Unit 4: Solving Recurrence Relations

Agenda:

- Expressing runtime / operation cost of a recursive code using a recursive relation (CLRS p.35-37, 65-67)
- Solving recurrence relations:
  - Finding the solution: Iterated Substitution (CLRS p.83-87)
  - Proving the solution: Induction
  - Alternative Techniques: Recurrence Tree & Guess and Test
  - Master Theorem (CLRS p.93-97)

We have already seen several recursive codes

- procedure InsertionSort(A, n)

```
if (n > 1) then
    InsertionSort(A, n − 1)
    x ← A[n]
    PutInPlace(A, n − 1, x)
```

procedure PutInPlace(A, j, x)

```
if (j = 0) then
    A[1] ← x
else if (x > A[j]) then
    A[j + 1] ← x
else          ** i.e., x ≤ A[j]
    A[j + 1] ← A[j]
    PutInPlace(A, j − 1, x)
```

- procedure fib1(n)

```
if (n < 2) then
    return n
else
    return fib1(n − 1) + fib1(n − 2)
```

We have already seen several recursive codes

- Another sorting algorithm (we will discuss it later lengthly)

  <u>procedure Merge-Sort$(A; lo, hi)$</u>
  if $(lo < hi)$ then
      $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$
      Merge-Sort$(A; lo, mid)$
      Merge-Sort$(A; mid + 1, hi)$
      Merge$(A; lo, mid, hi)$
      $**$ Merge is a function that takes an array with $A[lo, mid]$ and
      $**$ $A[mid + 1, hi]$ sorted and makes $A[lo, hi]$ sorted
      $**$ Merge runs in $O(n)$ time and makes at most $n - 1$ Key Comparisons (KC)

- And here's some example merely for the sake of an example:

  <u>procedure QZ(n)</u>
  if $(n > 1)$ then
      $a \leftarrow n \times n + 37$
      $b \leftarrow a \times \texttt{QZ}(\frac{n}{2})$
      return $\texttt{QZ}(\frac{n}{2}) \times \texttt{QZ}(\frac{n}{2}) + n$
  else
      return $n \times n$

- How can we analyze the runtime of such recursive codes?

Recurrence relations — $\texttt{PutInPlace}(A, n, x)$

- $\underline{\texttt{procedure PutInPlace}(A, j, x)}$

```
if (j = 0) then
    A[1] ← x
else if (x > A[j]) then
    A[j + 1] ← x
else            ** i.e., x ≤ A[j]
    A[j + 1] ← A[j]
    PutInPlace(A, j − 1, x)
```

- The first step is to express the runtime of `PutInPlace` based on the code.
- Let $T(n)$ denote the worst-case #KC `PutInPlace` makes on input size n.
  - Why are we looking at #KC and not runtime?
  - If we are dealing with complicated elements, KC is the runtime bottle neck (other operations touch only indices and pointers)
  - Simpler for our analysis (it's a concrete count and we don't have to introduce some new constants)
  - If we make a good choice of operations we consider "costly" then the runtime is proportional to the number of such operations
- So what is $T(n)$?
  - `if` (line 1) makes no KC
  - `else-if` (line 3) makes one KC
  - Option 1: no more KC (as function halts)
  - Option 2: we make additional KC due to the *recursive call*
- Hence, $T(n) = 0 + 1 + \max\{0, T(n-1)\} = 1 + T(n-1)$
- What are we missing?
- Base case. $T(0) = 0$ (no KC)

Recurrence relations — `PutInPlace`(A, n, x)

- Let $T(n)$ denote the worst-case #KC `PutInPlace` makes on input size n.
- So $T(n) = \begin{cases} 0 & , \text{ if } n = 0 \\ 1 + T(n-1) & , \text{ o/w} \end{cases}$
- Such a form of the function $T(n)$ is called a recurrence relation: Expressing the value of $T(n)$ a function of the values $\{T(0), T(1), ..., T(n-1)\}$
  - Why can't $T(n)$ (or $T(n+1)$ for that matter) appear on the RHS of a recurrence relation?
- Our goal: convert $T(n)$ into a closed-form solution — in the sense of big-O notation
  - I.e., if we can express $T(n)$ as an exact form (e.g., $T(n) = 72n^2\log^5(n-4) + 14n\log^4(n) - 28n$) that's great.
  - But it is fine to derive the conclusion that $T(n) \le c \cdot n^2 \log^5(n)$ for some c and all sufficiently large ns, hence $T(n) = O(n^2 \log^5(n))$.
- Finding closed-form solution of $T(n)$ requires two steps:

  (1) finding the solution

  (2) proving the solution

WARNING: Remember what is truly important!

- The focus of this unit is indeed on solving recurrence relations.
- But remember: it's *just a technical tool* to get a closed-form runtime of an algorithm.
- I.e., it's a calculation. Nothing more.
  - It's not a simple calculation.
  - It takes effort to master this calculation, and you should master this calculation.
  - So we have a whole unit on doing this calculation.
- But the **most important** part is to **infer the right recurrence relation** that represents that runtime of the code!
- ...but that is also the part which is impossible to teach — it just boils down to understanding what the code does.

WARNING: Remember what is truly important!

- **Most important: infer the right recurrence relation from the code**!
- An exercise:

```
procedure QZ(n)
if (n > 1) then
    a ← n × n + 37
    b ← a × QZ(n/2)
    return QZ(n/2) × QZ(n/2) + n
else
    return n × n
```

- Denote $T(n)$ as the #arithmetic-operations done by QZ(n).
  - In the base case $(n \leq 1)$
    - we do one multiplication.
  - In the general case we do
    - 3 recursive calls, all on the same size of input $(\frac{n}{2})$
    - and 5 arithmetic operations (2 in the assignment of a, 1 in the assignment of b, 2 in the return call).
- Thus: $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 3T(\frac{n}{2}) + 5 & \text{if } n \geq 2 \end{cases}$

WARNING: Remember what is truly important!

- **Most important: infer the right recurrence relation from the code!**
- An exercise:

```
prod(A; p, q)      ** returns the product of all elements in A[p...q]
if (q > p) then
   mid ← ⌊(p+q)/2⌋
   return prod(A; p, mid) × prod(A; mid + 1, q)
else
   return A[p]
```

- Denote $T(n)$ as #multiplications that `prod` does on an array of size n.
  - In the base case $(n = 1)$
    - we do zero multiplications.
  - In the general case — when n is even — we do
    - 2 recursive calls, both on the same size of input ($\frac{n}{2}$)
    - and 1 multiplication (in the `return` call).
- Thus: $T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(\frac{n}{2}) + 1 & \text{if } n \geq 2 \end{cases}$
- Note: since our goal is to have a sense of the runtime as $n \to \infty$, it is ok to make assumptions such as n is divisible by 2 (or 3, 4, 10 or 117...) (More about this later in this unit.)

WARNING: Remember what is truly important!

- **Most important: infer the right recurrence relation from the code**!
- An exercise:

```
foo(A; p, q)
if (q > p + 1) then
    r1 ← ⌊(p+q)/3⌋
    r2 ← q − r1
    return   (foo(A; p, r2) + foo(A, r1, q))×
               (foo(A; p, r1) + foo(A; r1 + 1, r2) + foo(A; r2 + 1, q))
else if (q = p + 1) then
    return A[p] × A[q]
else
    return A[p]
```

- Denote $T(n)$ as #arithmetic-operations that `foo` does on elements of A when A's size is n.
- HW: argue that

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ 2T(\frac{2n}{3}) + 3T(\frac{n}{3}) + 4 & \text{if } n \geq 3 \text{ and divisible by } 3 \end{cases}.$$

Recurrence Relations

- *Recurrence relation*: A relation defined recursively — in terms of itself.
- Must have *base case* and *general case*.
- Examples:

$$f(n) = \begin{cases} 1, & \text{if } n = 1 \\ n + f(n-1), & \text{if } n \geq 2 \end{cases}$$

$$f(n) = \begin{cases} 1, & \text{if } n \leq 10 \\ 2^n + 5f(\lfloor \frac{n}{5} \rfloor) + \frac{n}{10} f(3), & \text{if } n > 10 \end{cases}$$

$$f(n) = \begin{cases} 34 & , \text{ if } n \leq 5 \\ f(n-7) + 12f(\lceil \sqrt{n} \rceil) + 6n^2 & , \text{ if } n > 5 \end{cases}$$

- How are recurrence-relations (in this course) derived?
- Arise in the analysis of recursive algorithms
- Therefore, it is safe to assume
  - $T(n) \geq 0$ for any n, or even ≥ 1 — code cannot consume negative resources
  - $T(1), T(2), ..,$ up to some constant – are all at most some constant (Unless the code does something really weird, on any input of size atmost, say, 5 it takes only some $f(5)$-runtime.)
  - $T(n)$ is monotonically increasing (not always true, but quite often is)
  - $T(n) = g(n) + \sum\limits_{0 \leq i < n} a_i T(i)$ where a_i is the number of recursive calls on an instance of size i (a natural).

1- Iterated substitution

- An easy example: consider the following recurrence

$$T(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1 + T(n-1), & \text{if } n \geq 1 \end{cases}$$

- Particular cases:

n	1	2	3	4	5	6	7
$T(n)$	1 $= 1$	$1+1$ $= 2$	$1+2$ $= 3$	$1+3$ $= 4$	$1+4$ $= 5$	$1+5$ $= 6$	$1+6$ $= 7$

Sometimes you can see the solution from this bottom-up approach.

- Often however, it is best to do top-down and plug-in the formula of T repeatedly (always applied to the largest term)
- General case:

$$\begin{aligned} T(n) &= 1 + T(n-1) \\ &= 1 + 1 + T(n-2) \\ &= 1 + 1 + 1 + T(n-3) \\ &= \ldots \\ &\overset{\text{the } i\text{th row}}{=} \underbrace{1 + 1 + \ldots + 1}_{i} + T(n-i) \\ \ldots &\overset{\text{stops}}{=} \underbrace{1 + 1 + \ldots + 1}_{n} + T(0) = n \end{aligned}$$

- This is merely our *guess* as to $T(n)$'s closed form — still need to prove it!

1- Iterated substitution

- Another example: $T(n) = \begin{cases} 0, & \text{if } n \leq 2 \\ 13n^2 + T(n-3), & \text{if } n \geq 3 \end{cases}$
- In iterated substitution, it is ok to assume n is of particular form. Here, we assume is it divisible by 3.

$$\begin{array}{lcl} T(n) & = & 13n^2 + T(n-3) \\ & = & 13n^2 + 13(n-3)^2 + T(n-6) \\ & = & 13\left(n^2 + (n-3)^2 + (n-6)^2\right) + T(n-9) \\ & \vdots & \\ & \overset{\text{the } i\text{th row}}{=} & 13\left(n^2 + (n-3)^2 + ... + (n-3(i-1))^2\right) + T(n-3i) \\ & \vdots & \\ & \overset{\text{stops}}{=} & 13\left(n^2 + (n-3)^2 + (n-6)^2 + ... + 3^2\right) + T(0) \\ & = & 13\sum_{i=0}^{n/3}(n-3i)^2 \end{array}$$

- We are not done yet. We aim for a closed form!

1- Iterated substitution

- Another example: $T(n) = \begin{cases} 0, & \text{if } n \leq 2 \\ 13n^2 + T(n-3), & \text{if } n \geq 3 \end{cases}$
- Our guess:

$$T(n) \quad = \quad 13 \sum_{i=0}^{n/3} (n-3i)^2$$

- We are not done yet. We aim for a closed form!
- Option 1:

  (i) find formula for $\sum_i (n-3i)^2$, use it to get a close-form guess.

  (ii) prove that $T(n)$ =closed-form by induction.
- Option 2:

  (i) leave the guess in a summation form;

  (ii) prove that $T(n)$ =summation via induction;

  (iii) reason that $13\sum_{i=0}^{n/3}(n-3i)^2 \in \Theta(n^3)$

  - All summands $\leq n^2$ so sum $\leq 13 \cdot \frac{n}{3} \cdot n^2 = \frac{13}{3}n^3$.
  - Largest $\frac{n}{6}$ summands $\geq (n/2)^2$ so sum $\geq 13 \cdot \frac{n}{6} \cdot \frac{n^2}{4} = \frac{13}{24}n^3$.
- Option 3:

  (i) do the above reasoning as part of the guess;

  (ii) prove via induction that $\frac{13}{24}n^3 \leq T(n) \leq \frac{13}{3}n^3$, which immediately implies $T(n) \in \Theta(n^3)$.
- Note: in all cases, in the induction proof **you must use explicit constants!**

2- Proving the Guess via Induction:

- First example: $T(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1 + T(n-1), & \text{if } n \geq 1 \end{cases}$
- We guessed $T(n) = n$.
- As ever, anything that involves recursion is proved via induction.
- Base case: $T(0) = 0$, by definition.
- Induction step: Fix n. Assuming $T(n-1) = n - 1$, we show $T(n) = n$
  $T(n) = 1 + T(n-1) \stackrel{\text{IH}}{=} 1 + n - 1 = n \quad \square$.

- Remember: you MUST prove your guess, otherwise, it is a mere guess.
- Remember: when the recursive relation involves $T(n/2)$ or multiple $T(i)$s — prove it using full/complete induction
- Remember: the result must be in the simplest closed-form you can (no sums, no recursions).

2- Proving the Guess via Induction:

- Another example: $T(n) = \begin{cases} 0, & \text{if } n \leq 2 \\ 13n^2 + T(n-3), & \text{if } n \geq 3 \end{cases}$
- (Option 2:) Our *guess* is $T(n) = \sum_{i=0}^{n/3}(n-3i)^2$.
- Claim: For every $n \geq 3$ divisible by 3 we have $T(n) = \sum_{i=0}^{n/3}(n-3i)^2$.
- Proof: Base case: $T(3) = 13 \times 9 = 13((3-0)^2 + (3-3)^2)$.
  Induction step: Fix n. Assuming the required holds for $T(n)$, we show it also holds for $T(n+3)$.

$$\begin{aligned} T(n+3) = 13(n+3)^2 + T(n) &\overset{\text{IH}}{=} 13(n+3)^2 + \sum_{i=0}^{n/3}(n-3i)^2 \\ &= 13(n+3)^2 + \sum_{i=0}^{n/3}(n+3-3(i+1))^2 \\ &= 13(n+3)^2 + \sum_{i=1}^{\frac{n}{3}+1}(n+3-3i)^2 \\ &= 13(n+3-0)^2 + \sum_{i=1}^{\frac{n+3}{3}}(n+3-3i-3)^2 = \sum_{i=0}^{\frac{n+3}{3}}(n+3-3i)^2 \quad \square \end{aligned}$$

- Remember that we now need to show that $\sum_{i=0}^{n/3}(n-3i)^2 \in \Theta(n^3)$.

2- Proving the Guess via Induction:

- Another example: $T(n) = \begin{cases} 0, & \text{if } n \leq 2 \\ 13n^2 + T(n-3), & \text{if } n \geq 3 \end{cases}$
- (Option 3:) Our *guess* is $\frac{13}{24}n^3 \leq T(n) \leq \frac{13}{3}n^3$
- It is completely fine to also try and guess some constants. So, for example, let's pick 0.1 and 10 as our constants.
- Claim: For every $n \geq 3$ we have $\frac{1}{10}n^3 \leq T(n) \leq 10n^3$.
- Proof: Base case: we simply verify $T(3) = 13 \times 9 \in [\frac{27}{10}, 10 \times 27]$,
  $T(4) = 13 \times 16 \in [\frac{64}{10}, 64 \times 10]$,
  $T(5) = 13 \times 25 \in [\frac{125}{10}, 125 \times 10]$.
  Induction step: Assuming the required holds for $T(n)$, we show it also holds for $T(n+3)$.

$$\begin{aligned} T(n+3) &= 13(n+3)^2 + T(n) \overset{\text{IH}}{\leq} 13(n+3)^2 + 10n^3 \\ &= 10n^3 + 13n^2 + 78n + 117 = 10\left(n^3 + 1.3n^2 + 7.8n + 11.7\right) \\ &\leq 10\left(n^3 + 9n^2 + 27n + 27\right) = 10(n+3)^3 \end{aligned}$$

$$\begin{aligned} T(n+3) &= 13(n+3)^2 + T(n) \overset{\text{IH}}{\geq} 13(n+3)^2 + 0.1n^3 \\ &= 0.1n^3 + 13n^2 + 78n + 117 = 0.1\left(n^3 + 130n^2 + 780n + 1170\right) \\ &\geq 0.1\left(n^3 + 9n^2 + 27n + 27\right) = 0.1(n+3)^3 \quad \square \end{aligned}$$

Another example

- procedure InsertionSort(A, n)

```
if (n > 1) then
    InsertionSort(A, n − 1)
    x ← A[n]
    PutInPlace(A, n − 1, x)
```

procedure PutInPlace(A, j, x)

```
if (j = 0) then
    A[1] ← x
else if (x > A[j]) then
    A[j + 1] ← x
else                ** i.e., x ≤ A[j]
    A[j + 1] ← A[j]
    PutInPlace(A, j − 1, x)
```

- Let $T(n)$ = Worst-case #KC made by InsertionSort on input of size n.
- $T(n) = \begin{cases} 0, & \text{if } n = 1 \\ T(n-1) + T_{PI}(n-1), & \text{if } n > 1 \end{cases}$
  with T_{PI} = worst-case #KC in PutInPlace
- Because we solved $T_{PI}(n) = n$ we get
  $$T(n) = \begin{cases} 0, & \text{if } n = 1 \\ T(n-1) + n - 1, & \text{if } n > 1 \end{cases}$$
- HW: Solve this.

Recurrence relations — merge sort analysis

- Merge sort recall:
  - Divide the whole list into 2 sublists of equal size;
  - Recursively merge sort the 2 sublists;
  - Combine the 2 sorted sublists into a sorted list: uses $\leq n-1$ KC
- Assumptions:
  - n (number of keys in the whole list) is a power of 2;
    This makes the analysis easier (since each time we are dividing by 2)
  - Let $T(n)$ denote $\#$KC for a list of size n
- Deriving recurrence relation:
  - Merge sort on 2 sublists $2 \times T(\frac{n}{2})$
  - Assembling needs $n-1$ KC (in the WC)
  - $T(n) = \begin{cases} 0 & , \quad \text{if} n = 1 \\ (n-1) + 2 \cdot T(\frac{n}{2}) & , \quad \text{otherwise} \end{cases}$
- Solving recurrence relation:

Merge sort analysis — solving the recurrence relation

- Particular case:
  $T(1) = 0$,
  $T(2) = 1$,
  $\ldots$
- General case:

$$\begin{array}{rcl} T(n) & = & (n-1) + 2 \times T(\frac{n}{2}) \\ & = & (n-1) + 2 \times \left((\frac{n}{2} - 1) + 2 \times T(\frac{n}{4})\right) \\ & = & \ldots \end{array}$$

Solving Merge Sort (Cont'd)

- We assume $n = 2^k$ so:

$$\begin{aligned}
T(2^k) &= (2^k - 1) + 2 \times T(2^{k-1}) \\
&= (2^k - 1) + 2 \times \left((2^{k-1} - 1) + 2 \times T(2^{k-2})\right) \\
&= (2^k - 1) + (2^k - 2) + 2^2 \times T(2^{k-2}) \\
&= (2^k - 1) + (2^k - 2) + 2^2 \times \left((2^{k-2} - 1) + 2 \times T(2^{k-3})\right) \\
&= (2^k - 1) + (2^k - 2) + (2^k - 2^2) + 2^3 \times T(2^{k-3}) \\
&= (2^k - 2^0) + (2^k - 2^1) + (2^k - 2^2) + 2^3 \times T(2^{k-3}) \\
&= (2^k - 2^0) + (2^k - 2^1) + (2^k - 2^2) + (2^k - 2^3) + 2^4 \times T(2^{k-4}) \\
&= \ldots \\
&= (2^k - 2^0) + (2^k - 2^1) + (2^k - 2^2) + \ldots + (2^k - 2^{k-1}) + 2^k \times T(2^{k-k}) \\
&= (2^k - 2^0) + (2^k - 2^1) + (2^k - 2^2) + \ldots + (2^k - 2^{k-1}) \\
&= k \times 2^k - \textstyle\sum_{i=0}^{k-1} 2^i \\
&= (k-1)2^k + 1
\end{aligned}$$

  Since $n = 2^k$, we have $k = \lg n$. So, $T(n) = n(\lg n - 1) + 1$.

1. Variable substitution makes guessing easy ...
2. In recurrence solving always assume n being some power whenever necessary (ignore floor and ceiling).
3. Need to transform back to original variable.
4. Don't forget: This is just a guess. Must be followed by proof (by induction).

Closed form proof by induction:

- Recurrence: $T(n) = \begin{cases} 0 & \text{if } n = 1 \\ (n-1) + 2 \times T(\frac{n}{2}) & \text{if } n \geq 2 \end{cases}$
  Guessed closed form: $T(n) = n(\lg n - 1) + 1, n \geq 1$
- Assuming $n = 2^k, k \geq 0$
- Base case: $T(1) = 0$ and indeed $1(\lg(1) - 1) + 1 = 0$.
- Inductive step: Assuming that $T(2^k) = 2^k(k-1) + 1$, $k \geq 0$, want to show $T(2^{k+1}) = 2^{k+1}k + 1$.
  By recurrence relation,

$$\begin{aligned} T(2^{k+1}) &= (2^{k+1} - 1) + 2 \times T(2^k) \\ &= (2^{k+1} - 1) + 2^{k+1}(k-1) + 2 \\ &= k2^{k+1} + 1. \quad \blacksquare \end{aligned}$$

- Extending to n which isn't a power of 2 is just tedious.
- ... and also uninteresting if we assume the runtime is monotone:
  Since $\exists$ integer k s.t. $n \leq 2^k < 2n$ (why?), then:
  $T(n) \leq T(2^k) = 2^k(k-1) + 1 \leq (2n) \cdot (\lg(2n) - 1) + 1 = 2n \lg(n) + 1$.
  Similarly, $T(n) \geq T(2^{k-1}) = 2^{k-1}(k-2) + 1 \geq \frac{1}{2}n(\lg(n) - 2)$
- Conclusion: $T(n) \in \Theta(n \log(n))$.

How NOT to Prove a Recursion

- Here's a wrong guess $T(n) \in O(n)$ with a *wrong* proof.
- Let's prove by induction that $T(n) = O(n)$ for any natural n.
  - Base case: clearly $T(1), T(2), T(3), T(4)$ are all $O(1)$.
  - Induction step: assume that $T(i) = O(i)$ for any $1 \leq i < n$ and we have

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + n - 1 \\ &= 2 \cdot O(\tfrac{n}{2}) + O(n) \\ &= O(n) + O(n) + O(n) = O(n) \quad \blacksquare \end{aligned}$$

- The problem is that the statement "$T(i) = O(i)$ for any $1 \leq i < n$" is meaningless!
  - big-O notation is asymptotic!
  - That is why it is wiser to use $\in O(f(n))$ rather than $= O(f(n))$.
- Your induction should *always* prove the implicit statement.
  In our case: $\exists c > 0, n_0$ such that $T(n) \leq c \cdot n$ for any $n \geq n_0$.
- If you try to prove this you run into difficulties:
  - Induction step: assume that $T(i) \leq c \cdot i$ for any $n_0 \leq i < n$ and we have

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + n - 1 \\ &= 2c \cdot \tfrac{n}{2} + n - 1 \\ &= cn + (n-1) \not\leq c \cdot n \end{aligned}$$

  - We failed $\Rightarrow$ we need to change our guess.

From #KC to Running Time Analysis:

- So #KC in `MergeSort` $\in \Theta(n \log(n))$.
  We now wish to deduce that WC running time is $\Theta(n \log n)$
- Which direction is obvious?
  - Lower bound: even if each KC takes one "unit of time" then our running time is $n(\lg(n) - 1) + 1 \geq \frac{1}{2} n \lg(n) \in \Omega(n \lg(n))$
  - Note: this follows because we proved $T(n) = \Theta(n \log(n))$. Had we only proven upper bound (big-O), it wasn't enough to derive the $\Omega(\cdot)$ conclusion.
- Upper bound: `Merge` takes $O(n)$ times. So $\exists c_1, n_0$ such that its running time $\leq c_1 n$ on input of size $n \geq n_0$.
- `Merge-Sort` takes $O(1)$ time on any input of size $\leq n_0$.
- Hence, if $R(n)$ denotes the running time of `Merge-Sort` on n-size input, we have
  $$R(n) = \begin{cases} c_3, & \text{if } n \leq n_0 \\ c_1 \cdot n + c_2 + 2R(\frac{n}{2}), & \text{if } n \geq n_0 \end{cases}$$
- Set C to be any number $\geq c_1 + c_2 + c_3$ and we get
  $R(n) \leq \begin{cases} C, & \text{if } n \leq n_0 \\ Cn + 2R(\frac{n}{2}), & \text{if } n \geq n_0 \end{cases}$,and this recursion solves to $C \cdot n(\lg(n))$.
- Conclusion: merge sort WC running time is $\Theta(n \log n)$.

An exercise:

- Examine the running time of $\mathtt{QZ}(n)$

```
procedure QZ(n)

if (n > 1) then
    a ← n × n + 37
    b ← a × QZ(n/2)
    return QZ(n/2) × QZ(n/2) + n
else
    return n × n
```

- If we only consider arithmetic operations then:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3T(\frac{n}{2}) + 5 & \text{if } n \geq 2 \end{cases}$$

- Again, we use Iterated Substitution to obtain a proper guess
- Then we prove our guess by induction

Exercise (Cont'd):

- For simplicity, assume n is a power of 2, say $n = 2^k$:
-

$$
\begin{aligned}
T(2^k) &= 3 \times T(2^{k-1}) + 5 \\
&= 3 \times \left(3 \times T(2^{k-2}) + 5\right) + 5 \\
&= 3^2 \times T(2^{k-2}) + 3 \times 5 + 5 \\
&= 3^2 \times \left(3 \times T(2^{k-3}) + 5\right) + 3 \times 5 + 5 \\
&= 3^3 \times T(2^{k-3}) + 3^2 \times 5 + 3 \times 5 + 5 \\
\\
&= \ldots \\
&= 3^k \times T(2^{k-k}) + 3^{k-1} \times 5 + 3^{k-2} \times 5 + \ldots + 3 \times 5 + 5 \\
&= 3^k + 5 \times \left(\sum_{i=0}^{k-1} 3^i\right) \\
&= 3^k + 5 \times \left(\frac{3^k - 1}{2}\right) \\
&= 3.5 \times 3^k - 2.5
\end{aligned}
$$

- So, our guess is: $T(n) = 3.5 \times 3^{\log n} - 2.5 = 3.5 \times n^{\log 3} - 2.5$.

Exercise (Cont'd):

- Next, prove $T(2^k) = 3.5 \times 3^k - 2.5$, for $k \geq 0$, by induction
- Base step: $k = 0$ and $T(2^0) = 1 = 3.5 - 2.5$.
- Inductive step: Assume that $T(2^{k-1}) = 3.5 \times 3^{k-1} - 2.5$.
  By recurrence relation
  $$T(2^k) = 3 \times T(\tfrac{2^k}{2}) + 5 = 3 \times T(2^{k-1}) + 5,$$
  so
  $$T(2^k) = 3 \times \Big(3.5 \times 3^{k-1} - 2.5\Big) + 5 = 3.5 \times 3^k - 2.5.$$
  Thus, it holds for inductive step too.
- Therefore, $T(2^k) = 3.5 \times 3^k - 2.5$ holds for any $k \geq 0$.
- Namely, $T(n) = 3.5 \times 3^{\log_2(n)} - 2.5 = 3.5 \times n^{\log_2(3)} - 2.5 \in \Theta(n^{\log_2(3)})$.

Other Techniques for Solving Recurrence Relations:

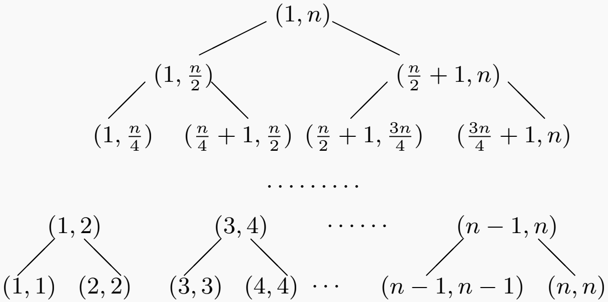
- Recurrence tree - a visual approach towards finding solution
  - Draw down a rooted tree. Each node represents a call to the recursive function.
  - The root: the first (original) call on an instance of size n
  - For each node — its children are the recursive calls this node makes (one node per each call).
  - And so the leafs = the calls to the function where the base-case is applied and there are no further recursive calls.
  - $\Rightarrow$ the number of nodes in the tree — the total number of function calls we make during the entire computation.
  - Assign to each node a weight: the amount of work done by this (single) function call
  - So the overall execution time: the sum of all weights in all the nodes.

Other Techniques for Solving Recurrence Relations:

- Recurrence tree - a visual approach towards finding solution
- "Guess and Test" - a guessing approach for finding & proving a solution.
  - Guess that $T(n)$ solves to $\Theta(f(n))$.
  - Look for constants $c, d > 0$ such that $T(n) \leq c \cdot f(n)$ and $T(n) \geq d \cdot f(n)$ for all sufficiently large n — by trying to prove the claim: $d \cdot f(n) \leq T(n) \leq c \cdot f(n)$ inductively.
  - Both the base case and the inductive steps should induce constraints on c and d.
  - If your guess is right, then there will be c and d satisfying all the constraints you've collected;
  - If your guess is wrong, you will find no c / no d satisfy these constraints, and you have to adjust your guess and try again.
- Read more in the following slides, and in CLRS (p. 83-92)

2- Recurrence tree :

- This is another method to find a recurrence relation's solution
- Not so formal as iterated substitution; more visual
- Represent the computation as a rooted tree: each recursive call is represented by a node
  - The root is the first call (to the instance of size n)
  - For every node — its children are the recursive calls made in the execution of the node's call
  - The leaves — the base case of the recursion
- Consider the Merge-Sort and the tree for the recursive calls of it:



- Question: the number of KC per cell?

Merge sort recursion tree (KC per cell):

- To each node you assign a weight: the amount of work done by this call (not including the recursive calls)
- The total amount of work you do: the sum of weights of all nodes.
- Assuming `merge`(n) takes $\sim n$ KC:

$$
\begin{array}{cccccccc|ll}
 & & & (1, n) & & & & & \texttt{level 1:} & n \\
 & (1, \frac{n}{2}) & & & & (\frac{n}{2}+1, n) & & & \texttt{level 2:} & n \\
(1, \frac{n}{4}) & & (\frac{n}{4}+1, \frac{n}{2}) & & (\frac{n}{2}+1, \frac{3n}{4}) & & (\frac{3n}{4}+1, n) & & \texttt{level 3:} & n \\
 & & & \cdots\cdots\cdots & & & & & \cdots & \\
(1,2) & & (3,4) & & \cdots\cdots & & (n-1, n) & & \texttt{level } k\texttt{:} & n \\
(1,1) \; (2,2) & & (3,3) \; (4,4) & \cdots & & (n-1, n-1) & & (n,n) & \texttt{level } k+1\texttt{:} & 0 \\
 & & & & & & & & \texttt{total:} & kn
\end{array}
$$

Where $k = \log n$.

- Therefore, the running time of Merge-Sort is (as found before): $\Theta(n \log n)$.
- Note: the recurrence tree method is not as applicable nor as formal as the iterated substitution.

3- Guess and Test method:

- First make a guess for the closed form of the recurrence
- Guess can come from the iterated substitution, recurrence tree, or previous experiences
  - **But regardless of the method, your guess must to be verified!**
- Prove the guess by induction
- May have to change the guess if the inductive proof fails

- **Example:** Find a closed form for
  $$T(n) = \begin{cases} T(\frac{n}{2}) + 2T(\frac{n}{4}) + 2n & \text{if } n \geq 4 \\ i & \text{if } 1 \leq n \leq 3 \end{cases}$$
- **Solution:** We guess that $T(n) \in \Theta(n \log n)$
- Need to show that there are constants $c, d > 0$ and naturals n_0, n_1 such that:
  (i) $T(n) \leq cn \log n$ for any $n \geq n_0$
  and (ii) $T(n) \geq dn \log n$ for any $n \geq n_1$.

- (i) Base case: $T(4) = T(2) + T(1) + 8 = 2 + 1 + 8 = 11$ so $T(4) \leq c \cdot 4 \cdot \lg(4) = 8c$ for $c \geq 11/8$.
- Assume $T(i) \leq ci \log i$ for all values of $i < n$, with $i \geq 4$. (Note the use of full induction!)

$$\begin{aligned} T(n) &= T(\tfrac{n}{2}) + 2T(\tfrac{n}{4}) + 2n \\ &\leq c\tfrac{n}{2}\log\tfrac{n}{2} + 2c\tfrac{n}{4}\log\tfrac{n}{4} + 2n \\ &\leq c\tfrac{n}{2}(\log n - 1) + c\tfrac{n}{2}(\log n - 2) + 2n \\ &= cn\log n + (2 - \tfrac{3c}{2})n \leq cn\log n, \end{aligned}$$

  if we take $c \geq \frac{4}{3}$.
- We have shown that for $c = \frac{11}{8} = \max\{\frac{4}{3}, \frac{11}{8}\}$ and $n \geq 4$: $T(n) \leq \frac{11}{8} \cdot n\log n$.
- Note that we could have started with a guess of $c = 100$ and the induction would follow through too...

- (ii) $T(n) \geq \frac{1}{100} n \log n$ for any $n \geq 4$.
- Base case: $T(4) = 11 \geq \frac{1}{100} \cdot 4 \lg(4)$.
- Induction step: Assume $T(i) \geq \frac{1}{100} i \log i$ for all values of $i < n$, with $i \geq 4$.

$$
\begin{array}{rcl}
T(n) & = & T(\frac{n}{2}) + 2T(\frac{n}{4}) + 2n \\
& \geq & \frac{1}{100} \cdot \frac{n}{2} \log \frac{n}{2} + 2 \cdot \frac{1}{100} \cdot \frac{n}{4} \log \frac{n}{4} + 2n \\
& \geq & \frac{n}{200} \left(\log n - 1 + \log n - 2 \right) + 2n \\
& \geq & \frac{2n \log(n)}{200} + \left(2 - \frac{3}{200} \right) n \\
& \geq & \frac{1}{100} n \log n
\end{array}
$$

- Combining (i) + (ii) we get: $T(n) \in \Theta(n \log n)$.

- Note: Sometimes we need to revise our guess
- The correct guess is not always obvious; the method requires practice;

4- Master Theorem Method

The next method we see is to use a theorem called Master Theorem.
(It is proven using the iterative substitution method.)

- **Master Theorem:**
  Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function.
  Let $T(n)$ be defined by the recurrence

  $$T(n) = aT(\frac{n}{b}) + f(n).$$

  Then $T(n)$ can be bounded asymptotically as follows:

  1. If $f(n) \in O(n^{\log_b a - \epsilon})$, for some $\epsilon > 0$ then $T(n) \in \Theta(n^{\log_b a})$,
  2. If $f(n) \in \Theta(n^{\log_b a} \log^k n)$ for ~~some constant $\epsilon > 0$ and~~ some $k \geq 0$ then $T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$,
  3. If $f(n) \in \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(\frac{n}{b}) \leq \delta f(n)$ for some constant $\delta < 1$ and all sufficiently large n, then $T(n) \in \Theta(f(n))$.

Some examples:

1. $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 7T(\frac{n}{2}) + n^2 & \text{if } n \geq 2 \end{cases}$
   $a = 7$, $b = 2$, $f(n) = n^2 \Rightarrow \log_b a = \log_2 7 > 2.8$, so $f(n) \in O(n^{\lg 7 - 0.1})$ and $T(n) \in \Theta(n^{\lg 7})$
2. $T(n) = \begin{cases} 1 & \text{if } n \leq 2 \\ 14T(\frac{n}{3}) + n^3 & \text{if } n \geq 3 \end{cases}$
   $a = 14, b = 3, f(n) = n^3 \Rightarrow \log_b(a) = \log_3(14) \in (2, 3)$, since $f(n) \in \Omega(n^{\log_3(14)+\epsilon})$ for $\epsilon = \frac{3 - \log_3(14)}{2}$, and $14\left(\frac{n}{3}\right)^3 \leq \frac{14}{27}n^3$ (i.e. with $\delta = 2/3$ in case 3) $T(n) \in \Theta(n^3)$
3. $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\frac{n}{2}) + n & \text{if } n \geq 2 \end{cases}$
   $a = 2$, $b = 2$, $f(n) = n$, since $n^{\log_b a} = n = f(n)$, we have $f(n) \in \Theta(n^{\log_2 2} \log^0 n)$ and so (by case 2) $T(n) \in \Theta(n \log n)$.
4. $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 5T(\frac{n}{2}) + n^2 \log n & \text{if } n \geq 2 \end{cases}$
   $a = 5$, $b = 2$, $f(n) = n^2 \log n$. So $\log_b(a) = \lg 5 > 2.3$, so $f(n) \in O(n^{\lg 5 - 0.1})$ and $T(n) \in \Theta(n^{\lg 5})$
5. $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 9T(\frac{n}{3}) + n^2 \log^5 n & \text{if } n \geq 2 \end{cases}$
   $a = 9$, $b = 3$, $f(n) = n^2 \log^5 n$. Since $\log_b a = 2$: $f(n) \in \Theta(n^{\log_b a} \log^5 n)$. Thus (by case 2): $T(n) \in \Theta(n^2 \log^6 n)$.

Master Theorem doesn't always apply:

$$T(n) = \begin{cases} 4T(\frac{n}{2}) + \frac{n^2}{\log n} & \text{if } n \geq 2 \\ 1 & \text{if } n = 1 \end{cases}$$

$a = 4, b = 2, \log_b a = 2$

$f(n) = \frac{n^2}{\log n} \notin \Theta(n^2)$;

$f(n) = \frac{n^2}{\log n} \in O(n^2)$ but $f(n) = \frac{n^2}{\log n} \notin O(n^{2-\epsilon})$ for any positive constant ϵ.

What we can do to get the closed form?

— iterated substitution!

$$
\begin{aligned}
& T(2^k) \\
= \quad & 4 \times T(2^{k-1}) + \tfrac{2^{2k}}{k} \\
= \quad & 4^2 \times T(2^{k-2}) + 4 \times \tfrac{2^{2(k-1)}}{k-1} + \tfrac{2^{2k}}{k} \\
= \quad & 4^2 \times T(2^{k-2}) + \tfrac{2^{2k}}{k-1} + \tfrac{2^{2k}}{k} \\
\\
= \quad & 4^3 \times T(2^{k-3}) + 4^2 \times \tfrac{2^{2(k-2)}}{k-2} + \tfrac{2^{2k}}{k-1} + \tfrac{2^{2k}}{k} \\
= \quad & 4^3 \times T(2^{k-3}) + \tfrac{2^{2k}}{k-2} + \tfrac{2^{2k}}{k-1} + \tfrac{2^{2k}}{k} \\
\\
= \quad & 4^k \times T(1) + \tfrac{2^{2k}}{k-(k-1)} + \ldots + \tfrac{2^{2k}}{k-1} + \tfrac{2^{2k}}{k} \\
= \quad & 4^k \times T(1) + 2^{2k}\left(\tfrac{1}{1} + \tfrac{1}{2} + \tfrac{1}{3} + \ldots + \tfrac{1}{k}\right) \\
= \quad & 4^k \times T(1) + 4^k \times H(k)
\end{aligned}
$$

Therefore, $T(n) = n^2 \times T(1) + n^2 \times H(\log n) \in \Theta(n^2 H(\log n))$. Further we have $H(k) \in \Theta(\log k)$ (in fact $H(k) = \ln k + \Theta(1)$), thus $T(n) \in \Theta(n^2 H(\log n)) = \Theta(n^2 \log(\log(n)))$.

An exercise — dealing with floor & ceiling:

Prove that $T(n)$ defined by the following recurrence is in $O(\log n)$:

$$T(n) = \begin{cases} 1, & \text{if } n = 1, \\ T(\lceil \frac{n}{2} \rceil) + 1, & \text{if } n \geq 2 \end{cases}$$

- Examine some small cases:
  $T(1) = 1$
  $T(2) = 2$
  $T(3) = T(4) = 3$
  $T(5) = T(6) = T(7) = T(8) = 4$
  $\ldots$
  Guess: $T(n) = k + 1$, for any $2^{k-1} < n \leq 2^k$
- Prove the above guessed (by induction).
- Now you only need to get the closed form for n being a power of 2 ...
- By iterated substitution, $T(2^k) = k + 1$ (again, prove by induction)
  So, $T(n) = \log n + 1$ for any n which is a power of 2.
- Now, prove by induction on k that for any n satisfying $2^{k-1} < n \leq 2^k$ we have $T(n) = k + 1$.
- Conclusion: since $T(n) = \lceil \log n \rceil + 1 \leq \log(n) + 2 \leq 2\log(n)$, for $n \geq 4$, $T(n) \in O(\log(n))$

Summary:

- When analyzing the runtime of a recursive code — express the runtime / the cost of a key-operation using a recurrence relation
  - Remember that this is the **most important** step.
  - Make sure you understand the code, you follow it line-by-line, and that you are able to *clearly explain* how you derived this particular relation.
- To solve the recurrence relation:
  - Find the solution — using iterated substitution
    Plugging in repeatedly the value of $T(i)$ until a pattern emerges
  - Prove it using induction
- Your induction proof MUST use explicit constants
  - And NEVER an induction hypothesis of the form $T(n) = O(f(n))$
- Recurrence relations of the type: $aT(n/b) + f(n)$ — use Master Theorem
  - But you have to check that it indeed applies (we fall into one of the three cases)
  - And explain which case it falls into