Divide and Conquer: Recursion Trees

Madhavan Mukund

`https://www.cmi.ac.in/~madhavan`

Programming, Data Structures and Algorithms using Python

Week 8

Solving recurrences

- Divide and conquer involves breaking up a problem into disjoint subproblems and combining the solutions efficiently

Solving recurrences

- Divide and conquer involves breaking up a problem into disjoint subproblems and combining the solutions efficiently
- Complexity $T(n)$ is expressed as a recurrence

Solving recurrences

- Divide and conquer involves breaking up a problem into disjoint subproblems and combining the solutions efficiently
- Complexity $T(n)$ is expressed as a recurrence
- For searching and sorting, we solved simple recurrences by repeated substitution
  - Binary search: $T(n) = T(n/2) + 1$, $T(n)$ is $O(\log n)$
  - Merge sort: $T(n) = 2T(n/2) + n$, $T(n)$ is $O(n \log n)$

Solving recurrences

- Divide and conquer involves breaking up a problem into disjoint subproblems and combining the solutions efficiently
- Complexity $T(n)$ is expressed as a recurrence
- For searching and sorting, we solved simple recurrences by repeated substitution
  - Binary search: $T(n) = T(n/2) + 1$, $T(n)$ is $O(\log n)$
  - Merge sort: $T(n) = 2T(n/2) + n$, $T(n)$ is $O(n \log n)$
- For integer multiplication, the analysis became more complicated
  - Naive divide and conquer: $T(n) = 4T(n/2) + n$, $T(n)$ is $O(n^2)$
  - Karatsuba's algorithm: $T(n) = 3T(n/2) + n$, $T(n)$ is $O(n^{\log 3})$

Solving recurrences

- Divide and conquer involves breaking up a problem into disjoint subproblems and combining the solutions efficiently
- Complexity $T(n)$ is expressed as a recurrence
- For searching and sorting, we solved simple recurrences by repeated substitution
  - Binary search: $T(n) = T(n/2) + 1$, $T(n)$ is $O(\log n)$
  - Merge sort: $T(n) = 2T(n/2) + n$, $T(n)$ is $O(n \log n)$
- For integer multiplication, the analysis became more complicated
  - Naive divide and conquer: $T(n) = 4T(n/2) + n$, $T(n)$ is $O(n^2)$
  - Karatsuba's algorithm: $T(n) = 3T(n/2) + n$, $T(n)$ is $O(n^{\log 3})$
- Is there a uniform way to compute the asymptotic expression for $T(n)$?

Recursion trees

- Recursion tree Rooted tree with one node for each recursive subproblem

Recursion trees

- Recursion tree Rooted tree with one node for each recursive subproblem
- Value of each node is time spent on that subproblem excluding recursive calls

Recursion trees

- Recursion tree Rooted tree with one node for each recursive subproblem
- Value of each node is time spent on that subproblem excluding recursive calls
- Concretely, on an input of size n

Recursion trees

- Recursion tree Rooted tree with one node for each recursive subproblem
- Value of each node is time spent on that subproblem excluding recursive calls
- Concretely, on an input of size n
  - $f(n)$ is the time spent on non-recursive work

Recursion trees

- Recursion tree Rooted tree with one node for each recursive subproblem
- Value of each node is time spent on that subproblem excluding recursive calls
- Concretely, on an input of size n
  - $f(n)$ is the time spent on non-recursive work
  - r is the number of recursive calls

Recursion trees

- Recursion tree Rooted tree with one node for each recursive subproblem
- Value of each node is time spent on that subproblem excluding recursive calls
- Concretely, on an input of size n
  - $f(n)$ is the time spent on non-recursive work
  - r is the number of recursive calls
  - Each recursive call works on a subproblem of size n/c

Recursion trees

- Recursion tree Rooted tree with one node for each recursive subproblem
- Value of each node is time spent on that subproblem excluding recursive calls
- Concretely, on an input of size n
  - $f(n)$ is the time spent on non-recursive work
  - r is the number of recursive calls
  - Each recursive call works on a subproblem of size n/c
- Resulting recurrence: $T(n) = rT(n/c) + f(n)$

Recursion trees

- Recursion tree Rooted tree with one node for each recursive subproblem
- Value of each node is time spent on that subproblem excluding recursive calls
- Concretely, on an input of size n
  - $f(n)$ is the time spent on non-recursive work
  - r is the number of recursive calls
  - Each recursive call works on a subproblem of size n/c
- Resulting recurrence: $T(n) = rT(n/c) + f(n)$
- Root of recursion tree for $T(n)$ has value $f(n)$

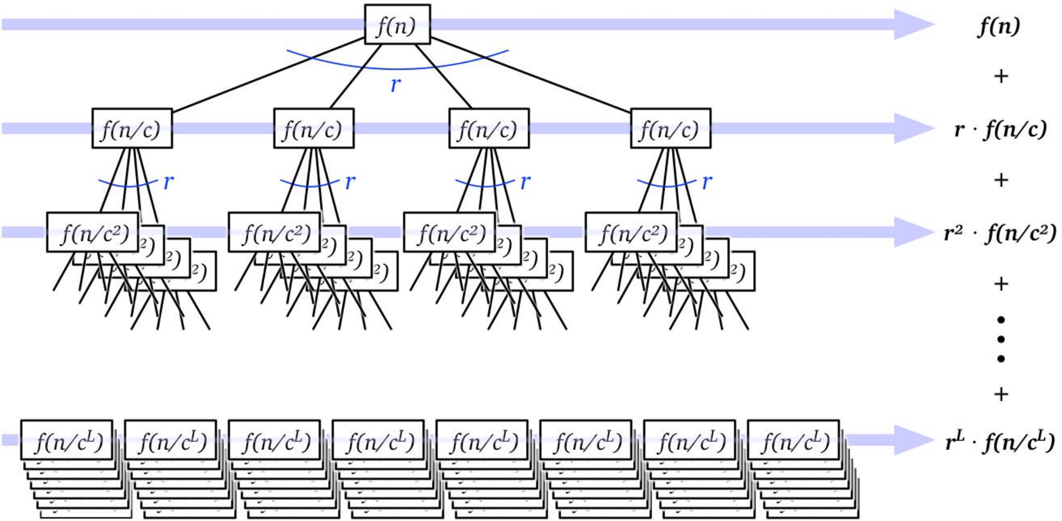
Recursion trees

- Recursion tree Rooted tree with one node for each recursive subproblem
- Value of each node is time spent on that subproblem excluding recursive calls
- Concretely, on an input of size n
  - $f(n)$ is the time spent on non-recursive work
  - r is the number of recursive calls
  - Each recursive call works on a subproblem of size n/c
- Resulting recurrence: $T(n) = rT(n/c) + f(n)$
- Root of recursion tree for $T(n)$ has value $f(n)$
- Root has r children, each (recursively) the root of a tree for $T(n/c)$

Recursion trees

- Recursion tree Rooted tree with one node for each recursive subproblem
- Value of each node is time spent on that subproblem excluding recursive calls
- Concretely, on an input of size n
  - $f(n)$ is the time spent on non-recursive work
  - r is the number of recursive calls
  - Each recursive call works on a subproblem of size n/c
- Resulting recurrence: $T(n) = rT(n/c) + f(n)$
- Root of recursion tree for $T(n)$ has value $f(n)$
- Root has r children, each (recursively) the root of a tree for $T(n/c)$
- Each node at level d has value $f(n/c^d)$
  - Assume, for simplicity, that n was a power of c

Recursion tree for $T(n) = rT(n/c) + f(n)$



Recursion trees

- Leaves correspond to the base case $T(1)$
  - Safe to assume $T(1) = 1$, asymptotic complexity ignores constants

Recursion trees

- Leaves correspond to the base case $T(1)$
  - Safe to assume $T(1) = 1$, asymptotic complexity ignores constants
- Level i has r^i nodes, each with value $f(n/c^i)$

Recursion trees

- Leaves correspond to the base case $T(1)$
  - Safe to assume $T(1) = 1$, asymptotic complexity ignores constants
- Level i has r^i nodes, each with value $f(n/c^i)$
- Tree has L levels, $L = \log_c n$

Recursion trees

- Leaves correspond to the base case $T(1)$
  - Safe to assume $T(1) = 1$, asymptotic complexity ignores constants
- Level i has r^i nodes, each with value $f(n/c^i)$
- Tree has L levels, $L = \log_c n$
- Total cost is $T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i)$

Recursion trees

- Leaves correspond to the base case $T(1)$
  - Safe to assume $T(1) = 1$, asymptotic complexity ignores constants
- Level i has r^i nodes, each with value $f(n/c^i)$
- Tree has L levels, $L = \log_c n$
- Total cost is $T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i)$
- Number of leaves is r^L
  - Last term in the level by level sum is $r^L \cdot f(1) = r^{\log_c n} \cdot 1 = n^{\log_c r}$
  - Recall that $a^{\log_b c} = c^{\log_b a}$

Recursion trees

- Tree has $\log_c n$ levels, last level has cost is $n^{\log_c r}$
- Total cost is $T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i)$

Recursion trees

- Tree has $\log_c n$ levels, last level has cost is $n^{\log_c r}$
- Total cost is $T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i)$
- Think of the total cost as a series. Three common cases

Recursion trees

- Tree has $\log_c n$ levels, last level has cost is $n^{\log_c r}$
- Total cost is $T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i)$
- Think of the total cost as a series. Three common cases
- Decreasing Each term is a constant factor smaller than previous term
  - Root dominates the sum, $T(n) = O(f(n))$

Recursion trees

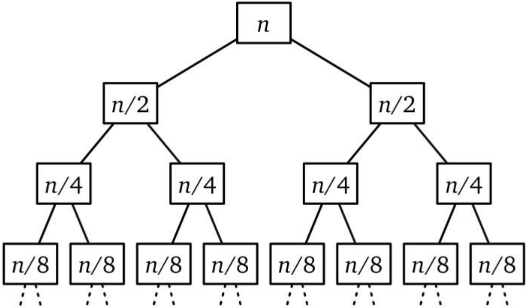
- Tree has $\log_c n$ levels, last level has cost is $n^{\log_c r}$
- Total cost is $T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i)$
- Think of the total cost as a series. Three common cases
- Decreasing Each term is a constant factor smaller than previous term
  - Root dominates the sum, $T(n) = O(f(n))$
- Equal All terms in the series are equal
  - $T(n) = O(f(n) \cdot L) = O(f(n) \log n)$ — $\log_c n$ is asymptotically same as $\log n$

Recursion trees

- Tree has $\log_c n$ levels, last level has cost is $n^{\log_c r}$
- Total cost is $T(n) = \sum_{i=0}^{L} r^i \cdot f(n/c^i)$
- Think of the total cost as a series. Three common cases
- Decreasing Each term is a constant factor smaller than previous term
  - Root dominates the sum, $T(n) = O(f(n))$
- Equal All terms in the series are equal
  - $T(n) = O(f(n) \cdot L) = O(f(n) \log n)$ — $\log_c n$ is asymptotically same as $\log n$
- Increasing Series grows exponentially, each term a constant factor larger than previous term
  - Leaves dominate the sum, $T(n) = O(n^{\log_c r})$

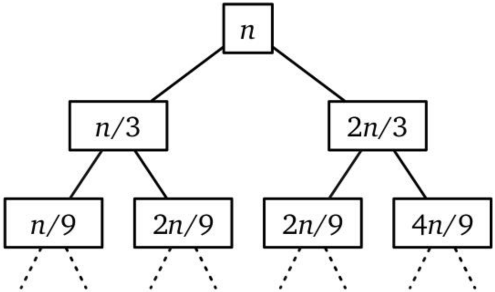
Examples

- Merge sort
  - $T(n) = 2T(n/2) + n$
  - Series is equal, $T(n)$ is $O(n \log n)$



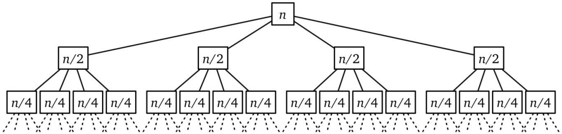
Examples

- Merge sort
  - $T(n) = 2T(n/2) + n$
  - Series is equal, $T(n)$ is $O(n \log n)$
- Quick sort with pivot always in the middle third of values
  - $T(n) = T(n/3) + T(2n/3) + n$
  - Unequal partitions, allow “holes”
  - Depth is $\log_{\frac{3}{2}} n = O(\log n)$
  - Series is equal, $T(n)$ is $O(n \log n)$



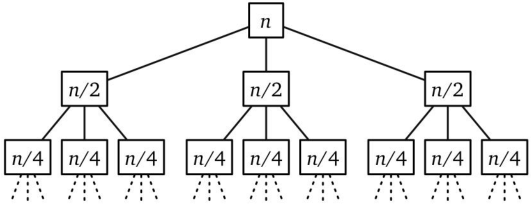
Examples

- Merge sort
  - $T(n) = 2T(n/2) + n$
  - Series is equal, $T(n)$ is $O(n \log n)$
- Quick sort with pivot always in the middle third of values
  - $T(n) = T(n/3) + T(2n/3) + n$
  - Unequal partitions, allow "holes"
  - Depth is $\log_{\frac{3}{2}} n = O(\log n)$
  - Series is equal, $T(n)$ is $O(n \log n)$
- Naive integer multiplication, exponential, $T(n) = n^{\log_2 4} = n^2$



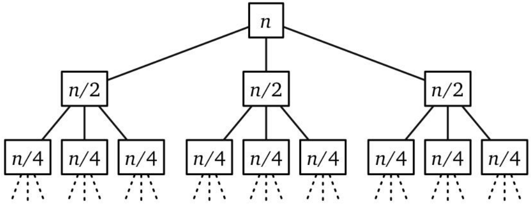
Examples

- Merge sort
  - $T(n) = 2T(n/2) + n$
  - Series is equal, $T(n)$ is $O(n \log n)$
- Quick sort with pivot always in the middle third of values
  - $T(n) = T(n/3) + T(2n/3) + n$
  - Unequal partitions, allow "holes"
  - Depth is $\log_{\frac{3}{2}} n = O(\log n)$
  - Series is equal, $T(n)$ is $O(n \log n)$
- Naive integer multiplication, exponential, $T(n) = n^{\log_2 4} = n^2$
- Karatsuba, exponential, $T(n) = n^{\log_2 3}$



Examples

- Merge sort
  - $T(n) = 2T(n/2) + n$
  - Series is equal, $T(n)$ is $O(n \log n)$
- Quick sort with pivot always in the middle third of values
  - $T(n) = T(n/3) + T(2n/3) + n$
  - Unequal partitions, allow "holes"
  - Depth is $\log_{\frac{3}{2}} n = O(\log n)$
  - Series is equal, $T(n)$ is $O(n \log n)$
- Naive integer multiplication, exponential, $T(n) = n^{\log_2 4} = n^2$
- Karatsuba, exponential, $T(n) = n^{\log_2 3}$



- Acknowledgment
  Illustrations from
  Algorithms by Jeff Erickson,
  `https://jeffe.cs.illinois.edu/teaching/algorithms/`