\log_b n-1} (a/b^d)^i
\]

\[
= n^{\log_b a} + cn^d \cdot \sum_{i=0}^{\log_b n-1} (a/b^d)^i
\]
\[ n^{\log_b a} + cn^d \cdot \sum_{i=0}^{\log_b n-1} \left(\frac{a}{b^d}\right)^i \]

Case 1: \( a < b^d \) \((d > \log_b a)\). The second term is a geometric series with the ratio smaller than 1, so \( \sum_{i=0}^{\log_b n-1} (a/b^d)^i = O(1) \). The first term is \( n^{\log_b a} \) with \( \log_b a < \log_b b^d = d \), so we get \( T(n) = O(n^d) \).

Case 2: \( a = b^d \). The first term is \( n^d \). In the summation, we have \( \log_b n \) terms and they are all equal \( a/b^d = 1 \), so the second term is \( cn^d \log_b n \). Thus we get \( T(n) = O(n^d \log n) \).

Case 3: \( a > b^d \) \((d < \log_b a)\). Summing the geometric series in the second term, we get

\[
\sum_{i=0}^{\log_b n-1} (a/b^d)^i = \frac{(a/b^d)^{\log_b n-1}}{a/b^d - 1} = \frac{b^d}{a-b^d} \left(a^{\log_b n/b^d} - 1\right)
\]

So

\[
T(n) = n^{\log_b a} + c \frac{b^d}{a-b^d} \left(n^{\log_b a} - n^d\right) = O(n^{\log_b a}).
\]
Divide-and-Conquer
Master Theorem

Theorem (Master Theorem): If $T(n) = aT([n/b]) + O(n^d)$ for some constants $a > 0$, $b > 1$, and $d \geq 0$, then

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

$$T(n) = \begin{cases} 
O(n^d) & \text{if } a < b^d \\
O(n^d \log n) & \text{if } a = b^d \\
O(n^{\log_b a}) & \text{if } a > b^d 
\end{cases}$$
Master method: binary search

$T(n) = T(n/2) + O(1)$

$a = 1; \ b = 2; \ d = 0;$

$log_b a = log_2 1 = 0; \ log_b a = d;$

$T(n) = O(n^0 \log n) = O(\log n)$

• Search for key 13:
  15 → 6 → 7 → 13
(d) Algorithm PRINTUS \((n: integer)\)
   
   \[\begin{align*}
   &\text{if } n < 4 \\
   &\quad \text{print("U")} \\
   &\text{else} \\
   &\quad \text{PRINTUS}\left(\lfloor n/4 \rfloor\right) \\
   &\quad \text{PRINTUS}\left(\lfloor n/4 \rfloor\right) \\
   &\quad \text{for } i \leftarrow 1 \text{ to } 11 \text{ do print("U")}
   \end{align*}\]

(d) There are 2 recursive calls, each with parameter \(\lfloor n/4 \rfloor\). Since we are looking for an asymptotic solution, we can ignore rounding. Then the number of letters printed can be expressed by the recurrence:

\[X(n) = 2X(n/4) + 11.\]

We apply the Master Theorem with \(a = 2, b = 4, c = 11, d = 0\). Here, we have \(a > b^d\), so the solution is \(\Theta(n^{\log_4 2})\).
Divide-and-Conquer

Algorithm PRINTVS \((n : \text{integer})\)

\[
\begin{align*}
&\text{if } n < 3 \\
&\quad \text{print(“V”)} \\
&\text{else} \\
&\quad \text{for } j \leftarrow 1 \text{ to } 9 \text{ do PRINTVS}([n/3]) \\
&\quad \text{for } i \leftarrow 1 \text{ to } 2n^3 \text{ do print(“V”)}
\end{align*}
\]

There are 9 recursive calls, each with parameter \([n/3]\). Since we are looking for an asymptotic solution, we can ignore rounding. Then the number of letters printed can be expressed by the recurrence:

\[
X(n) = 9X(n/3) + 2n^3.
\]

We apply the Master Theorem with \(a = 9\), \(b = 3\), \(c = 2\), \(d = 3\). Here, we have \(a < b^d\), so the solution is \(\Theta(n^3)\).
Linear-time selection
Linear-time selection

• **Problem:** Select the $i$-th smallest element in an unsorted array of size $n$ (assume distinct elements)

• Trivial solution: sort $A$, select $A[i]$ time complexity is $O(n \log n)$

• Can we do it in linear time? Yes, thanks to Blum, Floyd, Pratt, Rivest, and Tarjan
Linear-time selection

Select (A, start, end, i) /* i is the i-th order statistic */

1. divide input array A into \([n/5]\) groups of size 5
   (and one leftover group if \(n \% 5\) is not 0)
2. find the median of each group of size 5 by sorting
   the groups of 5 and then picking the middle element
3. call Select recursively to find \(x\), the median of the \([n/5]\)
   medians
4. partition array around \(x\), splitting it into two arrays
   \(L\) (elements smaller than \(x\)) and \(R\) (elements bigger than \(x\))
5. \(k \Leftarrow |L| + 1\)
   if \((i = k)\) then return \(x\)
   else if \((i < k)\) then Select \((L, i)\)
   else Select \((R, i - k)\)

\([r]\) means the \textit{ceiling} (rounding to the next integer) of real number \(r\)
def selection(a, rank):
    n = len(a)
    if n <= 5:
        return rank_by_sorting(a, rank)
    medians = [rank_by_sorting(a[i:i+5], 3)
                for i in range(0, n-4, 5)]
    median = selection(medians, (len(medians) + 1) // 2)
    L, R = [], []
    for x in a:
        if x < median:
            L += [x]
        else:
            R += [x]
    if rank <= len(L):
        return selection(L, rank)
    else:
        return selection(R, rank - len(L))
Example

Let us run \textbf{Select}(A, 1, 28, 11), where

A=\{12, 34, 0, 3, 22, 4, 17, 32, 3, 28, 43, 82, 25, 27, 34, 2, 19, 12, 5, 18, 20, 33, 16, 33, 21, 30, 3, 47\}

Note that the elements in this example are not distinct.
Example

First make groups of 5

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>4</td>
<td>43</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>34</td>
<td>17</td>
<td>82</td>
<td>19</td>
<td>33</td>
</tr>
<tr>
<td>0</td>
<td>32</td>
<td>25</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>27</td>
<td>5</td>
<td>33</td>
</tr>
<tr>
<td>22</td>
<td>28</td>
<td>34</td>
<td>18</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>47</td>
</tr>
</tbody>
</table>
Example

Then find medians in each group

\[
\begin{array}{ccccccc}
0 & 4 & 25 & 2 & 20 & 3 \\
3 & 3 & 27 & 5 & 16 & 30 \\
12 & 17 & 34 & 12 & 21 & 47 \\
34 & 32 & 43 & 19 & 33 & 33 \\
22 & 28 & 82 & 18 & & \\
\end{array}
\]
Then find median of medians

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>25</th>
<th>2</th>
<th>20</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>27</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>34</td>
<td>12</td>
<td>21</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>32</td>
<td>43</td>
<td>19</td>
<td>33</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>28</td>
<td>82</td>
<td>18</td>
<td>33</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

12, 12, 17, 21, 30, 34
Example

Use 17 as the pivot value and partition original array

\[
\begin{array}{cccccccc}
0 & 4 & 25 & 2 & 20 & 3 \\
3 & 3 & 27 & 5 & 16 & 30 \\
12 & 17 & 34 & 12 & 21 & 47 \\
34 & 32 & 43 & 19 & 33 & \\
22 & 28 & 82 & 18 & 33 & \\
\end{array}
\]

12, 12, 17, 21, 30, 34
Example

After partitioning

$L = \{12, 0, 3, 4, 3, 2, 12, 5, 16, 3\}$

$L$ contains 10 elements smaller than 17

$\{17\}$  this is the 11-th smallest

$R = \{34, 22, 32, 28, 43, 82, 25, 27, 34, 19, 18, 20, 33, 33, 21, 30, 47\}$

$R$ contains 17 elements bigger than 17
Linear-time selection

- Finding the median of medians guarantees that $x$ causes a “good split”
- At least a constant fraction of the $n$ elements $\leq x$ and a constant fraction $> x$
- Analysis: we need to find the worst case for the size of $L$ and $R$
Observation: At least $1/2$ of the medians found in step 2 are greater than the median of medians $x$. So at least half of the $\lceil n/5 \rceil$ groups contribute 3 elements that are bigger than $x$, except for the one group with less than 5 elements and the group with $x$ itself.
Linear-time selection: analysis

• Therefore there are
  \[ 3([1/2 \left\lfloor n/5 \right\rfloor] - 2) \geq (3n/10) - 6 \]
  elements are \( > x \) (or \( < x \))

• So worst-case split has at most \( (7n/10) + 6 \) elements in
  “big” section of the problem, that is:
  \[ \max\{|L|, |R|\} < (7n/10) + 6 \]
Linear-time selection: analysis

Running Time:
1. $O(n)$ (break into groups of 5)
2. $O(n)$ (sorting 5 numbers and finding median is $O(1)$ time)
3. $T([n/5])$ (recursive call to find median of medians)
4. $O(n)$ (partition is linear time)
5. $T(7n/10 + 6)$ (maximum size of subproblem)

Recurrence relation

$$T(n) = T([n/5]) + T(7n/10 + 6) + O(n) \quad n > 80$$

$$= \Theta(1) \quad n \leq 80$$
Linear-time selection: analysis

Fact: \( T(n) = T(\lfloor n/5 \rfloor) + T(7n/10 + 6) + O(n) \) is \( O(n) \)

Proof:

Base case: easy (omitted).

\[
T(n) = T(\lfloor n/5 \rfloor) + T(7n/10 + 6) + O(n) \\
\leq c\lfloor n/5 \rfloor + c(7n/10 + 6) + O(n) \\
\leq c((n/5) + 1) + 7cn/10 + 6c + O(n) \\
= cn - [c(n/10 - 7) - dn] \\
\leq cn \quad \text{This step holds since } n \geq 80 \text{ implies } (n/10 - 7) \text{ is positive.}
\]

Choosing \( c \) big enough makes \( c(n/10 - 7) - dn \) positive, so last line holds.
Summary (1/3)

• **Goal:** analyze the worst-case time-complexity of iterative and recursive algorithms

• **Tools:**
  – Pseudo-code
  – Big-O, Big-Omega, Big-Theta notations
  – Recurrence relations
  – Discrete Math (summations, induction proofs, methods to solve recurrence relations)
Summary (2/3)

• Pure iterative algorithm:
  – Analyze the loops
  – Determine how many times the inner core is repeated as a function of the input size
  – Determine the worst-case for the input
  – Write the number of repetitions as a function of the input size
  – Simplify the function using big-O or big-Theta notation (optional)
Summary (3/3)

• Recursive + iterative algorithm:
  – Analyze the recursive calls and the loops
  – Determine how many recursive calls are made and the size of the arguments of the recursive calls
  – Determine how much extra processing (loops) is done
  – Determine the worst-case for the input
  – Derive a recurrence relation
  – Solve the recurrence relation
  – Simplify the solution using big-O, or big-Theta