

1 Runtime Analysis

Find out the Big-Θ bound of the following functions:

1. public void f1(int N) {
   if (N <= 0) {
      return;
   }
   System.out.println("working");
   return f1(N-1) + f1(N-1) + f1(N-1);
}

Each node has a constant runtime as Θ(1), so we could count the total amount of nodes and it will be our answer.

Count Nodes.

\[ 1 + 3 + 9 + \ldots + 3^{N-1} < 2 \times 3^{N-1} \]

- Remove the constants
- We have approx \(3^N\) nodes.
- For each node the runtime is \(Θ(1)\).
- Result: \(Θ(1 \times 3^N) = Θ(3^N)\)

- When \(N = 3\), the last layer has height = 3-1 = 2, The last layer has total of \(3^2 = 9\) nodes.
  \(\text{diff: } 9 - 4 = 5\)

- When \(N = 4\), \(h = 3\), last-layer = 27
  \((1+3+9) < 27\) \(\text{diff: } 27 - 13 = 14\)

- \(N = 5\), \(h = 4\), last-layer = 81
  \(\text{diff: } 81 - 41\)

- Conclusion:
  \((1+3+9+\ldots+3^{N-2}) < 3^N - 1\)

Therefore:
\[(1+3+9+\ldots+3^{N-2}) + 3^{N-1} < 2 \times 3^{N-1}\]
2. public void f2(int N, int M) {
    int count = 0;
    for(int i = N/2; i < N; i++) {
        for(int j = 1; j <= N; j = 2 * j) {
            for(int k = 1; k <= N; k = k * 2) {
                count++;
            }
            execute logn times
        }execute n/2 times = \Theta(n)
    }
}

- How many $*2$ operation will get to $N$ so the for loop stops?
  
  if we execute $h$ time and we stop,
  
  $N = 1*2*2*\ldots*2$
  
  = $2^h$
  
  $h = \log N \leq$ so we have execute $\log n$ times.

- Result: they're nested.

  $\Theta(n \log n \cdot \log n)$
  
  = $\Theta(n \log^2 n)$
3. public void f3(int N, int M) {
    if (N <= 0) {
        return;
    }
    for(int i = 0; i < M; i++) {
        System.out.println("working");
    }
    return f3(N-1, M) + f3(N-1, M) + f3(N-1, M);
}

The only difference between ① and ② is that each node execute a for loop with $\Theta(M)$. The runtime is the same for all nodes, so we keep the same total amount of nodes from ①: $3^N$

Res:

$\Theta (3^N \cdot M) = \Theta (M3^N)$
4. public void f4(int N) {
    if (N <= 0) {
        return;
    }
    for(int i = 0; i < N; i++) {
        System.out.println("working");
    }
    return f4(N-1) + f4(N-1);
}

This time, different from section question 3, the runtime of each level is not the same. Can't use counting method as

1. Since nodes have different

<table>
<thead>
<tr>
<th>Height</th>
<th>No. of Nodes</th>
<th>Each Node Runtime</th>
<th>Runtime by Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>N</td>
<td>1 - N = N</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>N-1</td>
<td>2(N-1)</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>N-2</td>
<td>4(N-2)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>N-1</td>
<td>2^N-1</td>
<td>1</td>
<td>2^N-1 . 1</td>
</tr>
</tbody>
</table>

Sum of runtime by each level:

\[ N + 2(N-1) + 4(N-2) + 8(N-3) + \ldots + 2^{N-1}(N-(N-1)) \]

\[ = N + 2N-2 + 4N-8 + 8N-24 + \ldots + 2^{N-1} \]

\[ \leq \left( 1 + 2 + 4 + 8 + \ldots + 2^{N-1} \right) N - (2+8+24+\ldots) \]

\[ = (2 - 2^{N-1}) N \]

\[ = 2^N \]

\[ \Theta(2^N) \]

Ignore the constants. So the constants could be ignored.
2 B-Trees

1. Construct a 2-3 B-Tree with keys in order: 1, 6, 7, 2, 4, 5, 30, 25.
2. Construct a 2-3-4 B-Tree with the same keys above.

\[1, 6, 7, 2, 4, 5, 30, 25\]