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:

search(x, k, n) // x is a sorted list with n elements
for i = 1 to n
if x(i) = k return i
else return NOT FOUND

Complexity: O(N)

A Better Way

Search for a value in a sorted list.
Binary Search

binary_search(x, k, n)
if x is empty return NOT FOUND
pivot = [n/2]
if x[pivot] = k return pivot
elseif k < x[pivot] return binary_search(x(1 : pivot-1), k, pivot-1)
else return binary_search(x(pivot+1 : n), k, n-pivot)

Example – Binary Search

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

Analysis of Binary Search

Compare with pivot
Return or choose new pivot
O(1)

Q(N/2) elements

Worst case: element not found
Recurrence: T(N) = T(N/2) + O(1)
Complexity: \log N
Mergesort

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

```python
def merge_sort(s):
    n = length(s)
    if n == 1:
        return
    left = merge_sort(s[:n//2])
    right = merge_sort(s[n//2+1:n])
    return merge(left, right)

def merge(a, b):
    x = empty list
    if a is empty:
        append b to x
    else if b is empty:
        append a to x
    else:
        if top(a) < top(b):
            append pop(a) to x
        else:
            append pop(b) to x
```

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)

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 = 4
      - c ≥ 2 log 2, for c = 2
  - Assume true for n/2, show true for n
    T(n) = 2 T(n/2) + n
    ≤ 2 c(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: \( T(n) = aT\left(\frac{n}{b}\right) + O(n^d) \), \( a > 0, b > 1, d \geq 0 \)
  
  Then:
  
  \[
  T(n) = O(n^d) \quad \text{if } d > \log_b a
  \]
  
  \[
  = O(n^d \log n) \quad \text{if } d = \log_b a
  \]
  
  \[
  = O(n^{\log_2 d}) \quad \text{if } d < \log_b a
  \]

Sample Problems

\[ T(n) = T(\sqrt{n}) + 1 \]

- \( O(1) \) cost at each level
  
  Height of tree is \( k \) such that
  
  \[ \sqrt{n} = n^{(1/2^k)} = 1 \]
  
  \[ \text{Notation: } \sqrt[n]{x} = x^{1/n} \]
  
  With the floor, this equates to
  
  \[ n^{(1/2^k)} < 2 \]
  
  \[ \log n < 2^{k} \]
  
  \[ \log \log n < k \]
  
  Complexity: \( O(\log \log n) \)

Sample Problems

\[ T(n) = T(\log n) + 1 \]

- \( O(1) \) at each level
  
  \( \log^* n \) levels [\( \log^* \) is the “iterated logarithm”]
  
  Effectively constant time:
  
  \( \log^* 2^{5536} = 5 \)

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

\[ T(n) = 3T(n/2) + n^2 \]

Tree is height \( \log n \);

- Cost at level \( i \) is \( (\frac{1}{4})^i n^2 \);

- Summing costs,
  
  \[ T(n) = \sum_{i=0}^{\log n} (\frac{1}{4})^i n^2 \]

Using identity \( \sum_{i=0}^{\infty} x^i = 1/(1-x) \) for \( x < 1 \),

\[ T(n) \leq n^2 \sum_{i=0}^{\infty} (\frac{1}{4})^i = 4 n^2 = O(n^2) \]

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

- \( T(n) = T(n/4) + T(3n/4) + n \)

  - Start by guessing; using the tree method, we note that the height of the tree is \( \log_{4/3} n \);

  - At each level of the tree the maximum cost is \( n \) (less as the \( n/4 \) branches end in leaves sooner than the \( 3n/4 \) branches);

  - Guess \( O(n \log n) \)
\[ T(n) = T(n/4) + T(3n/4) + n \] (con’t)

\[ T(n) = T(n/4) + T(3n/4) + n \]

Check guess of \( O(n \log n) \) using induction:

Assume true for \( 1 \ldots n-1 \). Then,

\[ T(n) \leq c (n/4) \log (n/4) + c (3n/4) \log (3n/4) + n \]
\[ T(n) \leq cn/4 \{ \log n - \log 4 + 3 \log n + 3 \log 3 - 3 \log 4 \} + n \]
\[ T(n) \leq cn/4 \{ 4 \log n - 4 \log 4 + 3 \log 3 \} + n \]
\[ T(n) < cn \log n - cn [4 \log 4 - 3 \log 3]/4 + n \]

let \( d = [4 \log 4 - 3 \log 3]/4 \), note: \( d > 0 \)

\[ T(n) < cn \log n + (1 - c d) n \]
choose \( c > 1/d \)
\[ T(n) < cn \log n \]

---

\[ T(n) = T(n/4) + T(3n/4) + n \] (con’t)

\[ T(n) = T(n/4) + T(3n/4) + n \]

Still need to check base cases:

- \( T(1) = 1 \)
- \( T(2) = T(1) + T(1) + 2 = 4 \)
- \( T(3) = T(1) + T(2) + 3 = 8 \)
- \( T(4) = T(1) + T(3) + 4 = 13 \)

We can choose \( c \) to ensure \( T(n) < cn \log n \) for \( n = 1 \ldots 4 \), and \( c > 1/d \) from previous step and we’re done.