Recurrences: Master Theorem

CSE 3318 – Algorithms and Data Structures

Alexandra Stefan

University of Texas at Arlington

Merge sort (CLRS)

- Recurrence formula
  - Here n is the number of items being processed
  - Base case:
    - T(1) = c
    - (In the code, see for what value of n there is NO recursive call. Here when $le \geq r \Rightarrow n \leq 1$)
  - Recursive case:
    - $T(n) = 2T(n/2) + cn$
    - also ok:
    - $T(n) = 2T(n/2) + \Theta(n)$

$\boxed{T(n/2)}$

$\boxed{T(n/2)}$

```
Merge_sort(A,le,r) //n = r-le+1
   if (le>=r) return
   else
       m = floor((le+r)/2)
       Merge_sort(A,le,m);
       Merge_sort(A,m+1,r);
       Merge(A,le,m,r);
```

Mergesort

```
Merge_sort(A,le,r) //n = r-le+1
  if (le>=r) return
  else
      m = floor((le+r)/2)
      Merge_sort(A,le,m);
      Merge_sort(A,m+1,r);
      Merge(A,le,m,r);
```

- How many recursive calls are EXECUTED (called) in one function call ?
- What is the local TC?
- draw tree

Binary Search - recursive

```
/* Adapted from Sedgewick, n = right-left+1 */
int search(int A[], int left, int right, int v)
  { int m = (left+right)/2;
    if (left > right) return -1;
    if (v == A[m]) return m;
    if (left == right) return -1;
    if (v < A[m])
         return search(A, left, m-1, v);
    else
          return search(A, m+1, right, v);
  }
```

- How many recursive calls are EXECUTED (called) in one function call ?
- What is the local TC
- draw tree

Recurrences

- Examples:
  - $T(n) = 2T(n/2) + n$ (base cases: $T(0) = T(1) = c$, recurrence for TC of Mergesort)
  - $T(n) = T(n-3) + 500$ (base cases: $T(0)=T(1)=T(2) = c$)
  - $f(n) = 4f(n/5) + c$ (is c a constant here?) (base cases: $f(0)=f(1) = 6$)
  - $S(n) = S(n/3) + n^2 \lg n$ (base cases: $S(0)=S(1) = 20$)

- Same meaning: $n/cn/\Theta(n)$
  - $T(n) = 2T(n/2) + n$
  - $T(n) = 2T(n/2) + cn$
  - $T(n) = 2T(n/2) + \Theta(n)$

- Used to
  - Describe the time complexity of recursive algorithms.
  - Compute the number of nodes in certain trees.
  - Another way to express the time complexity of non-recursive algorithms (e.g. insertion sort).
    (Identify the subproblems and the local work)

- Methods for solving recurrences:
  - trees (recurrence tree)
  - ***Master Theorem***
  - expansion of the recurrence into a summation by using repeated substitution

$T(1) = c$

$T(N) = 4T(N/5)+c$

Using the Master Theorem

- Review: know how to apply a theorem
  - check if the conditions are met
  - apply it

- Be able to write the recurrence formula for a piece of code.

- Given a recurrence, decide if Master Theorem can be used to solve it or not

- Applying Master Theorem
  - Identify which case of the theorem to use
  - check the condition(s)
  - solve recurrence (if conditions were met)

Master Theorem – simplified versions M1 and M2

M1 (Master Theorem easy 1): Let ***a≥1*** and ***b>1***, and let $\boldsymbol{T(n)}$ be defined on the nonnegative integers by the recurrence:

$\boldsymbol{T(n) = aT\left(\frac{n}{b}\right) + n^p}$, where we interpret $\boldsymbol{\frac{n}{b}}$ to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.

1. $If\ \log_b a < p\ then\ T(n) = \Theta(n^p)$.
2. $If\ \log_b a > p\ then\ T(n) = \Theta(n^{\log_b a})$
3. $If\ \log_b a == p\ then\ T(n) = \Theta(n^p lgn)$

M2 (Master Theorem easy 2): Let ***a≥1*** and ***b>1***, and let $\boldsymbol{T(n)}$ be defined on the nonnegative integers by the recurrence:

$\boldsymbol{T(n) = aT\left(\frac{n}{b}\right) + n^p(lgn)^k}$ $,where$ $\boldsymbol{\log_b a == p}$ and $\boldsymbol{k \geq 0}$, where we interpret $\boldsymbol{\frac{n}{b}}$ to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$,

then $\boldsymbol{T(n) = \Theta\left(n^p(lgn)^{k+1}\right)}$

M3 – Extension of M2 for k<0 – **not required**.

- M3a) if k>-1 , then T(n) = $\Theta(n^p (\log n)^{k+1})$.
- M3b) if k==-1, then T(n) = $\Theta(n^p \log\log n)$
- M3c) if k<-1, then T(n) = $\Theta(n^p)$.

- Check the notes handwritten in class to see how to apply these theorems.
- The Master Theorem from wikipedia and other sources cover more cases, but are more difficult to understand https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)

Examples of equations that do not match the Master Theorem requirements

- Give examples of recurrences that cannot be solved with Master Thm (see also https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms))
- T(n) = 0.5 T(n/2) +n^2 (bad a)
- S(n) = 2S(3n) + n (bad b: b = 3n = n/b => 3 = 1/b => b= 1/3 not > 1)
- U(n) = U(n-4) + 5 (bad smaller pb size: n-b not n/b)
- T(n) = 3T(n/2) - n (bad local time complexity: negative)
- How about:
  - $T(n) = 5T(2n/3) + n^4$ a = 5>=1, b = 3/2 > 1, we can apply master thm

Solve recurrences

Math review and practice form: https://forms.office.com/r/F8At7KsgmJ

a) $T(n) = 9T(n/3) + n^4$

b) $T(n) = 8T(n/2) + n^2$

c) $T(n) = 16T(n/2) + n^4$

d) $T(n) = 9T(n/3) + n^2 \lg n^3$

e) $T(n) = 8T(n/2) + \lg n$

f) $T(n) = 2T(n/7) + \lg n$

g) $T(n) = 2T(n/3) + T(n/2) + n$ (**)

Common Recurrences Review

1. *Halve* problem in <u>constant</u> time :

   $T(n) = T(n/2) + c$ $\Theta(\lg(n))$

2. *Halve* problem in <u>linear</u> time :

   $T(n) = T(n/2) + n$ $\Theta(n)$ (~2n)

3. Break (and put back together) the problem into *2 halves* in <u>constant</u> time:

   $T(n) = 2T(n/2) + c$ $\Theta(n)$ (~2n)

4. Break (and put back together) the problem into *2 halves* in <u>linear</u> time:

   $T(n) = 2T(n/2) + n$ $\Theta(n \lg(n))$

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

   $T(n) = T(n-1) + c$ $\Theta(n)$

6. Reduce the problem size by 1 in <u>linear</u> time:

   $T(n) = T(n-1) + n$ $\Theta(n^2)$

Come back later to this slide and:

- solve the recurrences (with tree or Master Thm)
- give examples of algorithms that have these recurrences. Think:
  - n is the input size (e.g. array size)
  - + ?? is the local work (or local TC)
    (it excludes the work/TC of recursive calls)

Given a Recursive function (code) => Write the Recurrence

```
int foo(int N){
  int a,b,c;
  if(N<=3) return 1500; // Note N<=3
  a = 2*foo(N-1);
 // a = foo(N-1)+foo(N-1);
  printf("A");
  b = foo(N/2);
  c = foo(N-1);
  return a+b+c;
}
```

Identify
- base case
- recursive case

The recurrence formula captures the number of times recursive calls ACTUALLY EXECUTE as we run the instructions in the function.

Base case: T(__) = __________

Recursive case: T(__) = ______________________

T(N) gives us the Time Complexity for foo(N). We need to solve it (find the closed form)

Code => Recurrence => Θ

The recursive case of the recurrence formula captures the number of times the recursive call ACTUALLY EXECUTES as you run the instructions in the function.

```
void bar(int N){
 int i,k,t;
 if(N<=1) return;
 bar(N/5);
 for(i=1;i<=5;i++){
   bar(N/5);
 }
 for(i=1;i<=N;i++){
   for(k=N;k>=1;k--)
     for(t=2;t<2*N;t=t+2)
       printf("B");
 }
 bar(N/5);
}
```

Base case: T(__) = __________

Recursive case: T(__) = ____________________

Solve T(N)

Compare

```
void foo1(int N){
     if (N <= 1) return;
     for(int i=1; i<=N; i++){
        foo1(N-1);
     }
}

T(0)=T(1) = c
T(N) = N*T(N-1) + cN
```

```
void foo2(int N){
     if (N <= 5) return;
     for(int i=1; i<=N; i++){
        printf("A");
     }
     foo2(N-1); //outside of the loop
}

T(N) = c for all 0≤N≤5   (BaseCase(s))
T(N) = T(N-1) + cN      (Recursive Case)
```

```
int foo3(int N){
    if (N <= 20) return 500;
    for(int i=1; i<=N; i++){
       return foo3(N-1);
// No loop. Returns after the first iteration.
    }
}
```

T(N) = c for all 0≤N≤20   Do not confuse what the function returns with its time complexity. For the base case, c is not 500. At most, c is 2 (from the 2 instructions: one comparison, *N<=20*, and one return, *return 500*)

T(N) = T(N-1) + **c**

The recursive case of the recurrence formula captures the number of times the recursive call **ACTUALLY EXECUTES** as you run the instructions in the function. E.g. pay attention to code: 2*foo(N/3) vs foo(N/3) + foo(N/3)

Code =>recurrence

```
int search(int A[], int L, int R, int v){
   int m = (L+R)/2;
   if (L > R) return -1;
   if (v == A[m]) return m;
   if (L == R) return -1;
   if (v < A[m]) return search(A,L,m-1,v);
   else          return search(A,m+1,R,v);
}
(Use:  N = R-L+1)
Here, for the same value of N, the behavior depends also on data in A and val.
Best case T(N) = c => search is Θ(1) in best case
Worst case: T(N) = T(N/2) + c  => T(N) = Θ(lg(N)) => search is Θ(lg(N))in worst case
⇒ We will report in general: search is O(lg(N))
```

Code => recurrence

```
int weird(int A[], int N){
   if (N<=4) return 100;
   if (N%5==0) return weird(A,N/5);
   else        return weird(A,N-4)+weird(A, N-4);
}
```

Here, the behavior depends on N so we can explicitly capture that in the recurrence formulas:

Base case(s): $T(N) = c$ for all $0 \le N \le 4$ (BC)

Recursive case(s):

$T(N) = T(N/5)+c$ for all $N>4$ that are multiples of 5   (RC1)

$T(N) = 2*T(N-4) + c$ for all other N   (RC2)

For any N, in order to solve, we need to go through a mix of the 2 recursive cases => cannot easily solve. => try to find lower and upper bounds.

Note that RC1 has the best behavior: only one recurrence and smallest subproblem size (i.e. N/5) => use this for a lower bound =>

$T_{lower}(N) = T(N/5)+c = \Theta(\log_5 N)$, (and $T(N) \ge T_{lower}(N)$) => **$T(N) = \Omega(\log_5 N)$**

Note that RC2 has the worst behavior: 2 recurrences and both of larger subproblem size (i.e. N-4) => use this for an upper bound =>

$T_{upper}(N) = 2*T(N-4)+c = \Theta(2^{N/4})$, (and $T(N) \le T_{upper}(N) = \Theta(2^{N/4})$) => **$T(N) = O(2^{N/4})$**

We have **Ω and O** for T(N), but we cannot compute **Θ** for it.

Recurrence => Code

- Give a piece of code/pseudocode for which the time complexity recursive formula is:
  - T(1) = c   and
  - T(N) = N*T(N/2) + cN

Recurrence => Code
Answer

- Give a piece of code/pseudocode for which the time complexity recursive formula is:
  - $T(1) = c$ and
  - $T(N) = N*T(N/2) + cN$

```
void foo(int N){
    if (N <= 1) return;
    for(int i=1; i<=N; i++)
        foo(N/2);
}
```