CS 341: ALGORITHMS

Lecture 4: divide & conquer
Readings: see website

Trevor Brown (co-taught with Anna Lubiw)
https://www.student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca

DIVIDE-AND-CONQUER DESIGN STRATEGY

- **divide**: Given a problem instance $I$, construct one or more smaller problem instances $I_1, \ldots, I_a$
  - These are called **subproblems**
  - Usually, want subproblems to be small compared to the size of $I$ (e.g., half the size)
- **conquer**: For $1 \leq j \leq a$, solve instance $I_j$ recursively, obtaining solutions $S_1, \ldots, S_a$
- **combine**: Given solutions $S_1, \ldots, S_a$, use an appropriate combining function to find the solution $S$ to the problem instance $I$
  - i.e., $S = \text{Combine}(S_1, \ldots, S_a)$

D&C PROTO-ALGORITHM

```plaintext
if BaseCase(I) return Result(I)
subproblems = [I_1, I_2, \ldots, I_a]
subsol = []
for j = 1..a
    subsol[j] = DnC_template[I_j]
return Combine(subsol)
```

CORRECTNESS

- Prove base cases are correct
- Inductively assume subproblems are solved correctly
- Show they are correctly assembled into a solution

RUNTIME/SPACE COMPLEXITY?

- Techniques covered in this lecture
  - Model complexities using recurrence relations
  - Solve with substitution, master theorem, etc.
**WORKED EXAMPLE: DESIGN OF MERGESORT**

Here, a problem instance consists of an array $A$ of $n$ integers, which we want to sort in increasing order. The size of the problem instance is $n$.

- **divide:** Split $A$ into two subarrays $A_L$ consists of the first $[n/2]$ elements in $A$ and $A_R$ consists of the last $[n/2]$ elements in $A$.
- **conquer:** Run Mergesort on $A_L$ and $A_R$.
- **combine:** After $A_L$ and $A_R$ have been sorted, use a function Merge to merge $A_L$ and $A_R$ into a single sorted array. Recall that this can be done in time $\Theta(n)$ with a single pass through $A_L$ and $A_R$. We simply keep track of the "current" element of $A_L$ and $A_R$, always copying the smaller one into the sorted array.

**MERGE: CONQUER AND COMBINE**

**PSEUDOCODE FOR MERGESORT**

```plaintext
1. Mergesort(A[1..n])
   2. if n == 1 then return A
   3. nL = ceil(n/2)
   4. aL = A[1..nL]
   5. aR = A[(nL+1) .. n]
   6. sL = Mergesort(aL)
   7. sR = Mergesort(aR)
   8. return Merge(sL, sR)
```

**PSEUDOCODE FOR MERGE**

```plaintext
Merge(aL[1..nL], aR[1..nR])

// aOut - (nL+nR) = empty array
if nL == 0 then return aOut else if nR == 0 then return aOut

if aL[iL] < aR[iR]
   aOut[iOut++] = aL[iL++]
else
   aOut[iOut++] = aR[iR++]

while iL < nL and iR < nR
```

There are still elements left in both arrays
Right array is out of elements
Left array is out of elements
Analysis of Mergesort

Mergesort(A[1..n])
  if n == 1 then return A
  nL = cell(n/2)
  aL = A[1..nL]
  aR = A[(nL+1)..n]
  sL = Mergesort(aL)
  sR = Mergesort(aR)
  return Merge(sL, sR)

So, Mergesort(A) takes O(n) time, plus the time for its two recursive calls.

How can we analyze this recursive program structure?

Recurrence Relations

A crucial analysis tool for recursive algorithms

Suppose a1, a2, … is an infinite sequence of real numbers.
A recurrence relation is a formula that expresses a general term an in terms of one or more previous terms a1, a2, …, an−1.
A recurrence relation will also specify one or more initial values starting at a1.
Solving a recurrence relation means finding a formula for an that does not involve any previous terms a1, …, an−1.

There are many methods of solving recurrence relations. Two important methods are guess and check and the recursion tree method.

Mathematically Expressing the Complexity of Mergesort

Let T(n) denote the time to run Mergesort on an array of length n.
divide takes time Θ(n)
conquer takes time T(\left\lfloor \frac{n}{2} \right\rfloor) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)
combine takes time Θ(n)

Recurrence relation:

T(n) = \begin{cases} 
T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + Θ(n) & \text{if } n > 1 \\
Θ(1) & \text{if } n = 1.
\end{cases}

T(n) is a function of T(\left\lfloor \frac{n}{2} \right\rfloor) so T is a recurrence relation.

How can we compute/solve for T(n)?

To make this easier, assume n = 2^k, which lets us ignore floors/ceilings.

Recursion Tree Method

Evaluating recurrences with \(T(n/2)\) terms

Recursion tree method

If pants were pants, would it wear them like this?

Total = \(cn \times \text{levels}\)
So, mergesort has runtime \(\Theta(n \log n)\)

Can also compute using a table…

Recursion Tree Method

Evaluating recurrences with \(T(n/2)\) terms

Level | # of nodes | Runtime per node | Total runtime for level |
0     | 1          | \(cn\)          | \(cn\)                |
1     | 2          | \(c(n/2)\)      | 2\(c(n/2)\) = \(cn\) |
2     | 4          | \(c(n/4)\)      | 4\(c(n/4)\) = \(cn\) |
...   | ...        | ...             | ...                   |

\(\log n\) steps, \(c(\log n) = c\) \(\text{steps} \times c\) = \(cn\)
Sample recurrence for two recursive calls on problem size n/2

Where c and d are constants.

We can solve this recurrence relation when n is a power of two, by
constructing a recursion tree, as follows.

Step 1: Start with a one-node tree, say N, having the value T(n).
Step 2: Grow two children of N. These children, say N₁ and N₂,
have the value T(n/2), and the value of N is replaced by c.
Step 3: Repeat this process recursively, terminating when a node
reaches the value T(1) = d.
Step 4: Sum the values on each level of the tree, and then compute
the sum of all these terms; the result is T(n).

### GUESS-AND-CHECK METHOD

• Suppose we have the following recurrence:
  \[ T(0) = 4; \quad T(n) = T(n - 1) + 6n - 5 \]

• Guess the form of the solution \textbf{any} way you like

• My approach: the substitution method
  
  - Recursively substitute the formula into itself
  - Try to identify patterns to guess the final closed form

• Prove that the guess was correct

### SUBSTITUTE METHOD: WORKED EXAMPLE

Recurrence: \[ T(0) = 4; \quad T(n) = T(n - 1) + 6n - 5 \]

\[ T(n) = T(n - 2) + 6(n - 1) - 5 + 6n - 5 \quad \text{(substitute)} \]

\[ T(n - 2) + 6n - 6 - 5 + 6n - 5 \]

Compare: new terms\( (6n-5) - 6 \)

\[ = T(n - 2) + 2(6n - 5) - 6 \quad \text{(substitute)} \]

\[ = (6(n - 3) + 6(n - 2) - 5) + 2(6n - 5) - 6 \]

\[ = T(n - 3) + 6n - 2(6) + 5 + 2(6n - 5) - 6 \]

\[ = T(n - 3) + 3(6n - 5) - 6(1 + 2) \]

... Identify patterns and guess what happens in the limit

\[ = T(0) + n(6n - 5) - 6(1 + 2 + 3 + \cdots + (n - 1)) = \text{guess}(n) \]

• \( \text{guess}(n) = T(0) + n(6n - 5) - 6(1 + 2 + 3 + \cdots + (n - 1)) \)

• \[ = 4 + 6n^2 - 5n - 6n(n - 1)/2 \quad \text{(simplify)} \]

• \[ = 3n^2 - 2n + 4 \]

• Are we done?

• The form of \( \text{guess}(n) \) was an educated guess.

• To be formal, we must prove it correct using induction

### ANOTHER APPROACH

• Suppose you look for a while at the previous recurrence:
  
  \[ T(0) = 4; \quad T(n) = T(n - 1) + 6n - 5 \]

• With some experience, you might just guess it’s quadratic

• If you’re right, it should have the form:
  
  \[ an^2 + bn + c \]

• For some unknown constants \( a, b, c \)

• So, just carry the unknown constants into the proof!

• You can then determine what the constants must be
  for the proof to work out

\[ \text{PROOF} \]

• \( \text{guess}(n) = 3n^2 - 2n + 4 \)

• Want to prove: \( \text{guess}(n) = T(n) \) for all \( n \)

• Base case: \( \text{guess}(0) = 3(0)^2 - 2(0) + 4 = T(0) \)

• Inductive case: suppose \( \text{guess}(n) = T(n) \) for \( n \geq 0 \).

• \( \text{show} \) \( \text{guess}(n + 1) = T(n + 1) \).

• \[ T(n + 1) = T(n) + 6(n + 1) - 5 \quad \text{(by definition)} \]

• \[ = \text{guess}(n) + 6(n + 1) - 5 \quad \text{(by inductive hypothesis)} \]

• \[ = 3n^2 + 4n + 5 \quad \text{(substitute & simplify)} \]

• \[ \text{guess}(n + 1) = 3(n + 1)^2 - 2(n + 1) + 4 \quad \text{(by definition)} \]

• \[ = 3n^2 + 4n + 5 = T(n + 1) \quad \text{(simplify)} \]
• \( T(0) = 4 ; T(n) = T(n - 1) + 6n - 5 \); \( \text{guess}(n) = an^2 + bn + c \)
• Want to prove: \( \text{guess}(n) = T(n) \) for all \( n \)
• Base case: \( \text{guess}(0) = a(0)^2 + b(0) + c = T(0) = 4 \)
• This holds iff \( c = 4 \) (\( a, b \) are not constrained)
• Inductive case: suppose \( \text{guess}(n) = T(n) \) for \( n \geq 0 \).
  show \( \text{guess}(n + 1) = T(n + 1) \).
• \( T(n + 1) = T(n) + 6(n + 1) - 5 \) (by definition)
• \( = \text{guess}(n) + 6(n + 1) - 5 \) (by inductive hypothesis)
• \( = an^2 + bn + 4 + 6(n + 1) - 5 \) (substitute)
• \( = an^2 + (b + 6)n + 5 \) (simplify)

\[ \text{Recall: } \text{guess}(n) = an^2 + bn + c \text{ where } c = 4 \]
• Inductive case: suppose \( \text{guess}(n) = T(n) \) for \( n \geq 0 \).
  show \( \text{guess}(n + 1) = T(n + 1) \).
• \( T(n + 1) = an^2 + (b + 6)n + 5 \) (continue previous slide)
• \( \text{guess}(n + 1) = an(n + 1)^2 + b(n + 1) + 4 \) (by definition)
• \( = an^2 + 2n + 1 + bn + b + 4 \) (simplify, and...)
• \( = an^2 + (a + b + 4) \) (rearrange polynomial)
• We want this to be equal to \( T(n + 1) \)
• \( an^2 + (2a + b)n + (a + b + 4) = an^2 + (b + 6)n + 5 \)
  equivalent to \( 2a + b = (b + 6) \) and \( a + b + 4 = 5 \)
• first implies \( a = 3 \) plug into second to get \( b = 5 - 4 - 3 = -2 \)

\[ \text{So, inductive } \text{hypothesis is correct} \]

\[ \text{for } a = 3, b = -2, c = 4 \]

\[ \text{Example corresponding algorithm} \]

\[ \text{Let's rearrange this into a geometric sequence} \]

\[ \text{and solve} \]

\[ \text{SUM OVER ALL LEVELS WE GET} \]

\[ T(n) = da^l + \Sigma_{i=0}^{l-1} c^i \left( \frac{n}{b} \right)^{i} \]

\[ \text{LET'S rearrange this into a geometric sequence} \]

\[ \text{and solve} \]

\[ T(n) = da^l + cn^{\left\lceil \log_{b}{\frac{n}{d}} \right\rceil} \]

\[ \text{where } r = b^{k} \]

\[ \text{Recall formula: } \Sigma_{i=0}^{m-1} ar^{i} = \frac{a(r^{m-1} - 1)}{r - 1} \text{ if } r > 1 \]

\[ \text{if } r = 1 \]

\[ \text{if } 0 < r < 1 \]

\[ \text{So different solutions depending on } r \]

\[ \text{Case 1: } r = b^{k} > 1 \Leftrightarrow x - y > 0 \Leftrightarrow x > y \]

\[ \text{Case 2: } r = b^{k} = 1 \Leftrightarrow x - y = 0 \Leftrightarrow x = y \]

\[ \text{Case 3: } 0 < r = b^{k} < 1 \Leftrightarrow x - y < 0 \Leftrightarrow x < y \]
SOLVING THE GEOMETRIC SEQ

- Formula: $\sum_{i=0}^{n-1} ar^i = \begin{cases} \frac{r^{n+1} - 1}{r - 1} \in \Theta(r^n) & \text{if } r > 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a \frac{1 - r^n}{1 - r} \in \Theta(1) & \text{if } 0 < r < 1 \end{cases}$

- Case 1: $r = b^{-y} > 1 \iff x = y > 0 \iff x = y$
  - $T(n) = dn^x + cn^x \sum_{i=0}^{n-1} r^i \in dn^x + cn^x \Theta(n^x)$
  - $T(n) \in \Theta(n^x + n^x r^n) \in \Theta(n^x + n^x (b^{-y})^n) = \Theta(n^x + n^x (b^{-y})^n)$
  - Recall $b^y = n$, so $T(n) \in \Theta(n^x + n^x n^y) \in \Theta(n^x + n^x (b^{-y})^n)$
  - So $T(n) \in \Theta(n^x)$

- Case 2: $r = b^{-y} = 1 \iff x - y = 0 \iff x = y$
  - $T(n) = dn^x + cn^x \sum_{i=0}^{n-1} r^i \in dn^x + cn^x \Theta(n)$
  - $T(n) \in \Theta(n^x + n^x)$ since $x = y$
  - Recall $b^y = n$, so $\log_b n = \log_b n$. This means $j \in \Theta(\log n)$.
  - So $T(n) = \Theta(n^x + n^x \log n) = \Theta(n^x \log n)$

- Case 3: $0 < r = b^{x-y} < 1 \iff x - y < 0 \iff x < y$
  - $T(n) = dn^x + cn^x \sum_{i=0}^{n-1} r^i \in dn^x + cn^x \Theta(1)$
  - $T(n) \in \Theta(n^x + n^x)$
  - Since $x < y$, we simply have $T(n) \in \Theta(n^y)$

Note that the base case constant $a$ is not present in any of these complexities!

SOLVING THE GEOMETRIC SEQ

- Formula: $\sum_{i=0}^{n-1} ar^i = \begin{cases} \frac{r^{n+1} - 1}{r - 1} \in \Theta(r^n) & \text{if } r > 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a \frac{1 - r^n}{1 - r} \in \Theta(1) & \text{if } 0 < r < 1 \end{cases}$

- Case 1: $r = b^{-y} > 1 \iff x = y > 0 \iff x = y$
  - $T(n) = dn^x + cn^x \sum_{i=0}^{n-1} r^i \in dn^x + cn^x \Theta(n^x)$
  - $T(n) \in \Theta(n^x + n^x r^n) \in \Theta(n^x + n^x (b^{-y})^n)$
  - Recall $b^y = n$, so $T(n) \in \Theta(n^x + n^x n^y) \in \Theta(n^x + n^x (b^{-y})^n)$
  - So $T(n) \in \Theta(n^x)$

- Case 2: $r = b^{-y} = 1 \iff x - y = 0 \iff x = y$
  - $T(n) = dn^x + cn^x \sum_{i=0}^{n-1} r^i \in dn^x + cn^x \Theta(n)$
  - $T(n) \in \Theta(n^x + n^x)$ since $x = y$
  - Recall $b^y = n$, so $\log_b n = \log_b n$. This means $j \in \Theta(\log n)$.
  - So $T(n) = \Theta(n^x + n^x \log n) = \Theta(n^x \log n)$

- Case 3: $0 < r = b^{x-y} < 1 \iff x - y < 0 \iff x < y$
  - $T(n) = dn^x + cn^x \sum_{i=0}^{n-1} r^i \in dn^x + cn^x \Theta(1)$
  - $T(n) \in \Theta(n^x + n^x)$
  - Since $x < y$, we simply have $T(n) \in \Theta(n^y)$

SOME BONUS INTUITION FOR R CASES

Recall: $T(n) = dn^x + cn^x \sum_{i=0}^{n-1} r^i$ where $r = b^{-y}$
$x = \log_b n$ i.e. $\log_{\text{subproblem size}} \text{subproblems}$

<table>
<thead>
<tr>
<th>Case</th>
<th>$r$, $y$, $x$</th>
<th>Complexity of $T(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>heavy leaves</td>
<td>$r &gt; 1$, $y &lt; x$</td>
<td>$T(n) \in \Theta(n^y)$</td>
</tr>
<tr>
<td>balanced</td>
<td>$r = 1$, $y = x$</td>
<td>$T(n) \in \Theta(n^x \log n)$</td>
</tr>
<tr>
<td>heavy top</td>
<td>$r &lt; 1$, $y &gt; x$</td>
<td>$T(n) \in \Theta(n^x)$</td>
</tr>
</tbody>
</table>

- heavy leaves means that the value of the recursion tree is dominated by the values of the leaf nodes.
- balanced means that the values of the levels of the recursion tree are constant (except for the last level).
- heavy top means that the value of the recursion tree is dominated by the value of the root node.

MASTER THEOREM FOR RECURRENCES

- Simplified version

Consider recurrence:
$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^y)$ where $a \geq 1, b \geq 2$ and $n = b^i$
And let $x = \log_b n$.

$T(n) \in \begin{cases} \Theta(n^y) & \text{if } y < x \\ \Theta(n^x \log n) & \text{if } y = x \\ \Theta(n^x) & \text{if } y > x. \end{cases}$

WORKED EXAMPLES

Recall: simplified master theorem

Suppose that $a \geq 1$ and $b > 1$. Consider the recurrence
$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^y)$, where $n$ is a power of $b$.
Denote $x = \log_b n$. Then
$T(n) \in \begin{cases} \Theta(n^y) & \text{if } y < x \\ \Theta(n^x \log n) & \text{if } y = x \\ \Theta(n^x) & \text{if } y > x. \end{cases}$

Questions: $a=?$ $b=?$ $y=?$ $x=?$ which $\Theta$ function?

$T(n) = 2T(n/2) + cn$,
$a=2$, $b=2$, $y=1$, $x=1$
$\Theta(\text{subproblem size}) = \Theta(\text{subproblem size})$

$T(n) = 3T(n/2) + cn$,
$a=3$, $b=2$, $y=\log_2 3$, $x=\log_2 3$
$\Theta(\text{subproblem size}) = \Theta(\text{subproblem size})$

$T(n) = 4T(n/2) + cn$, $y=2$,
$a=4$, $b=2$, $y=\log_2 4$, $x=\log_2 4$
$\Theta(\text{subproblem size}) = \Theta(\text{subproblem size})$

$T(n) = 2T(n/2) + cn^2$
$a=2$, $b=2$, $y=3/2$, $x=1$
$\Theta(\text{subproblem size}) = \Theta(\text{subproblem size})$
REVISITING THE RECURSION TREE METHOD

• Recall: \( n = 2^j \); \( T(1) = 1 \); \( T(n) = 2T\left(\frac{n}{2}\right) + n \log n \)

Summing the values at all levels of the recursion tree, we have

\[
T(n) = 2^j \left(1 + \sum_{i=1}^{j} \frac{n}{2^i}\right) - 2^j \left(1 + \frac{j(j+1)}{2}\right).
\]

Since \( n = 2^j \), we have \( j = \log_2 n \) and \( T(n) \in \Theta(n \log^2 n) \).

MASTER THEOREM WHEN \( b^{j-1} < n < b^j \)

• if \( \frac{n}{b} \) is not always an integer!
  • floors/ceilings are 0
  • not a geometric sequence

Suppose we get a \( \text{big-O} \) bound for \( b^{j-1} < n < b^j \) by instead considering the larger problem size \( b^j \)

\[
T(n) = T\left(b^j\right) + \Theta\left(b^j \log b^j\right)
\] if \( y < x \)

\[
T(n) = T\left(b^j\right) + \Theta\left(b^j \log b^j\right)
\] if \( y = x \)

\[
T(n) = T\left(b^j\right) + \Theta\left(b^j \log b^j\right)
\] if \( y > x \)

Case 1 \((y < x)\):

\( (bn)^x = b^n x^y \) and \( b^n x^y \) is a constant

So \( T(n) \in \Theta(n^x) \)

Case 2 \((y = x)\):

\( (bn)^x \log bn = b^n x^y \log (b + \log n) \)

\( T(bn) \in \Theta(b^n x^{y+1} \log n) \)

So \( T(n) \in \Theta(n^x \log n) \)

Case 3 \((y > x)\):

\( (bn)^x = b^n x^y \)

So \( T(n) \in \Theta(n^x) \)

Can tackle \( \Omega \) similarly to get \( \Omega \)