Recurrences
(Method 4)

Alexandra Stefan

Recurrences

- Recursive algorithms
  - It may not be clear what the complexity is, by just looking at the algorithm.

  - In order to find their complexity, we need to:
    - Express the "running time" of the algorithm as a recurrence formula. E.g.: $f(N) = N + f(N-1)$
    - Find the complexity of the recurrence:
      - Expand it to a summation with no recursive term.
      - Find a concise expression (or upper bound), E(n), for the summation.
      - Find Θ,ideally, or O (big-Oh) for E(n).

- Recurrence formulas may be encountered in other situations:
  - Compute the number of nodes in certain trees.
  - Express the complexity of non-recursive algorithms (e.g. selection sort).

Common Recurrences

1. Reduce the problem size by 1 in <u>constant</u> time T(N) =
2. Reduce the problem size by 1 in <u>linear</u> time T(N) =
   – E.g. Check all items, eliminate 1
3. *Halve* problem in <u>constant</u> time T(N) =
4. *Halve* problem in <u>linear</u> time T(N) =
5. Break the problem into *2 halves* in <u>constant</u> time T(N) =
6. Break the problem into *2 halves* in <u>linear</u> time T(N) =

Generic expressions for recurrences

- The common recurrences on the previous slide have a common-type solution.
- After you solve a couple simple recurrences, consider the following generic ones:

G1. T(N) = c + k*T(N-s)

G2. T(N) = c + k*T(N/s)

G3. T(N) = N + k*T(N-s)

G4. T(N) = N + k*T(N/s)

- Pay attention to what each constant above affects:
  - The recursive term (T(N-s) or T(N/s)) => number of steps
  - c – can be ignored
  - k – cannot be ignored
  - Safer: do not ignore any constant when you expand the recurrence.

More Recurrences

M1. Reduce the problem size by 1 in logarithmic time

- E.g. Check lg(N) items, eliminate 1

M2. Reduce the problem size by 1 in N^2 time

- E.g. Check N^2 pairs, eliminate 1 item

M3. Algorithm that:

- takes $\Theta(1)$ time to go over N items.
- calls itself 3 times on data of size N-1.
- takes $\Theta(1)$ time to combine the results.

*M4. ** * Algorithm that:

- calls itself N times on data of size N/2.
- takes $\Theta(1)$ time to combine the results.
- This generates a difficult recursion.

Case 1: Reduce the problem size by 1 in constant time

- In this case, the algorithm proceeds in a sequence of similar steps, where at each step eliminates one item.
- Any examples of such an algorithm?

Case 1: Reduce the problem size by 1 in constant time

- In this case, the algorithm proceeds in a sequence of similar steps, where at each step eliminates one item.
- If the problem size is 1, it takes constant time to solve it (no recursive call needed).
- Any examples of such an algorithm?
  - Sequential search (recursive solution).
  - Recursive solution of sum from 1 to N.

Case 1: Reduce the problem size by 1 in constant time

- Let $T(N)$ be the running time.
- Then, $T(N) = ???$

Case 1: Reduce the problem size by 1 in constant time

- Let $T(N)$ be the running time.
- Then, $T(N) = 1 + T(N-1)$
- And $T(1) = 1$

Case 1: Reduce the problem size by 1 in constant time

- T(1) = 1
- T(N) = T(N-1) + 1
- $$
\begin{aligned}
T(N) &= 1 + T(N-1) && \text{(step 1)}\\
&= 1 + 1 + T(N-2) && \text{(step 2)}\\
&= 1 + 1 + 1 + T(N-3) && \text{(step 3)}\\
&\;\;...\\
&= 1 + 1 + \ldots + 1 + T(N-i) && \text{(step i: i of 1)}\\
&\;\;...\\
&= 1 + 1 + \;.\;\;.\;\;.\; + 1 + 1 + T(1) && \text{(step N-1)}\\
&= \underbrace{1 + 1 + \;.\;\;.\;\;.\; + 1 + 1 + 1}_{N} && \text{(step N-1)}\\
&= N\\
&= \Theta(N)
\end{aligned}
$$
- Conclusion: The algorithm takes linear time.

Case 1: Eliminating the recursive term

- To compute the number of steps
  - We want the term T(N-i) to be T(1) so we want:
  - N-i = 1 => i = N-1

- Note that if we used a constant c instead of 1, we would get the same complexity. Look at:
  - T(N) = c + T(N-1) and T(1) = c

Case 1: Reduce the problem size by 1 in constant time

- T(N) = T(N-1) + **c**, T(1) = c
- $$\begin{aligned}
T(N) &= c + T(N-1) && \text{(step 1)}\\
&= c + c + T(N-2) && \text{(step 2)}\\
&= c + c + c + T(N-3) && \text{(step 3)}\\
&\ldots\\
&= c + c + \ldots + c + T(N-i) && \text{(step i: i of c)}\\
&\ldots\\
&= c + c + . \;.\; . + c + c + T(1) && \text{(step N-1)}\\
&= \underbrace{c + c + . \;.\; . + c + c + c}_{\text{N terms}} && \text{(step N-1)}\\
&= c*N\\
&= \Theta(N)
\end{aligned}$$
- Conclusion: The algorithm takes linear time.

Case 2: Check All Items, Eliminate One

- In this case, the algorithm proceeds in a sequence of similar steps, where:
  - each step loops through all items in the input, and eliminates one item.
- Any examples of such an algorithm?

Case 2: Check All Items, Eliminate One

- In this case, the algorithm proceeds in a sequence of similar steps, where:
  - each step loops through all items in the input, and eliminates one item.
- Any examples of such an algorithm?
  - Selection Sort.

Case 2: Check All Items, Eliminate One

- Let T(N) be the running time.
- Then, T(N) = ???

Case 2: Check All Items, Eliminate One

- Let $T(N)$ be the running time.
- Then, $T(N) = T(N-1) + N$ (assume $T(1) = 1$).
  - Because we need to examine all items (N units of time), and then we need to run the algorithm on N-1 items.

Case 2: Check All Items, Eliminate One

- Let T(N) be the running time.
- Then, T(N) = T(N-1) + N (assume T(1) = 1).
- $$T(N) \overset{1}{=} N + T(N-1)$$
  $$\overset{2}{=} N + (N-1) + T(N-2)$$
  $$\overset{3}{=} N + (N-1) + (N-2) + T(N-3)$$
  $$\ldots$$
  $$\overset{i}{=} N + (N-1) + (N-2) + \ldots (N-(i-1)) + T(N-i)$$
  $$\ldots$$
  $$\overset{N-1}{=} N + (N-1) + \ldots + 3 + 2 + T(1)$$
  $$\overset{N-1}{=} N + (N-1) + \ldots + 3 + 2 + 1$$
  $$= (N*(N + 1)) / 2$$
  $$= (N^2 + N)/2$$
  $$= \Theta(N^2)$$
- Conclusion: The algorithm takes quadratic time.

Case 2: Check All Items, Eliminate One

- Variation: use $T(1) = c$ (instead of $T(1) = 1$)
- Then, $T(N) = T(N-1) + N$
- $T(N) \overset{1}{=} N + T(N-1)$

$$\overset{2}{=} N + (N-1) + T(N-2)$$
$$\overset{3}{=} N + (N-1) + (N-2) + T(N-3)$$
$$\ldots$$
$$\overset{i}{=} N + (N-1) + (N-2) + \ldots (N-(i-1)) + T(N-i)$$
$$\ldots$$
$$\overset{N-1}{=} N + (N-1) + \ldots + 3 + 2 + T(1)$$
$$\overset{N-1}{=} N + (N-1) + \ldots + 3 + 2 + c$$
$$= N + (N-1) + \ldots + 3 + 2 + 1 + c - 1$$
$$= c-1 + (N(N + 1) / 2)$$
$$= c-1 + ((N^2 + N)/2)$$
$$= \Theta(N^2)$$

Conclusion: The algorithm takes quadratic time.

Case 3: Halve the Problem in Constant Time

- Perform a *constant* number of operations, and then reduce the size of the input by *half*.
- Any example of such an algorithm?

Case 3: Halve the Problem in Constant Time

- Perform a *constant* number of operations, and then reduce the size of the input by *half*.
- Any example of such an algorithm?
  - Binary Search.

Case 3: Halve the Problem in Constant Time

- Whenever we analyze recursive algorithms that "halve the problem" it will be easier in the mathematical derivation to write the data size, N, as a power of 2: **$N = 2^n$**
- In this case we can replace one variable for the other. We can write the expressions in terms of N or n using: **$N = 2^n$ or $n = \lg(N)$**
- In all the following problems that have a recursive call for a problem half the size (e.g. $T(N) = \ldots T(N/2) \ldots$) we can use either N or n:
  - With N: $T(N) = \ldots T(N/2) \ldots = \ldots T(N/4) \ldots = \ldots T(N/2^i) \ldots$
  - With n: $T(2^n) = \ldots T(2^{n-1}) \ldots = \ldots T(2^{n-2}) \ldots = \ldots T(2^{n-i}) \ldots$
  - You can use whichever notation you prefer.

Case 3: Halve the Problem in Constant Time

- In this case, each step of the algorithm consists of:
  - performing a constant number of operations, and then reducing the size of the input by half.
- $\mathbf{T(2^n) = ???}$

Case 3: Halve the Problem in Constant Time

- In this case, each step of the algorithm consists of:
  - performing a constant number of operations, and then reducing the size of the input by half.
- $$\begin{aligned} \mathtt{T}(2^n) &= 1 + \mathtt{T}(2^{n-1}) \\ &= 2 + \mathtt{T}(2^{n-2}) \\ &= 3 + \mathtt{T}(2^{n-3}) \\ &\ \ldots \\ &= n + \mathtt{T}(2^0) \\ &= n + 1. \end{aligned}$$
- $\Theta(n)$ time for $N = 2^n$.
- Substituting *n* with *lg N*: $\Theta(lg\ N)$ time.

Case 4: Halve the Problem in Linear Time

- Perform a linear (i.e., $O(N)$) number of operations, and then reduce the size of the input by half.
- **`T(N) = ???`**

Case 4: Halve the Problem in Linear Time

- Perform a linear (i.e., *O(N)*) number of operations, and then reduce the size of the input by half.
- **`T(N) = T(N/2) + N`**

```
T(N) = T(N/2) + N
     = T(N/4) + N/2 + N
     = T(N/8) + N/4 + N/2 + N
     ...
     = 1 + 2 + 4 + ... + N/4 + N/2 + N
     = ??? (do you recognize this series?)
```

Case 4: Halve the Problem in Linear Time

- In this case, each step of the algorithm consists of:
  - Performing a linear (i.e., *O(N)*) number of operations, and then reducing the size of the input by half.
- ```
  T(N) = T(N/2) + N
       = T(N/4) + N/2 + N
       = T(N/8) + N/4 + N/2 + N
       ...
       = 1 + 2 + 4 + ... + N/4 + N/2 + N
       = ???
  ```
- $1 + 2 + 4 + \ldots + 2^n = ???$
- **What is the general formula for the above series?**
  ```

Case 4: Halve the Problem in Linear Time

- In this case, each step of the algorithm consists of:
  - Performing a linear (i.e., *O(N)*) number of operations, and then reducing the size of the input by half.
- **T(N) = T(N/2) + N**

```
T(N) = T(N/2) + N
     = T(N/4) + N/2 + N
     = T(N/8) + N/4 + N/2 + N
     ...
     = 1 + 2 + 4 + ... + N/4 + N/2 + N
     = about 2N
```

- *Θ(N)* time.

Case 5: Break Problem Into Two Halves in Constant Time

- The algorithm does:
  - *Constant* number of operations to *split the problem into two halves*.
  - Calls itself *recursively on each half*.
  - *Constant* number of operations to *combine the two answers*.
- **`T(N) = ???`**

Case 5: Break Problem Into Two Halves in Constant Time

- The algorithm does:
  - *Constant* number of operations to *split the problem into two halves*.
  - Calls itself *recursively on each half*.
  - *Constant* number of operations to *combine the two answers*.
- **T(N) = 2T(N/2) + 1**

```
T(N) = 2T(N/2) + 1
     = 4T(N/4) + 2 + 1
     = 8T(N/8) + 4 + 2 + 1
     ...
     = about 2N
```

Case 6: Break Problem Into Two Halves in Linear Time

- The algorithm does:
  - *N* operations to split the problem into two halves.
  - Calls itself recursively on each half.
  - *N* operations to combine the two answers.
- **`T(N) = ???`**

Case 6: Break Problem Into Two Halves in Linear Time

- The algorithm does:
  - *N* operations to split the problem into two halves.
  - Calls itself recursively on each half.
  - *N* operations to combine the two answers.
- ```
  T(N) = 2T(N/2) + N
       = 4T(N/4) + N + N
       = 8T(N/8) + N + N + N
       ...
       = N lg N
  ```
  ```

Case 6: Break Problem Into Two Halves in Linear Time

- The algorithm does:
  - *N* operations to split the problem into two halves.
  - Calls itself recursively on each half.
  - *N* operations to combine the two answers.
- Example?

‘General’ recurrence expressions

G1. $T(N) = c + k*T(N-s)$

G2. $T(N) = c + k*T(N/s)$

G3. $T(N) = N + k*T(N-s)$

G4. $T(N) = N + k*T(N/s)$

Pay attention to what each constant above affects:

- The recursive term (T(N-s) or T(N/s)) => number of steps
- c – can be ignored
- k – cannot be ignored
- Safer: do not ignore any constant when you expand the recurrence.

Solving Recurrences: Example M1

- Example that produces the function we just analyzed: T(N) = $\sum_{k=1}^{N} k^2$

Solving Recurrences: Example M1

- Suppose that we have an algorithm that at each step:
  - takes $O(N^2)$ time to go over N items.
  - eliminates one item and then calls itself with the remaining data.
- How do we write this recurrence?

Solving Recurrences: Example M1

- Suppose that we have an algorithm that at each step:
  - takes $O(N^2)$ time to go over N items.
  - eliminates one item and then calls itself with the remaining data.
- How do we write this recurrence?
- $T(N) = T(N-1) + N^2$

$$= T(N-2) + (N-1)^2 + N^2$$

$$= T(N-3) + (N-2)^2 + (N-1)2 + N^2$$

$$\ldots$$

$$= 1^2 + 2^2 + \ldots + N^2$$

$= \sum_{k=1}^{N} k^2$. How do we approximate that?

Solving Recurrences: Example M1

- We approximate $\sum_{k=1}^{N} \boldsymbol{k^2}$ using an integral:
- Clearly, $f(x) = x^2$ is a monotonically increasing function.
- So, $\sum_{k=1}^{N} k^2 \leq \int_1^{N+1} x^2 dx = \frac{(N+1)^3 - 1^3}{3}$

$$= \frac{N^3 + 2N^2 + 2N + 1 - 1}{3} = \Theta(N^3)$$

Solving Recurrences: Example M2

Solving Recurrences: Example M2

- Suppose that we have an algorithm that at each step:
  - takes Θ (lg(N)) time to go over N items.
  - eliminates one item and then calls itself with the remaining data.
- How do we write this recurrence?

Solving Recurrences: Example M2

- Suppose that we have an algorithm that at each step:
  - takes Θ (lg(N)) time to go over N items.
  - eliminates one item and then calls itself with the remaining data.
- How do we write this recurrence?
- $T(N) = T(N-1) + \lg(N)$

$$= T(N-2) + \lg(N-1) + \lg(N)$$

$$= T(N-3) + \lg(N-2) + \lg(N-1) + \lg(N)$$

…

$$= \lg(1) + \lg(2) + \ldots + \lg(N)$$

$= \sum_{k=1}^{N} lg(k)$. How do we compute that?

Solving Recurrences: Example M2

- We process $\sum_{k=1}^{N} \boldsymbol{lg(k)}$ using the fact that: $\lg(a) + \lg(b) = \lg(ab)$

- $\sum_{\mathrm{k}=1}^{\mathrm{N}} \lg(\mathrm{k}) = \lg(1) + \lg(2) + \ldots + \lg(\mathrm{N})$

$$
\begin{aligned}
&= \lg(N!) \\
&\cong \lg\left(\left(\frac{N}{e}\right)^N\right) \\
&= N \lg\left(\frac{N}{e}\right) \\
&= N \lg(N) - N \lg(e) = \Theta(N \lg(N))
\end{aligned}
$$

Solving Recurrences: Example M3

Solving Recurrences: Example M3

- Suppose that we have an algorithm that at each step:
  - takes Θ (1) time to go over N items.
  - calls itself 3 times on data of size N-1.
  - takes Θ (1) time to combine the results.
- How do we write this recurrence?

Solving Recurrences: Example M3

- Suppose that we have an algorithm that at each step:
  - takes *Θ* (1) time to go over N items.
  - calls itself 3 times on data of size N-1.
  - takes *Θ* (1) time to combine the results.
- How do we write this recurrence?
- $T(N) = 3T(N-1) + 1$ **(why use '+1' and not '+2'?)**

$$= 3^2T(N-2) + 3 + 1$$

$$= 3^3T(N-3) + 32 + 3 + 1$$

…

$$= 3^{N-1}T(1) + \underbrace{3^{N-2} + 3^{N-3} + 3^{N-4} + \cdots + 1}_{\text{finite summation}}$$

Note: *T*(1) is just a constant

Solving Recurrences: Example M3

- Suppose that we have an algorithm that at each step:
  - takes Θ (1) time to go over N items.
  - calls itself 3 times on data of size N-1.
  - takes Θ (1) time to combine the results.
- How do we write this recurrence?
- $T(N) = 3T(N-1) + 1$

$$= 3^2T(N-2) + 3 + 1$$

$$= 3^3T(N-3) + 3^2 + 3 + 1$$

…

$$= 3^{N-1}T(1) + 3^{N-2} + 3^{N-3} + 3^{N-4} + \cdots + 1$$

$$= O(3^N) + O(3^N) = O(3^N)$$

Solving Recurrences: Example M4

$T(1) = 20$

$T(N) = T\left(\frac{N}{2}\right) + N^6$ Step 1

$= T\left(\frac{N}{2^2}\right) + \left(\frac{N}{2}\right)^6 + N^6$ Step 2

$= T\left(\frac{N}{2^3}\right) + \left(\frac{N}{2^2}\right)^6 + \left(\frac{N}{2}\right)^6 + N^6$ Step 3

…

$= T\left(\frac{N}{2^i}\right) + \left(\frac{N}{2^{i-1}}\right)^6 + \cdots + \left(\frac{N}{2^2}\right)^6 + \left(\frac{N}{2}\right)^6 + N^6$ Step i

…

$= T(1) + \left(\frac{N}{2^{n-1}}\right)^6 + \cdots + \left(\frac{N}{2^2}\right)^6 + \left(\frac{N}{2}\right)^6 + N^6$ Step n

Solving Recurrences: Example M4

$= T(1) + \left(\frac{N}{2^{n-1}}\right)^6 + \cdots + \left(\frac{N}{2^2}\right)^6 + \left(\frac{N}{2}\right)^6 + N^6$ (Step n)

$= 20 + \left(\frac{N}{2^{n-1}}\right)^6 + \cdots + \left(\frac{N}{2^2}\right)^6 + \left(\frac{N}{2}\right)^6 + N^6$ Pull out N^6

$= 20 + N^6[\left(\frac{1}{2^{n-1}}\right)^6 + \cdots + \left(\frac{1}{2^2}\right)^6 + \left(\frac{1}{2}\right)^6 + 1]$ reorder

$$= 20 + N^6[1 + \left(\frac{1}{2}\right)^6 + \left(\frac{1}{2^2}\right)^6 + \cdots + \left(\frac{1}{2^{n-1}}\right)^6]$$

notice the increasing exponents. We will try to produce summation $1 + x^1 + x^2 + \ldots + x^{n-1}$

We use: $\left(\frac{1}{2^2}\right)^6 = \left(\frac{1}{2^6}\right)^2$. The advantage now is that $\left(\frac{1}{2^6}\right)$ can be used as x in the summation above

$= 20 + N^6[1 + \left(\frac{1}{2^6}\right)^1 + \left(\frac{1}{2^6}\right)^2 + \cdots + \left(\frac{1}{2^6}\right)^{n-1}]$

Since $\left(\frac{1}{2^6}\right) < 1$, $\Rightarrow 1 + x^1 + x^2 + \ldots + x^{n-1} \le 1/(1\text{-}x) \Rightarrow$

$\le 20 + N^6 * 1/[1 - \left(\frac{1}{2^6}\right)]$ Here 20 and $1/[1 - \left(\frac{1}{2^6}\right)]$ are constants and can be ignored.

$= \Theta(N^6)$ (Note that $N^6 \le T(N) \le N^6 * \text{ct}$)

Common Recurrences Review

1. Reduce the problem size by 1 in <u>constant</u> time

   $\Theta(N)$
2. Reduce the problem size by 1 in <u>linear</u> time

   $\Theta(N^2)$
3. *Halve* problem in <u>constant</u> time

   $\Theta(\lg(N))$
4. *Halve* problem in <u>linear</u> time

   $\Theta(N)$ (~2N)
5. Break the problem into *2 halves* in <u>constant</u> time

   $\Theta(N)$ (~2N)
6. Break the problem into *2 halves* in <u>linear</u> time

   $\Theta(N \lg(N))$