COMPSCI 130
Design and Analysis of Algorithms

3 – Divide and Conquer Algorithms
Mergesort
Recurrence Relations

Divide and Conquer

- Split problem into multiple smaller sub-problems
- Solve the sub-problems recursively
- Recombine solutions afterwards
- When splitting/recombination can be done efficiently, this approach is a winner

Example

Search for a value in a sorted list.
Obvious approach:

```plaintext
search(x, k, n) // x is a sorted list with n elements
for i ← 1 to n
  if x(i) = k return i
return NOTFOUND
```

Complexity: O(N)

A Better Way

Search for a value in a sorted list.

Binary Search

```plaintext
binary_search(x, k) // find element k in list x containing n elements
if x is empty
  return NOTFOUND
pivot ← ⌈n/2⌉ // look at element halfway through list
if x[pivot] = k
  return pivot // if found, return
else if k < x[pivot]
  return binary_search(x[1 : pivot-1], k)
else
  return binary_search(x[pivot+1 : n], k)
```

Example – Binary Search

Search for a value in a sorted list.
Example: search for 11 in the list 1-15

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Analysis of Binary Search

Compare with pivot
New pivot
Recurrence: T(N) = T(N/2) + O(1)

Worst case: element not found

Complexity: # of times we split the list in two before getting to length 1 = log N
Mergesort

- Divide and Conquer algorithm for sorting
  - Split input list in half
  - Sort the halves
  - Merge the sorted lists

merge_sort(x)

n ← length(x)
if n = 1 return x
left ← merge_sort(x[1 : ⌈n/2⌉])
right ← merge_sort(x[⌈n/2⌉+1 : n])
return merge(left, right)

merge(a, b)

// treat a, b as stacks
x ← empty list
loop
  if a is empty
    append b to x, return x
  else if b is empty
    append a to x, return x
  else if top(a) < top(b)
    append pop(a) to x
  else append pop(b) to x

Mergesort Illustrated

Analysis of Mergesort

Split = O(1)
Merge = O(N)
Recursion = 2 T(N/2)

Recurrence relation is T(N) = 2 T(N/2) + O(N)

Complexity: ?

Solving Recurrence Relations

- Tree method
- Substitution method: Guess, then prove
- Master Method: Pattern match
- Some rules of thumb
  - Ignore floor/ceiling
  - Base cases usually O(1)

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

Tree Method

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

Now just do the math:
O(log n) levels at O(n) cost each = O(n log n)

Substitution Method

- Guess — a good way to start is to list out values for small n
- Prove — using induction
- E.g., for T(n) = 2T(n/2) + n
  - Guess T(n) = O(n log n)
  - Base cases:
    - T(1) = 1:
      - T(1) = 2 T(1) + 2 ≤ 2 log 2, for c = 2
    - Assume true for n/2, show true for n
      - T(n) = 2 T(n/2) + n
        ≤ 2 T(n/2) log (n/2) + n
        = cn log n – cn log 2 + n
        ≤ cn log n

Master Method

- Often the quickest way – if your recurrence fits the pattern.
  If:  \( T(n) = aT\left(\lceil \frac{n}{b} \rceil \right) + O(n^d), \ a > 0, b > 1, d \geq 0 \)
  Then:
  \[
  T(n) = O(n^d) \quad \text{if} \ d > \log_b a \\
  = O(n^d \log n) \quad \text{if} \ d = \log_b a \\
  = O(n^{\log_b a}) \quad \text{if} \ d < \log_b a
  \]

Sample Problems

\( T(n) = T(\sqrt{n}) + 1 \)  
O(1) cost at each level  
Height of tree is \( k \) such that  
\[ \lfloor \sqrt{n} \rfloor = n^{k/2} \]

With the floor, this equates to
\[ n^{k/2} < 2 \]
\[ n < 2^{2^k} \]
\[ \log n < 2^k \]
\[ \log \log n < k \]
Complexity: \( O(\log \log n) \)

Sample Problem: \( T(n) = 3T(n/2) + n^2 \)

\[
T(n) = 3T(n/2) + n^2 \\
T(n) \downarrow \\
T(n/2) \downarrow \\
T(n/4) \downarrow \\
\ldots \\
T(1)
\]

\[
\begin{align*}
T(n) & = 3T(n/2) + n^2 \\
& = 3 \left( 3T(n/4) + n/4 \right) + n^2 \\
& = 9T(n/4) + 3n/4 + n^2 \\
& \ldots \\
& = 9^n n^2/4 + \ldots + 3n/4 + n
\end{align*}
\]

Sample Problem: \( T(n) = T(n/4) + T(3n/4) + n \)

\[
T(n) = T(n/4) + T(3n/4) + n \\
\quad \text{Start by guessing; using the tree method, we note that the height of the tree is } \log_{4/3} n; \\
\quad \text{At each level of the tree the maximum cost is } n \text{ (less as the } n/4 \text{ branches end in leaves sooner than the } 3n/4 \text{ branches);} \\
\quad \text{Guess } O(n \log n)
\]

Sample Problem: \( T(n) = T(n/4) + T(3n/4) + n \) (con’t)

\[
T(n) = T(n/4) + T(3n/4) + n \\
\quad \text{Check guess of } O(n \log n) \text{ using induction:} \\
\quad \text{Assume true for } 2 \ldots n-1. \ \text{Then,} \\
\quad T(n) \leq c(n/4 \log n/4) + c(3n/4 \log 3n/4) + n \\
\quad T(n) \leq c(n/4 \log n - \log 4 + 3 \log n - 3 \log 4) + n \\
\quad \text{let } d = (3 \log 4 - 3 \log 3)/4; \ \text{note: } d > 0 \\
\quad T(n) \leq c(n \log n + (1 - c)d) n \\
\quad T(n) < cn \log n
\]
T(n) = T(n/4) + T(3n/4) + n (con’t)

T(n) = T(n/4) + T(3n/4) + n
Still need to check base cases:
- T(1) = 1
- T(2) = T(1) + T(1) + 2 = 4
- T(3) = T(1) + T(2) + 3 = 8
- ...
We can choose c to ensure T(n) < c n log n for n = 2, 3, ..., 7 and c > 1/d from previous step and we’re done.