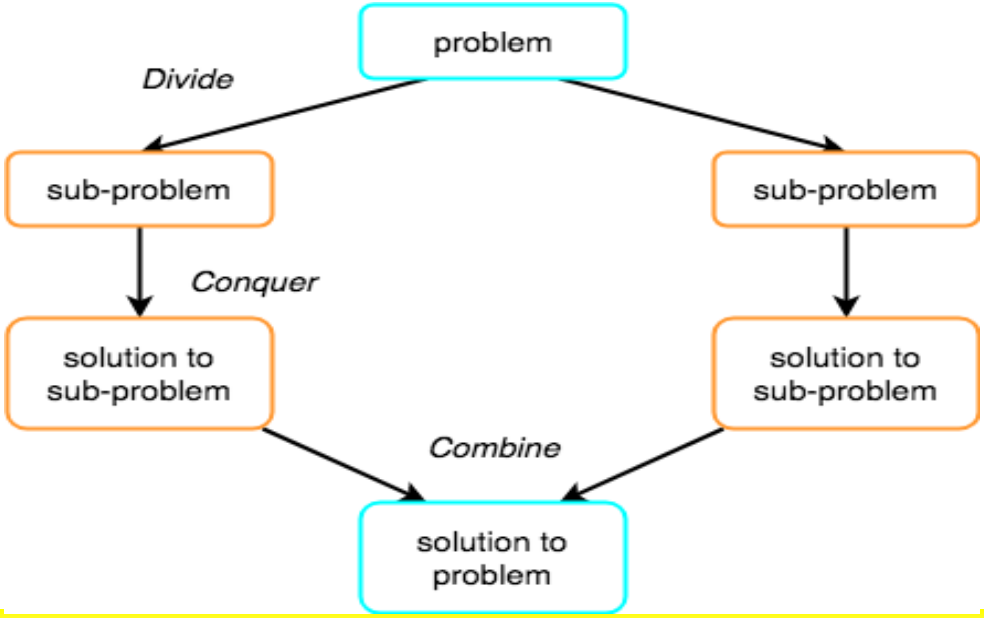


Divide-and-Conquer

Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

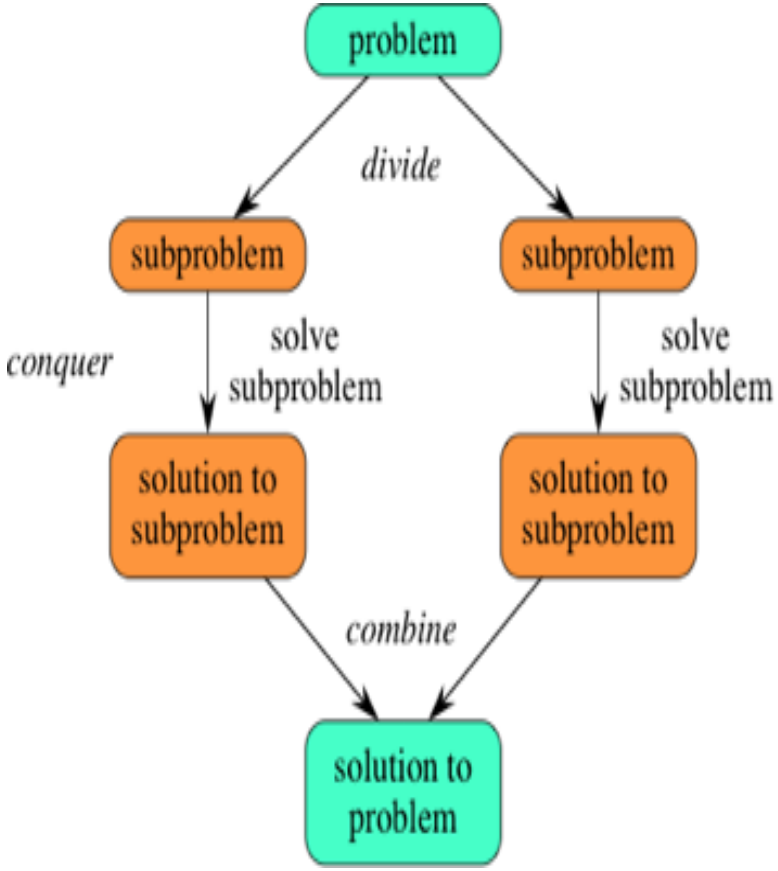
Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Divide and Conquer

- Divide and conquer was a successful military strategy long before it became an algorithm design paradigm.
- The wise general would attack so as to divide the enemy army into two forces and then mop up one after the other.
- To use divide and conquer as an algorithm design technique, we must divide the problem into two smaller sub-problems, solve each of them recursively, and then meld the two partial solutions into one solution to the full problem.
- Whenever the **merging takes less time than solving the two sub-problems**, we get an efficient algorithm

Divide-and-Conquer

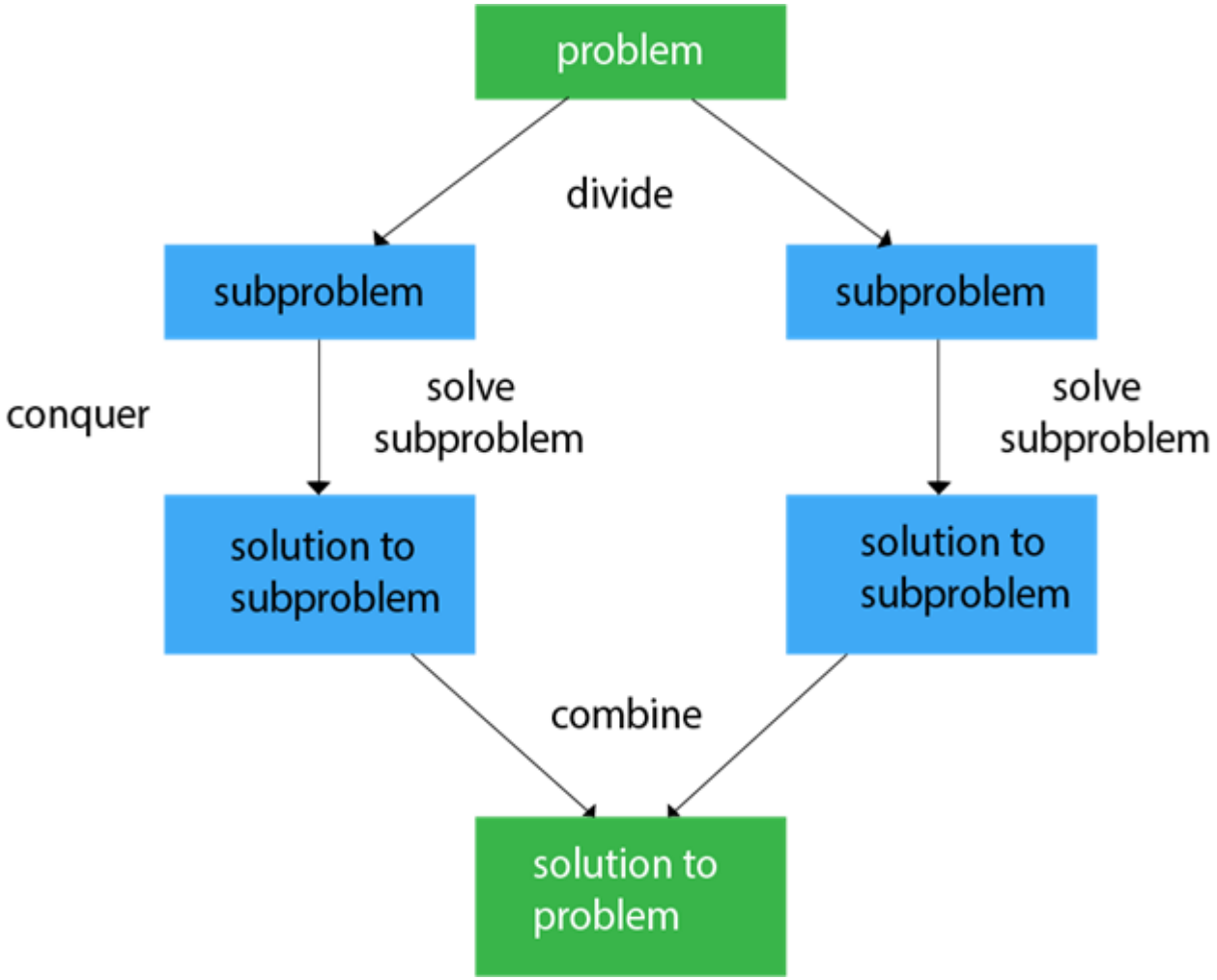
- Divide-and conquer is a general algorithm design paradigm:
  - **Divide**: **divide** the input data S in **two or more disjoint** subsets $S_1, S_2, \ldots$
  - **Recur**: **solve** the sub-problems recursively
  - **Conquer**: **combine** the solutions for $S_1, S_2, \ldots$, into a solution for S
- The **base case** for the recursion are sub-problems of **constant size**
- Analysis can be done using **recurrence equations**



Divide and Conquer – 3 steps

- First step is called **Divide** which is dividing the given problems into smaller sub problems which are identical to the original problem and also these sub problems may or may not be of same size.
- Second step is called **Conquer** where we solve these sub problems recursively.
- Third step is called **Combine** where we combine solutions of the sub problems to get solution for the original problem

Divide and Conquer – 3 steps



Divide and Conquer – 3 steps

- **Divide/Break** :This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.
- **Conquer/Solve**: This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.
- **Merge/Combine:** When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Fundamental of Divide & Conquer Strategy

1. Relational Formula: It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken sub-problems.

2. Stopping Condition: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called as **Stopping Condition**.

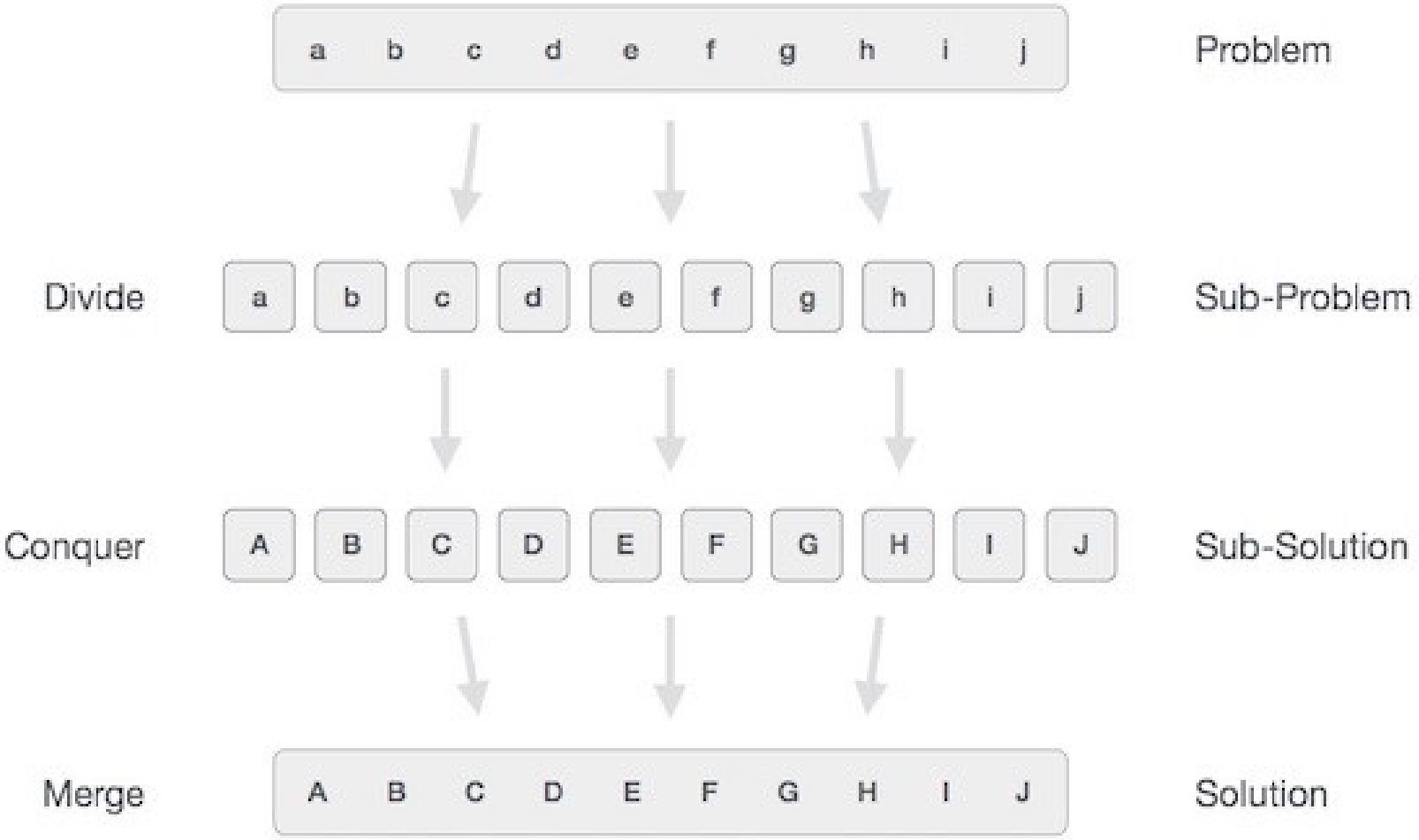
Points to be taken care of

- Threshold lower limit, below which the problem can’t be subdivided or where you want to handle the problem without further divisions.
- The size of the sub instances into which an instance is split.
- Number of such Instances The Algorithm used to combine sub-solutions.

Control abstraction Divide & Conquer

1. Algorithm **DAndC**(P)
2. {
3. If **Small(P)** then return **S(P)**;
4. else
5. {
6. divide P into smaller instances , P_1 , P_2 ,…, P_k , k ≥ 1
7. Apply **DAndC** to each of these sub-problems;
8. Return **Combine** (**DAndC**(P_1), **DAndC**(P_2) ,…, **DAndC**(P_k))
9. }
10. }

Divide and Conquer



Why Divide and Conquer?

- Sometimes it's the simplest approach
- Divide and Conquer is often more efficient than "obvious" approaches
  - E.g. Mergesort, Quicksort
- But, not necessarily efficient
  - Might be the same or worse than another approach
- Must analyze cost
- Note: divide and conquer **may or may not** be implemented recursively

Suitability of D&C

- Divide and conquer is **not suitable** where the solution of size n depends upon n sub-solutions, each of size (n-1).
- **Overlapping Sub problems** - Where sub problems have a dependency on each other and there is no easy way to merge those sub problems and merging will take significant time as solving the original problem.

Examples

- Mergesort is the classic example of a divide-and-conquer algorithm.
- It takes only linear time to merge two sorted listsof n/2 elements each of which was obtained in O(n log n) time.
- Divide and conquer is a design technique with many important algorithms to its credit, including **mergesort**, the **fast Fourier transform**, and **Strassen's matrix multiplication** algorithm.

Merge-Sort

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort*(S, C)

  Input sequence S with n elements, comparator C

  Output sequence S sorted according to C

  if $S.size() > 1$

    $(S_1, S_2) \leftarrow$ *partition*$(S, n/2)$

    mergeSort(S_1, C)

    mergeSort(S_2, C)

    $S \leftarrow$ *merge*(S_1, S_2)

Cost for a Divide and Conquer Algorithm

- Perhaps there is…
  - A cost for dividing into sub problems
  - A cost for solving each of several subproblems
  - A cost to combine results
- So (for *n* > *smallSize*)

$$T(n) = D(n) + \text{Sum}[\ T(\text{size}(I_i))\ ] + C(n)$$

  - often rewritten as

$$T(n) = b\ T(n/c) + f(n)$$

- These formulas are ***recurrence relations***

The concept: Recurrence relations

- Recurrence relations are recursive definitions of mathematical functions or sequences.
- For example, the recurrence relation

$$g(n) = g(n-1) + 2n - 1$$

$$\mathbf{g(0) = 0}$$

defines the function g(n) = n^2

The recurrence relation

$$\mathbf{f(n) = f(n-1) + f(n-2)}$$

$$\mathbf{f(1) = 1,\ f(0) = 1}$$

defines the famous Fibonacci sequence 1,1,2,3,5,8,13,....

What is a Recurrence Relation?

- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.
- A recurrence relation for *T(N)* is simply a recursive definition of *T(N).*
  - This means *T(N)* is written as a function of *T(k)*
  - where $k < N$.
- As exemplified by the two prior examples, two common types are:
  - ***T(N) = a T(N-1) + b***
  - ***T(N) = a T(N/2) + bN + c***

Example Type 1: Divide-and-conquer recurrence

- Assume a divide-and-conquer algorithm divides a problem of size n into a sub-problems.
- Assume each sub-problem is of size n/b.
- Assume f(n) **extra operations** are required to combine the solutions of sub-problems into a solution of the original problem.
- Let T(n) be the number of operations required to solve the problem of **size n**.

$$T(n) = a\,T(n/b) + f(n)$$

- In order to make the recurrence well defined T(n/b) term will actually be either $T(\lceil n/b \rceil)$ or $T(\lfloor n/b \rfloor)$.
- The recurrence will also have to have initial conditions. (e.g. T(1) or T(0))

Example: Recurrence Equations

- **Merge sort:** To sort an array of size n, we sort the left half, sort the right half, and then merge the two halves.

  We can do the merge in linear time (i.e., cn for some positive constant c).

  So, if T(n) denotes the running time on an input of size n, we end up with the recurrence

  $$\mathbf{T(n) = 2T(n/2) + cn}.$$

  This can also be expressed as **T(n) = 2T(n/2) + O(n).**

Example: Recurrence Equations

- **Selection Sort**: In selection sort, we find the smallest element in the input sequence and swap with the leftmost element and then recursively sort the remainder (less the leftmost element) of the sequence.

  This leads to the recurrence **$T(n) = cn + T(n-1)$.**

- **Polynomial Multiplication**: The straightforward divide-and-conquer algorithm to multiply two polynomials of degree n leads to **$T(n) = 4T(n/2) + cn$**.

  However, a clever rearrange of the terms improves this to

  $$T(n) = 3T(n/2)+cn.$$

Example Type 2: Recurrence Equations

Linear recurrence relations:

$$T(\mathbf{n}) = T(\mathbf{n-1}) + \mathbf{n} \text{ for } n>0 \text{ and } \mathbf{T(0) = 1}$$

- These types of recurrence relations can be easily solved using substitution method.

$T(n) = T(n-1) + n$

$= T(n-2) + (n-1) + n$

$= T(n-k) + (n-(k-1))\ldots.. (n-1) + n$

Substituting $k = n$, we get

$$T(n) = T(0) + 1 + 2+\ldots.. +n = n(n+1)/2 = O(n^2)$$

Example Type 3: Value substitution before solving

- Sometimes, recurrence relations can't be directly solved using techniques like substitution, recurrence tree or master method.
- Therefore, we need to convert the recurrence relation into appropriate form before solving.

Example: $T(n) = T(\sqrt{n}) + 1$

To solve this type of recurrence, substitute $n = 2^m$ as:

$$T(2^m) = T(2^m/2) + 1$$

Let $T(2^m) = S(m)$, $S(m) = S(m/2) + 1$

Solving by master method, we get

$S(m) = \Theta(\log m)$ As $n = 2^m$ or $m = \log_2(n)$,

$T(n) = T(2^m) = S(m) = \Theta(\log m) = \Theta(\log\log n)$

Recurrence Relation: Merge sort

```
1.  Algorithm MergeSort(low, high)
2.  / / a[low : high] is a global array to be sorted.
3.  / / Small(P) is true if there is only one element
4.  / / to sort. In this case the list is already sorted.
5.  {
6.  if (low < high) then / / If there are more than one element
7.  {
8.  / / Divide P into subproblems.
9.  / / Find where to split the set.
10. mid:= ⌊ (low + high)/2⌋;
11. / / Solve the subproblems.
12. MergeSort(low, mid);
13. MergeSort(mid + 1, high);
14. / / Combine the solutions.
15. Merge(low, mid, high);
16. }
17. }
```

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

Recurrence Equation Analysis

- The conquer step of **merge-sort** consists of merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes at most bn steps, for some constant b.
- Likewise, the basis case ($n < 2$) will take at b most steps.
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- We can therefore analyze the running time of merge-sort by finding a **closed form solution** to the above equation.
  - That is, a solution that has $T(n)$ only on the left-hand side.

Methods for solving Recurrence Relation

1. Substitution method
2. Telescopic Method
3. Recursion Tree
4. Guess & Test
5. Master's Theorem

- Time complexity of DAndC algorithm can be found out by solving the recurrence relation that represents the algorithm.

1: Iterative Substitution; example 001

- In the iterative substitution, or "plug-and-chug," technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$\begin{aligned} T(n) &= 2T(n/2) + bn \\ &= 2(2T(n/2^2)) + b(n/2)) + bn \\ &= 2^2 T(n/2^2) + 2bn \\ &= 2^3 T(n/2^3) + 3bn \\ &= 2^4 T(n/2^4) + 4bn \\ &= \ldots \\ &= 2^i T(n/2^i) + ibn \end{aligned}$$

- Note that base, T(n)=b, case occurs when $2^i=n$. That is, i = log n.
- So, $T(n) = bn + bn \log n$
- Thus, T(n) is O(n log n).

1: Iterative Substitution; example 002

$$T(2) = 1 \qquad (\text{assume } N = 2^k)$$

$$T(N) = 2T(N/2) + 2$$

$$T(N) = 2[2T(N/4) + 2] + 2$$

$$T(N) = 2^2 T(N/2^2) + (2^2 + 2)$$

$$T(N) = 2^r T(N/2^r) + \sum_{i=1}^{r-1} 2^i$$

1: Iterative Substitution; example 002

$$T(N) = 2^{k-1}T(N/2^{k-1}) + 2\sum_{i=0}^{k-2} 2^i$$

$$T(N) = 2^{k-1} + 2(2^{k-1} - 1)$$

$$T(N) = \frac{N}{2} + 2(\frac{N}{2} - 1)$$

Note : $\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$

$$T(N) = \frac{3N}{2} - 2$$

Note, although this is still *O(N),* it is better than the *T(N)=2N* found earlier.

Exercises

Solve the recurrence relations with **$T(1) = 1, N = 2^m$**

(a) $T(N) = T(N/2) + 1$

(b) $T(N) = T(N/2) + N$

(c) $T(N) = 2T(N/2) + 1$

(d) $T(N) = 2T(N - 1) + 1$

(e) $T(N) = 2T(N - 1) + N$

2. Telescopic Sum

- Re-write the recurrence relation for subsequent smaller values of **n** so that the left side of the re-written relation is equal to the first term of the right side of the previous relation.
- Re-write the relation until a relation involving the initial condition is obtained.
- Add the left sides and the right sides of the relations.
- Cancel the equal terms on the left and the right side.
- Add the remaining terms on the right side. The sum will give the general formula of the sequence

2. Telescopic Sum; example 001

- Given recurrence relation:

  T(1) = 1 ; N=1

  T(N) = T(N-1) + 1; N >1

Telescoping:

T(N) = ~~T(N-1)~~ + 1

~~T(N-1)~~ = ~~T(N-2)~~ + 1

~~T(N-2)~~ = T(N-3) + 1

….

T(3) = T(2) +1

T(2) = T(1) + 1

T(1) = **1**

- Adding the left and the right side and canceling equal terms, we obtain:
- T(N) = **1 + 1 + … + 1**
- There are N lines in the telescoping, thus the sum of the 1s is N:
- **T(N) = N**
- Hence T(N) = O(N)

2. Telescopic Sum; example 002

- **Merge sort:**

T(1) = 1 ; N=1

T(N) = 2T(N/2) + N; N > 1

Preprocessing –

- divide both sides by N

T(N) / N = T(N/2) / (N/2) + 1

N = 2^m

Telescoping:

T(N) / N = ~~T(N/2) / (N/2)~~ + 1

~~T(N/2) / (N/2)~~ = ~~T(N/4) / (N/4)~~ + 1

~~T(N/4) / (N/4)~~ = T(N/8) / (N/8) + 1

…

T(8)/8 = T(4)/4 + 1

T(4)/4 = T(2)/2 + 1

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

- Adding the left and the right side and canceling equal terms, we obtain:
- **T(N)/N = T(1) + 1 + … + 1**

2. Telescopic Sum; example 002

- **T(N)/N = T(1) + 1 + …. + 1**
- There are **logN equations**, thus

1 + … + 1 = logN

T(N)/N = T(1) + logN

T(N)/N = LogN + **1**

because T(1) = 1

T(N) = NlogN + N

Hence T(N) = O(NlogN)

- T(N) = O(NlogN)

2. Telescopic Sum; example 003

T (n) = T (n-1) +1 and **T (1) = θ (1)**.

Solution:

$$
\begin{aligned}
T(n) &= T(n-1)+1 \\
&= (T(n-2)+1)+1 \\
&= (T(n-3)+1)+1+1 \\
&= T(n-4)+4 \\
&= T(n-5)+1+4 \\
&= T(n-5)+5, \ldots = T(n-k)+k \text{ ; Where } k = n-1
\end{aligned}
$$

$T(n-k) = T(1) = \theta(1)$

$T(n) = \theta(1) + (n-1) = 1+n-1=n= \theta(n)$.

2. Telescopic Sum; example 004

$T(n) = 1$ **if** $n=1$

$T(n) = 2T(n-1)$ **if** $n>1$

$T(n) = 2T(n-1)$

$= 2[2T(n-2)]$

$= 2^2T(n-2) = 4[2T(n-3)]$

$= 2^3T(n-3)$

$= 8[2T(n-4)] = 2^4T(n-4)]$ EQ.1

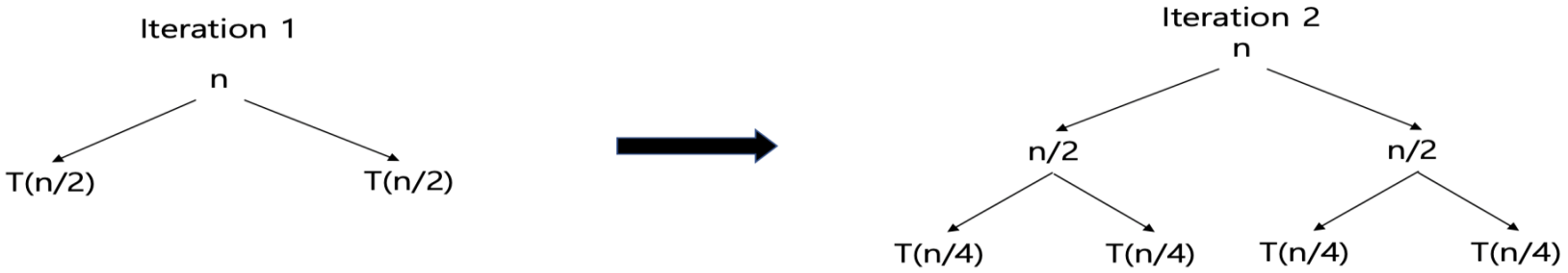
Repeat the procedure for i times $\Rightarrow T(n) = 2^i T(n-i)$

Putting $n-i=1$ or $i= n-1$ in (Eq.1) $\Rightarrow$

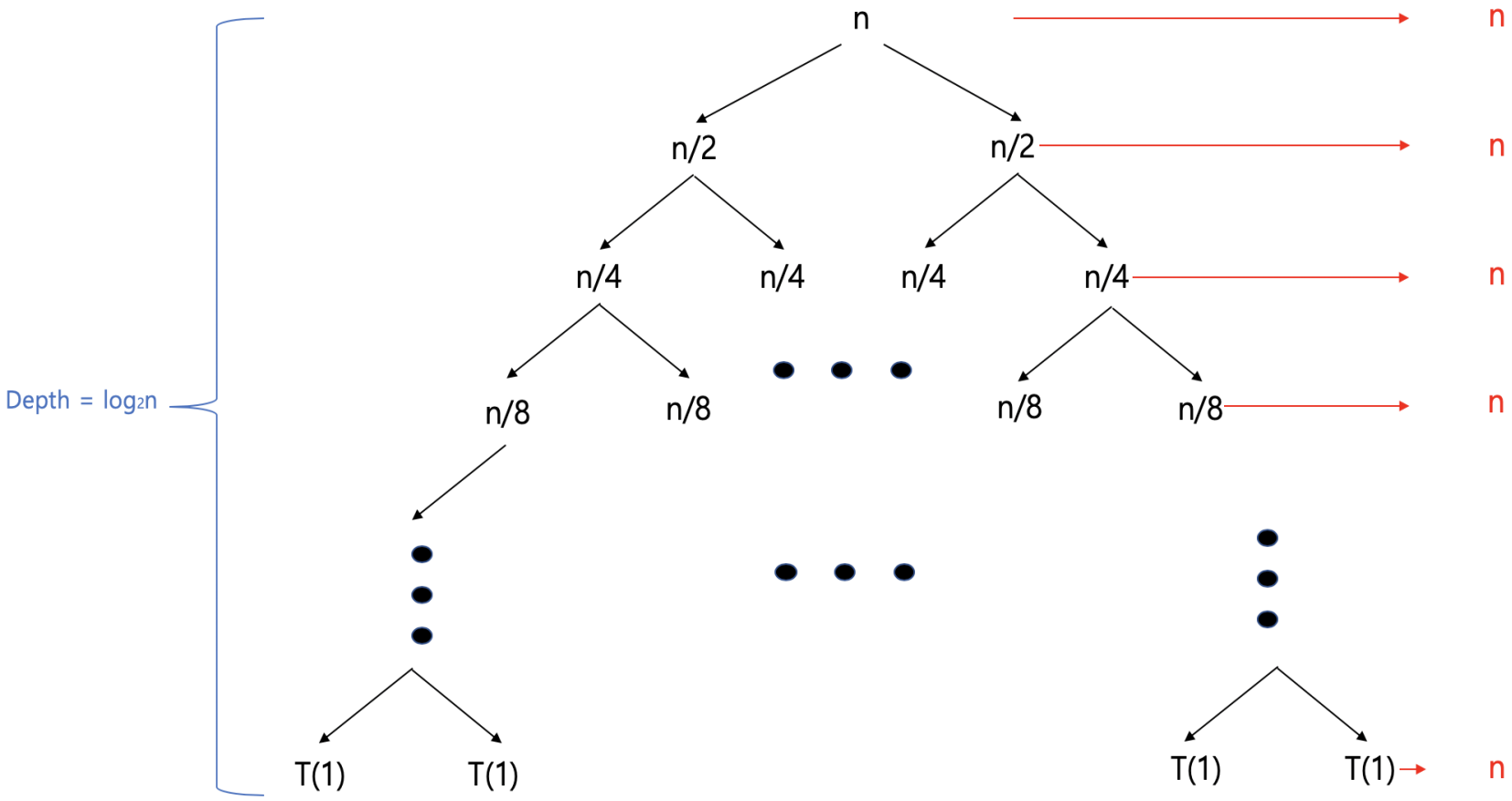
$T(n) = 2^{n-1} T(1) = 2^{n-1} .1$ {$T(1) =1$ .....given} $= 2^{n-1}$

3: The Recursion Tree

- *A **recursion tree** is a tree whose each node represents the cost of a certain recursive sub-problem. In order to get the cost of the entire algorithm, we need to sum up the costs in each node.*
- *One of the strengths of the recursion tree is that it is useful to visualize what happens when a recurrence is iterated.*
- *Its diagram shows the structure of recursive calls, which means that how the main-problem divided into sub-problems, and the costs for combining those sub-tasks.*
- ***The recursion tree of the merge sort with its recurrence relation T(n) =2T(n/2)+n***



Complete recursion tree of the merge sort: T(n) = 2T(n/2) + n



Total=n * log2n

Explaining the proof that T(n) = O(nlogn)

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(\frac{n}{2}) + n & n > 1 \end{cases}$$

$$T(n) = 2T(\frac{n}{2}) + n$$

$$= 2(2T(\frac{n}{4}) + \frac{n}{2}) + n = 4T(\frac{n}{4}) + 2n$$

$$= 2(2(2T(\frac{n}{8}) + \frac{n}{4}) + \frac{n}{2}) + n = 8T(\frac{n}{8}) + 3n$$

$$\cdots$$

$$= 2^{log_2 n} T(\frac{n}{2^{log_2 n}}) + n \times log_2 n$$

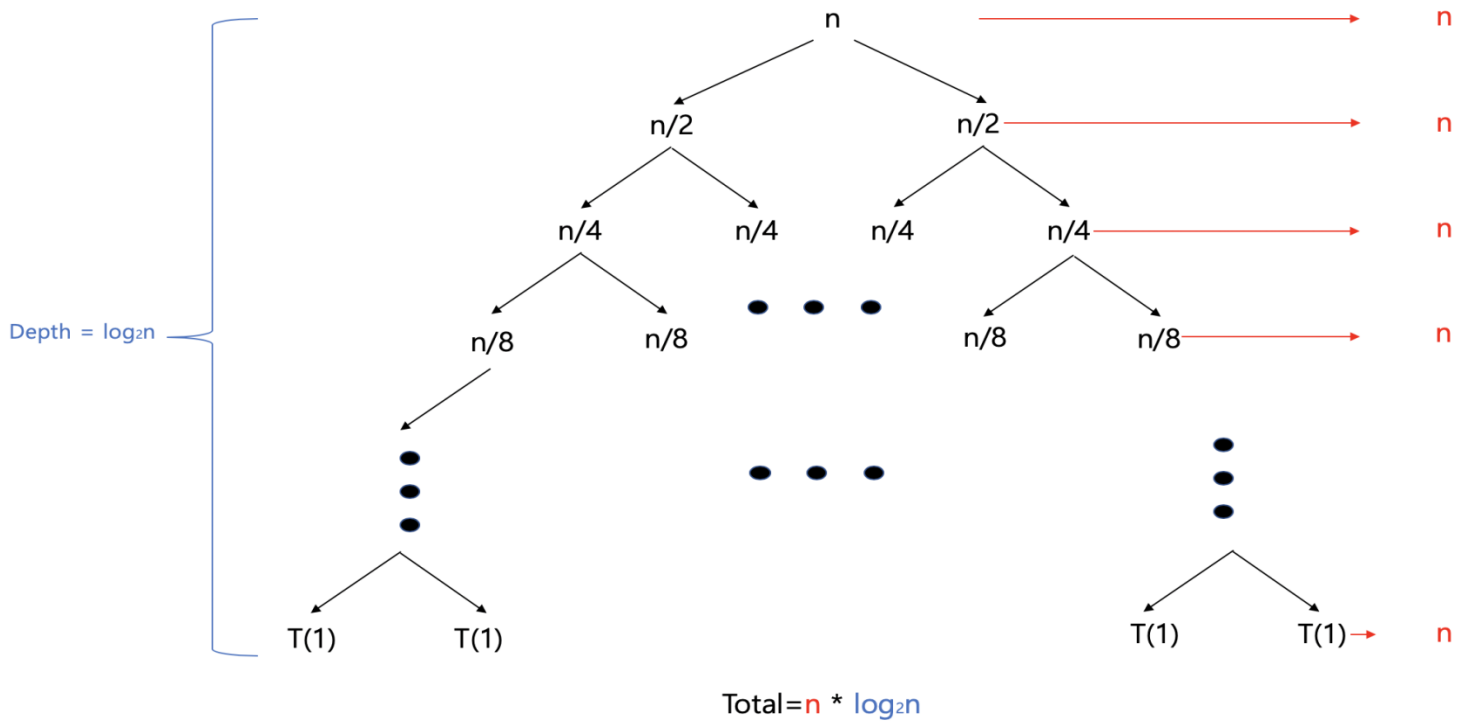
$$= nT(1) + n \times log_2 n \leq c \times n \times log_2 n$$

$$\therefore T(n) \in O(nlog_2 n)$$

The Recursion Tree

(1) The depth of the recursion tree is ***log base b of n*** *when the recurrence is in the form of T(n) = aT(n/b)+f(n)*

(2) The depth of the recursion tree means ***the length of the longest branch*** *among its branches.*

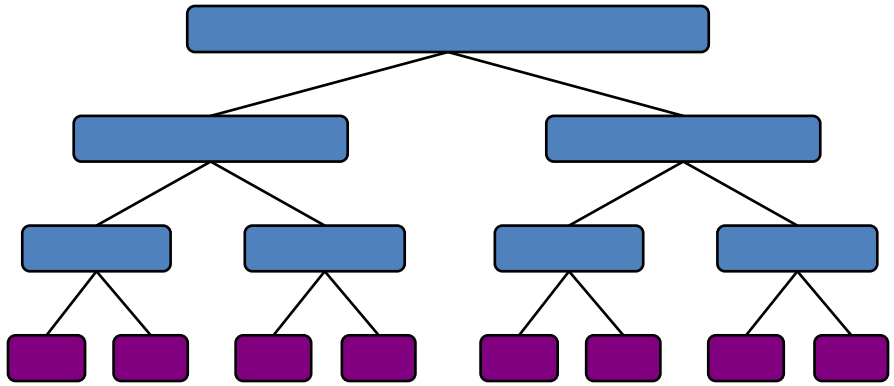


3: The Recursion Tree

- Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

depth	T's	size	time
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
…	…	…	…

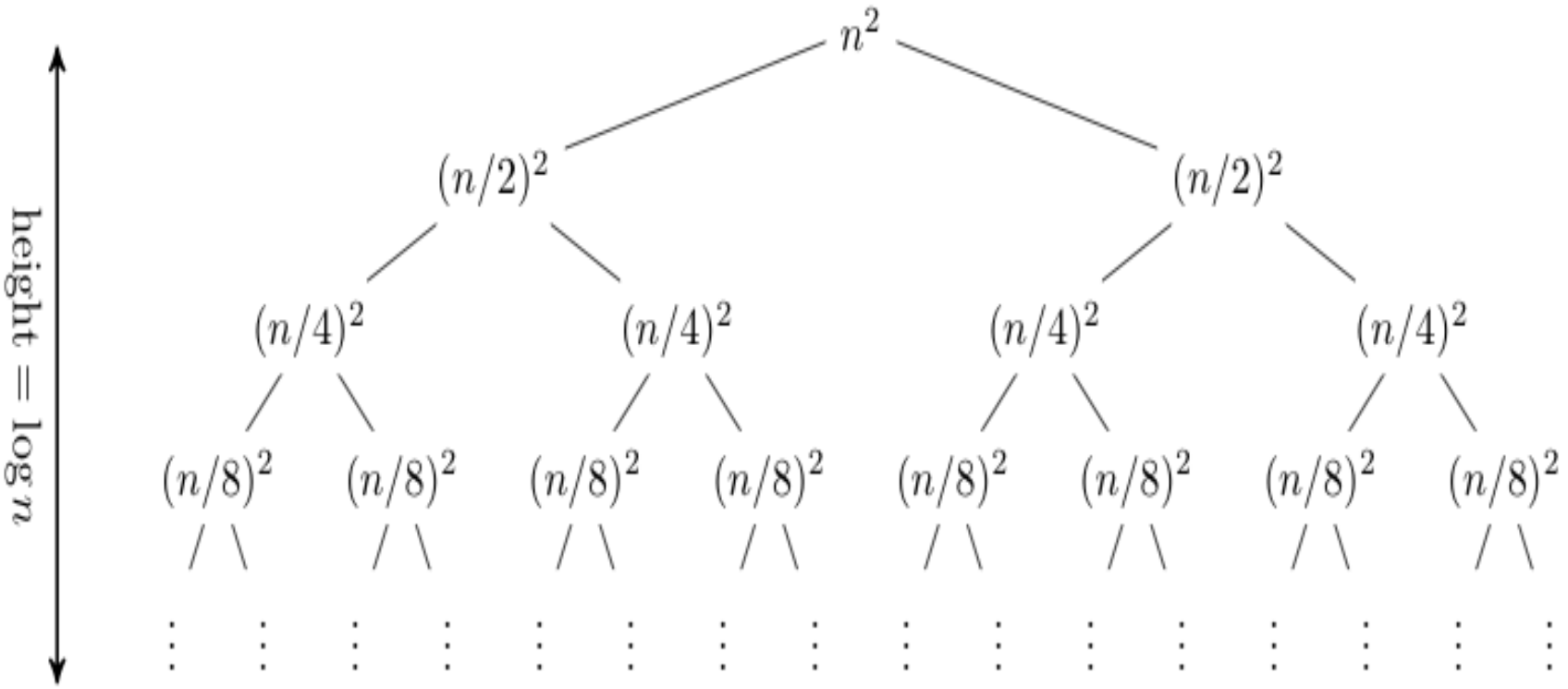


Total time = $bn + bn \log n$

(last level plus all previous levels)

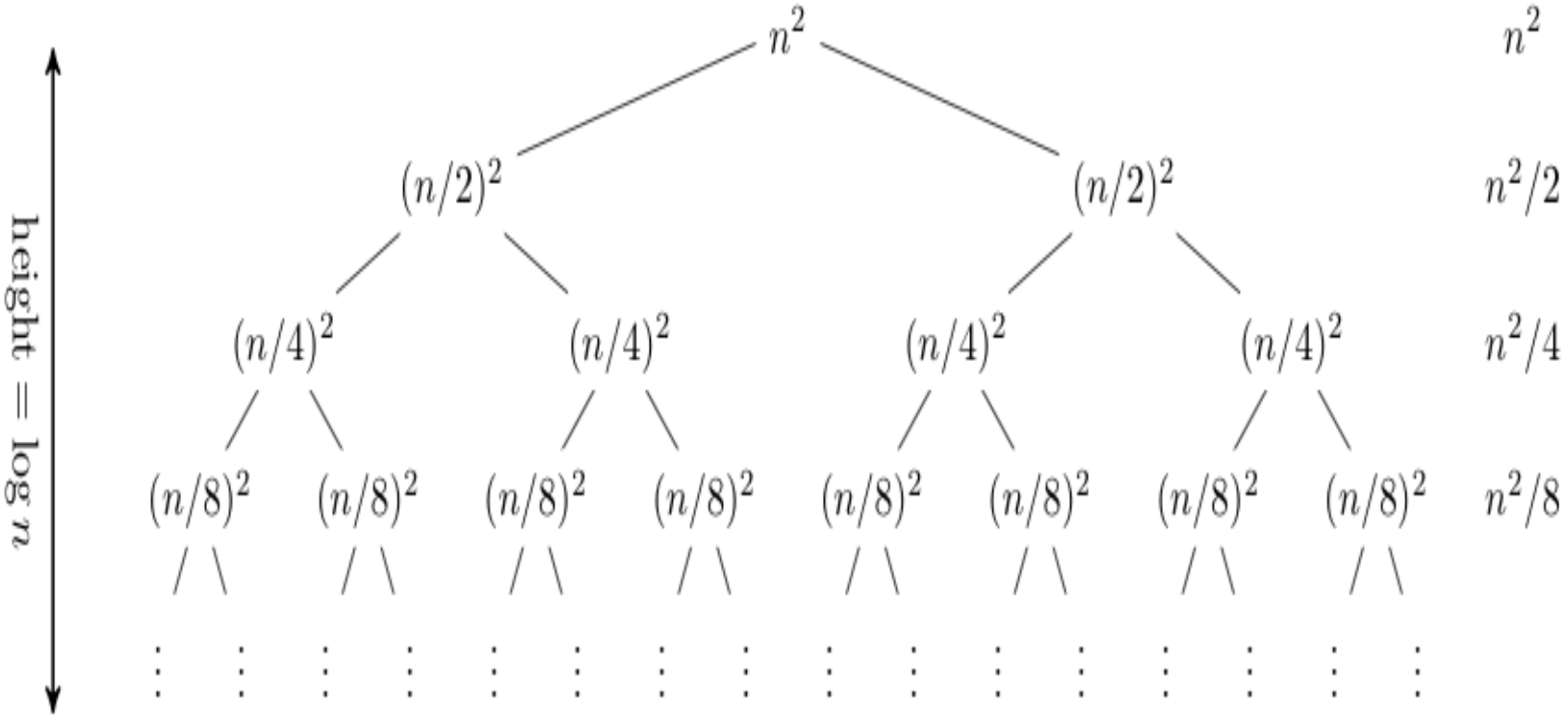
3: The Recursion Tree

- A ***recursion tree*** is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.
- Consider the recurrence : $T(n) = 2T(n/2) + n^2$.



3: The Recursion Tree

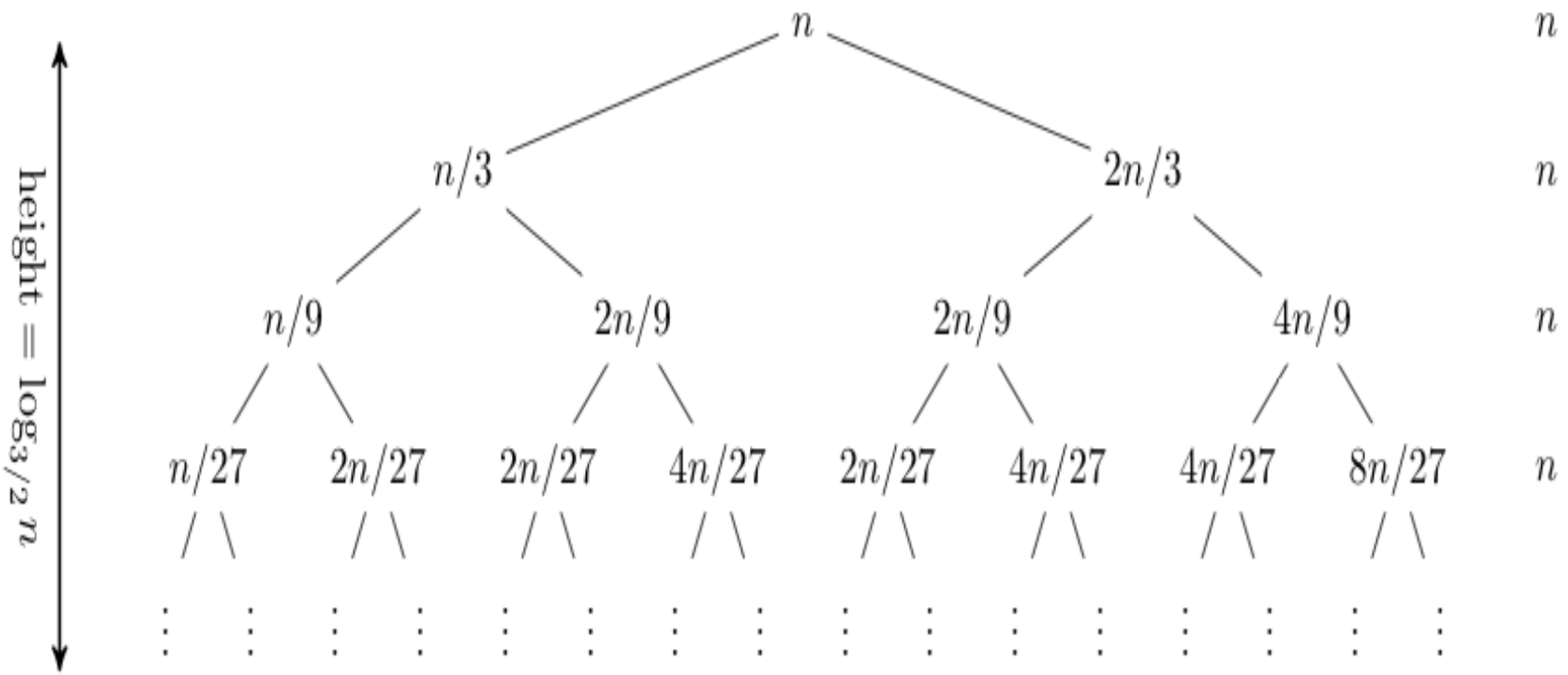
- sum across each row of the tree to obtain the total work done at a given level:



This a geometric series, thus in the limit the sum is $O(n^2)$. The depth of the tree in this case does not really matter; the amount of work at each level is decreasing so quickly that the total is only a constant factor more than the root.

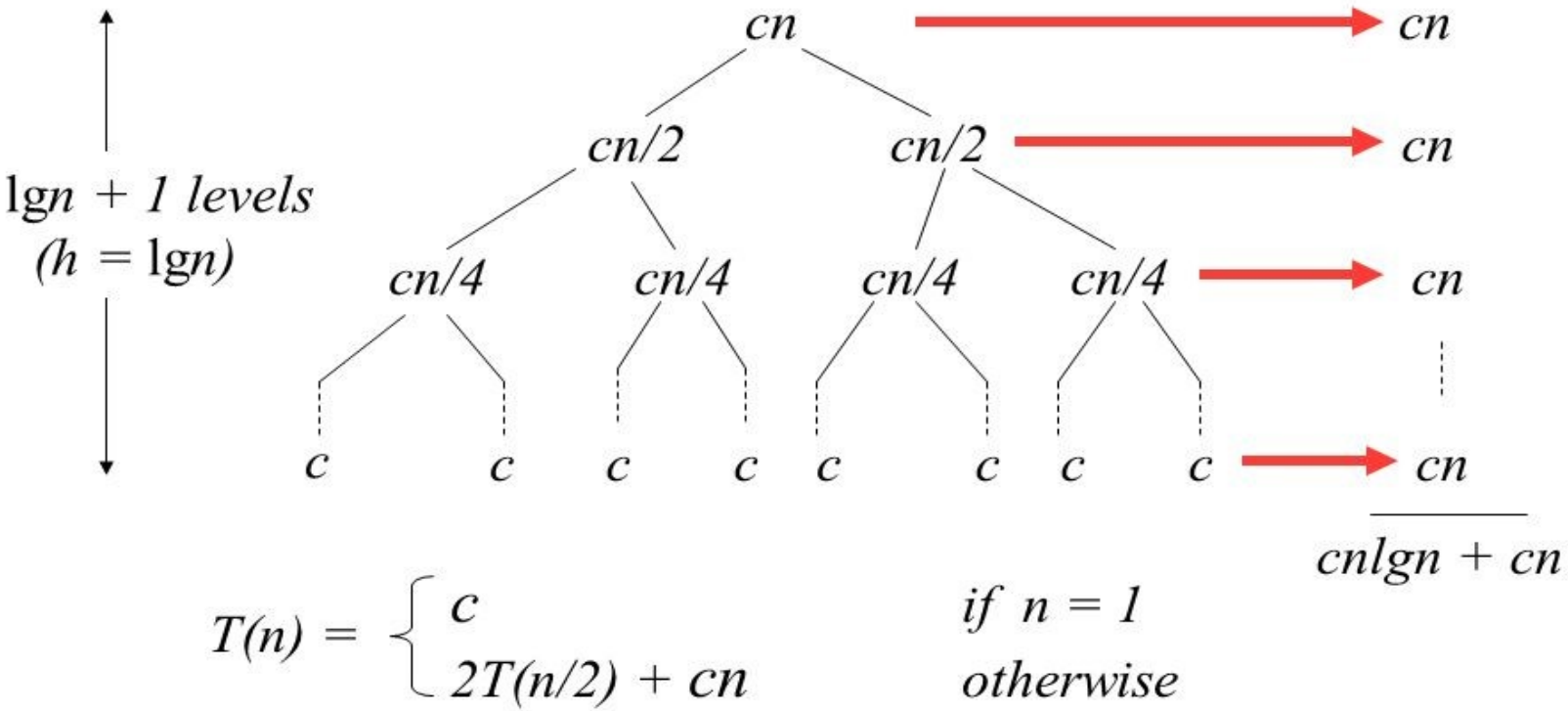
3: The Recursion Tree

*Considering T(n) = T(n/3) + T(2n/3) + n, e*xpanding out the first few levels, the recurrence tree is:



the tree here is not balanced: the longest path is the rightmost one, and its length is ***$\log_{3/2} n$***. Hence our guess for the closed form of this recurrence is ***O(n log n)***.

Recursion Tree for Merge-Sort



$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

Recurrence for worst-case running time for Merge-Sort

4: Guess-and-Test Method

- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

- Guess: **T(n) < cn log n**.

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &= 2(c(n/2)\log(n/2)) + bn\log n \\ &= cn(\log n - \log 2) + bn\log n \\ &= cn\log n - cn + bn\log n \end{aligned}$$

- **Wrong**: we cannot make this last line be less than **cn log n**

4: Guess-and-Test Method, Part 2

- Recall the recurrence equation:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

- Guess #2: **$T(n) < cn\log^2 n$.**

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &= 2(c(n/2)\log^2(n/2)) + bn\log n \\ &= cn(\log n - \log 2)^2 + bn\log n \\ &= cn\log^2 n - 2cn\log n + cn + bn\log n \\ &\leq cn\log^2 n \end{aligned}$$

  - if c > b.

- So, **$T(n)$ is $O(n\log^2 n)$**.
- In general, to use this method, you need to have a good guess and you need to be good at induction proofs.

5: Master Method

- Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

5: The Master Theorem

- The solution to the recurrence $T(n) = aT(n/b) + \Theta(n^k)$; $T(1) = \Theta(1)$, where a, b, and k are all constants, is given by:

$$T(n) = \Theta(n^k) \text{ if } a < b^k$$
$$T(n) = \Theta(n^k \log n) \text{ if } a = b^k$$
$$T(n) = \Theta(n \log_b a) \text{ if } a > b^k$$

- The theorem just given is a simplified version. The general version does not restrict the on recursive term to be a polynomial in terms of *n*.

Master Method, Example 1

- The form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Example:

$$T(n) = 4T(n/2) + n$$

Solution: $\log_b a = 2$, so case 1 says T(n) is $O(n^2)$.

Master Method, Example 2

- The form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Example: $T(n) = 2T(n/2) + n \log n$

Solution: $\log_b a = 1$, so case 2 says T(n) is $O(n \log^2 n)$.

Master Method, Example 3

- The form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Example:

$$T(n) = T(n/3) + n \log n$$

Solution: $\log_b a = 0$, so case 3 says T(n) is O(n log n).

Master Method, Example 4

- The form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Example:

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

Solution: $\log_b a = 3$, so case 1 says T(n) is $O(n^3)$.

Master Method, Example 5

- The form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
   provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Example:

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

Solution: $\log_b a = 2$, so case 3 says T(n) is $O(n^3)$.

Master Method, Example 6

- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Example:

$$T(n) = T(n/2) + 1 \quad \text{(binary search)}$$

Solution: $\log_b a = 0$, so case 2 says T(n) is O(log n).

Master Method, Example 7

- The form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
   provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

- Example:

$$T(n) = 2T(n/2) + \log n$$ (heap construction)

Solution: $\log_b a = 1$, so case 1 says T(n) is O(n).

Iterative "Proof" of the Master Theorem

- Using iterative substitution, let us see if we can find a pattern:

$$
\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a(aT(n/b^2)) + f(n/b)) + bn \\
&= a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\
&= \ldots \\
&= a^{\log_b n}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i) \\
&= n^{\log_b a}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i)
\end{aligned}
$$

- We then distinguish the three cases as
  - The first term is dominant
  - Each part of the summation is equally dominant
  - The summation is a geometric series

Proof of Masters Theorem

Let $T(n)$ be defined by the recurrence $T(n) = aT(n/b) + \Theta(n^k)$, for some constants $a \geq 1$, $b > 1$, and $k \geq 0$. Then we can conclude the following about the asymptotic complexity of $T(n)$:

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

Proof of Masters Theorem

Theorem

The solution to the equation $T(N) = aT(N/b) + \Theta(N^k)$, where $a \geq 1$ and $b > 1$, is

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{if } a > b^k \\ O(N^k \log N) & \text{if } a = b^k \\ O(N^k) & \text{if } a < b^k \end{cases}$$

Proof

we will assume that N is a power of b; thus, let $N = b^m$. Then $N/b = b^{m-1}$ and $N^k = (b^m)^k = b^{mk} = b^{km} = (b^k)^m$. Let us assume $T(1) = 1$, and ignore the constant factor in $\Theta(N^k)$. Then we have

$$T(b^m) = aT(b^{m-1}) + (b^k)^m$$

If we divide through by a^m, we obtain the equation

$$\frac{T(b^m)}{a^m} = \frac{T(b^{m-1})}{a^{m-1}} + \left\{\frac{b^k}{a}\right\}^m \qquad (10.3)$$

We can apply this equation for other values of m, obtaining

Proof of Masters Theorem

$$\frac{T(b^m)}{a^m} = \frac{T(b^{m-1})}{a^{m-1}} + \left\{\frac{b^k}{a}\right\}^m \tag{10.3}$$

We can apply this equation for other values of m, obtaining

$$\frac{T(b^{m-1})}{a^{m-1}} = \frac{T(b^{m-2})}{a^{m-2}} + \left\{\frac{b^k}{a}\right\}^{m-1} \tag{10.4}$$

$$\frac{T(b^{m-2})}{a^{m-2}} = \frac{T(b^{m-3})}{a^{m-3}} + \left\{\frac{b^k}{a}\right\}^{m-2} \tag{10.5}$$

$$\vdots$$

$$\frac{T(b^1)}{a^1} = \frac{T(b^0)}{a^0} + \left\{\frac{b^k}{a}\right\}^1 \tag{10.6}$$

We use our standard trick of adding up the telescoping equations (10.3) through (10.6). Virtually all the terms on the left cancel the leading terms on the right, yielding

Proof of Masters Theorem

We use our standard trick of adding up the telescoping equations (10.3) through (10.6). Virtually all the terms on the left cancel the leading terms on the right, yielding

$$\frac{T(b^m)}{a^m} = 1 + \sum_{i=1}^{m} \left\{ \frac{b^k}{a} \right\}^i \tag{10.7}$$

$$= \sum_{i=0}^{m} \left\{ \frac{b^k}{a} \right\}^i \tag{10.8}$$

Thus

$$T(N) = T(b^m) = a^m \sum_{i=0}^{m} \left\{ \frac{b^k}{a} \right\}^i \tag{10.9}$$

Proof of Masters Theorem

If $a > b^k$, then the sum is a geometric series with ratio smaller than 1. Since the sum of infinite series would converge to a constant, this finite sum is also bounded by a constant, and thus Equation (10.10) applies:

$$T(N) = O(a^m) = O(a^{\log_b N}) = O(N^{\log_b a}) \tag{10.10}$$

If $a = b^k$, then each term in the sum is 1. Since the sum contains $1 + \log_b N$ terms and $a = b^k$ implies that $\log_b a = k$,

$$\begin{aligned} T(N) &= O(a^m \log_b N) = O(N^{\log_b a} \log_b N) = O(N^k \log_b N) \\ &= O(N^k \log N) \end{aligned} \tag{10.11}$$

Finally, if $a < b^k$, then the terms in the geometric series are larger than 1, and the second formula in Section 1.2.3 applies. We obtain

$$T(N) = a^m \frac{(b^k/a)^{m+1} - 1}{(b^k/a) - 1} = O(a^m (b^k/a)^m) = O((b^k)^m) = O(N^k) \tag{10.12}$$

Suggested Exercise

Theorem

The solution to the equation $T(N) = aT(N/b) + \Theta(N^k \log^p N)$, where $a \geq 1$, $b > 1$, and $p \geq 0$ is

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{if } a > b^k \\ O(N^k \log^{p+1} N) & \text{if } a = b^k \\ O(N^k \log^p N) & \text{if } a < b^k \end{cases}$$

Theorem

If $\sum_{i=1}^{k} \alpha_i < 1$, then the solution to the equation $T(N) = \sum_{i=1}^{k} T(\alpha_i N) + O(N)$ is $T(N) = O(N)$.

Solving recurrence relation

Example 3.1 Consider the case in which $a = 2$ and $b = 2$. Let $T(1) = 2$ and $f(n) = n$. We have

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \end{aligned}$$

Substitution method

In general, we see that $T(n) = 2^i T(n/2^i) + in$, for any $\log_2 n \geq i \geq 1$. In particular, then, $T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + n \log_2 n$, corresponding to the choice of $i = \log_2 n$. Thus, $T(n) = nT(1) + n \log_2 n = n \log_2 n + 2n$. □

Application of masters theorem

Example 3.2 Look at the following recurrence when n is a power of 2:

$$T(n) = \begin{cases} T(1) & n = 1 \\ T(n/2) + c & n > 1 \end{cases}$$

Comparing with (3.2), we see that $a = 1$, $b = 2$, and $f(n) = c$. So, $\log_b(a) = 0$ and $h(n) = f(n)/n^{\log_b a} = c = c(\log n)^0 = \Theta((\log n)^0)$. From Table 3.1, we obtain $u(n) = \Theta(\log n)$. So, $T(n) = n^{\log_b a}[c + \Theta(\log n)] = \Theta(\log n)$. □

Example 3.3 Next consider the case in which $a = 2$, $b = 2$, and $f(n) = cn$. For this recurrence, $\log_b a = 1$ and $h(n) = f(n)/n = c = \Theta((\log n)^0)$. Hence, $u(n) = \Theta(\log n)$ and $T(n) = n[T(1) + \Theta(\log n)] = \Theta(n \log n)$. □

Example 3.4 As another example, consider the recurrence $T(n) = 7T(n/2) + 18n^2$, $n \geq 2$ and a power of 2. We obtain $a = 7$, $b = 2$, and $f(n) = 18n^2$. So, $\log_b a = \log_2 7 \approx 2.81$ and $h(n) = 18n^2/n^{\log_2 7} = 18n^{2-\log_2 7} = O(n^r)$, where $r = 2 - \log_2 7 < 0$. So, $u(n) = O(1)$. The expression for $T(n)$ is

$$\begin{aligned} T(n) &= n^{\log_2 7}[T(1) + O(1)] \\ &= \Theta(n^{\log_2 7}) \end{aligned}$$

as $T(1)$ is assumed to be a constant. □

Application of masters theorem

Example 3.4 As another example, consider the recurrence $T(n) = 7T(n/2) + 18n^2$, $n \geq 2$ and a power of 2. We obtain $a = 7$, $b = 2$, and $f(n) = 18n^2$. So, $\log_b a = \log_2 7 \approx 2.81$ and $h(n) = 18n^2/n^{\log_2 7} = 18n^{2-\log_2 7} = O(n^r)$, where $r = 2 - \log_2 7 < 0$. So, $u(n) = O(1)$. The expression for $T(n)$ is

$$\begin{aligned} T(n) &= n^{\log_2 7}[T(1) + O(1)] \\ &= \Theta(n^{\log_2 7}) \end{aligned}$$

as $T(1)$ is assumed to be a constant. □

Example 3.5 As a final example, consider the recurrence $T(n) = 9T(n/3) + 4n^6$, $n \geq 3$ and a power of 3. Comparing with (3.2), we obtain $a = 9$, $b = 3$, and $f(n) = 4n^6$. So, $\log_b a = 2$ and $h(n) = 4n^6/n^2 = 4n^4 = \Omega(n^4)$. From Table 3.1, we see that $u(n) = \Theta(h(n)) = \Theta(n^4)$. So,

$$\begin{aligned} T(n) &= n^2[T(1) + \Theta(n^4)] \\ &= \Theta(n^6) \end{aligned}$$

as $T(1)$ can be assumed constant. □

Application of masters theorem

$$T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$$

$T(n) = aT(n/b) + \Theta(n^k)$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

As one can see from the formula above:

$a = 8,\ b = 2,\ f(n) = 1000n^2$, so

$f(n) = O\left(n^c\right)$, where $c = 2$

Next, we see if we satisfy the case 1 condition:

$$\log_b a = \log_2 8 = 3 > c.$$

It follows from the first case of the master theorem that

$$T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^3\right)$$

Application of masters theorem

EX 1

$$T(n) = aT(n/b) + \Theta(n^k)$$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

$$T(n) = 2T(n/2) + dn + e$$

$$a = 2 \qquad b = 2 \qquad k = 1$$

$$\log_b a = 1 \quad = k$$

Case 1 $\qquad T(n) = O(n \log n)$

Application of masters theorem

EX 2

$$T(n) = aT(n/b) + \Theta(n^k)$$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

$$T(n) = 9T(n/3) + n$$

$$a = 9 \qquad b = 3 \qquad k = 1$$

$$\log_b a = 2 > k$$

Case 2 $\qquad T(n) = O(n^2)$

Application of masters theorem

EX 3

$$T(n) = aT(n/b) + \Theta(n^k)$$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

$$T(n) = 10T(n/3) + n$$

$$\log_b a = log_3 10 \approx 2.1 > k = 1$$

$$\text{Case 2 : } T(n) = O(n^{log_3 10}) \approx O(n^{2.1})$$

Application of masters theorem

EX 4

$$T(n) = aT(n/b) + \Theta(n^k)$$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

$$T(n) = 10T(n/3) + n^4$$

$$\log_b a = \log_3 10 \approx 2.1 < k = 4$$

Case 3: $T(n) = O(n^4)$

Application of masters theorem

EX 5

$$T(n) = aT(n/b) + \Theta(n^k)$$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

$$T(n) = T(2n/3) + 1$$

$$a = 1 \qquad b = 3/2 \qquad k = 0$$

$$\log_b a = \log_{3/2} 1 = 0 = k$$

Case 1: $T(n) = O(\log n)$

Application of masters theorem

EX 6

$$T(n) = aT(n/b) + \Theta(n^k)$$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

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

Cannot use the master theorem directly, but can still do some bounding

$$T(n/3) \leq T(n/2)$$

$$T(n) = T(n/2) + T(n/3) + n \leq 2T(n/2) + n = O(n \log n)$$

$$T(n) = O(n \log n)$$

Summary: **Master theorem**

- Master theorem lets you go from the recurrence to the asymptotic bound very quickly, so you're more like a pro.
- It typically works well for divide-and-conquer algorithms
- But master theorem does not apply to all recurrences. When it does not apply, you can:

  - ❑ do some upper/lower bounding and get a potentially looser bound
  - ❑ use the substitution method

Exercise

1. Solve the recurrence relation (3.2) for the following choices of a, b, and $f(n)$ (c being a constant):

   (a) $a = 1,\ b = 2$, and $f(n) = cn$

   (b) $a = 5,\ b = 4$, and $f(n) = cn^2$

   (c) $a = 28,\ b = 3$, and $f(n) = cn^3$

2. Solve the following recurrence relations using the substitution method:

   (a) All three recurrences of Exercise 1.

   (b)
   $$T(n) = \begin{cases} 1 & n \leq 4 \\ T(\sqrt{n}) + c & n > 4 \end{cases}$$

   (c)
   $$T(n) = \begin{cases} 1 & n \leq 4 \\ 2T(\sqrt{n}) + \log n & n > 4 \end{cases}$$

   (d)
   $$T(n) = \begin{cases} 1 & n \leq 4 \\ 2T(\sqrt{n}) + \frac{\log n}{\log \log n} & n > 4 \end{cases}$$

Example: Iterative Factorial Algorithm

```
int factorial (N)
{
   temp = 1;
   for i = 1 to N {
       temp =
temp*i;
   }
   return(temp);
}
```

Time Units to Compute

N loops.
c for the multiplication.

$$T(N) = \sum_{i=1}^{N} c = cN$$

Thus this requires *O(N)* computation.

Example: Recursive Factorial Algorithm

```
int factorial (N)
{
   if (N == 1)
return 1;
   else return(N *
factorial(N-1));
}
```

Time Units to Compute

---

c for the conditional
d for the multiplication and then T(N-1).

$$T(1) = c$$

$$T(N) = c + d + T(N-1)$$

T(N) = c+d+T(N-1) = f+T(N-1) is a recurrence relation.
How do we solve it?

Example : Search Algorithm

```
int a[N]; // An array
int value; // Value to be found

int search ( ) {
    for (i = 0; i < N; i++) {
        if (a[i] == value)
return(i);
    }
    return(-1);
}
```

Time Units to Compute

N loops

c for comparison

$$T(N) = \sum_{i=0}^{N-1} c = cN$$

Thus this is an *O(N)* algorithm.

Example : Binary Search Algorithm

```
int a[N]; // Sorted array
int x; // Value to be found

int bin_search (int left, int right) {
    if (left > right) return –1;

    int mid = (left+right) / 2;
    if (x < a[mid])
        bin_search(left, mid-1);
    else if (x > a[mid])
        bin_search(mid+1,right);
    else return mid;
}
```

Time Units to Compute
c for comparison.
d for computation of mid.
c for comparison.
T(N/2)
c for comparison.
T(N/2)

Thus **T(N) = T(N/2) + 3c+d = T(N/2)+f**

Analysis

$$T(1) = c \qquad (\text{assume} \quad N = 2^k)$$

$$T(N) = T(N/2) + f$$

$$T(N) = T(N/4) + 2f$$

$$T(N) = T(N/2^k) + kf = c + kf$$

$$T(N) = c + k \log N = O(\log N)$$

Multiplying Two n Bit Numbers

- The naive approach is to simply multiply the two numbers which takes O($\mathbf{n^2}$) time because we need to multiply each digit in one number with those in the second number, put them in the correct position and add them
- The divide-and-conquer technique to ***multiply two n bit number*** is attributed to a 1962 paper by Karatsuba, and indeed it is sometimes called ***Karatusba multiplication.***
- With divide-and-conquer multiplication, we split each of the numbers into two halves, each with $n/2$ digits.
- Let us call the two numbers we're trying to multiply $\boldsymbol{a}$ and $\boldsymbol{b}$, with the two halves of $\boldsymbol{a}$ being $\boldsymbol{a_L}$ (the left or upper half) and $\boldsymbol{a_R}$ (the right or lower half) and the two halves of $\boldsymbol{b}$ being $\boldsymbol{b_L}$ and $\boldsymbol{b_R}$.

Multiplying Two n Bit Numbers

$$
\begin{array}{rcl}
 & a_L & a_R \\
\times & b_L & b_R \\
\hline
 & a_L b_R & a_R b_R \\
+\ a_L b_L & a_R b_L & \\
\hline
a_L b_L & (a_L b_R + a_R b_L) & a_R b_R
\end{array}
$$

$$
\begin{aligned}
ab &= (a_L\ 10^{n/2} + a_R)\ (b_L\ 10^{n/2} + b_R) \\
&= a_L\ b_L\ 10^{n} + a_L\ b_R\ 10^{n/2} + a_R\ b_L\ 10^{n/2} + a_R\ b_R \\
&= a_L\ b_L\ 10^{n} + (a_L\ b_R + a_R\ b_L)\ 10^{n/2} + a_R\ b_R
\end{aligned}
$$

In order to multiply a pair of *n*-digit numbers, we can recursively multiply four pairs of *n*/2-digit numbers. The rest of the operations involved are all $O(n)$ operations

$$T(n) = 4T(\frac{n}{2}) + O(n)$$

Multiplying Two n Bit Numbers

- If we take a look at the above formula, there are four multiplications of size n/2, so we basically divided the problem of size n into four sub-problems of size n/2. But that doesn't help because solution of recurrence **T(n) = 4T(n/2) + O(n)** is **$O(n^2)$**.
- Suppose we are given a = 123456 and b = 654321. We can rewrite a as 123000+456 and b as 654000+321.
- As a result, a b = 123 654 106 + (123 321 + 456 654) 103 + 456 321.

$$T(n) = aT(n/b) + \Theta(n^k)$$

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

Multiplying Two n Bit Numbers

The idea works as follows: We're trying to compute

$$a_L\ b_L\ 10^n + (a_L\ b_R + a_R\ b_L)\ 10^{n/2} + a_R\ b_R$$

What we'll do is compute the following three products using recursive calls.

$$x_1 = a_L\ b_L$$
$$x_2 = a_R\ b_R$$
$$x_3 = (a_L + a_R)\ (b_L + b_R)$$

These have all the information that we want, since the following is true.

$$x_1\ 10^n + (x_3 - x_1 - x_2)\ 10^{n/2} + x_2$$

T(n) = 3T(n/2) + cn

$$= a_L\ b_L\ 10^n + ((a_L\ b_L + a_L\ b_R + a_R\ b_L + a_R\ b_R) - a_L\ b_L - a_R\ b_R)\ 10^{n/2} + a_R\ b_R$$
$$= a_L\ b_L\ 10^n + (a_L\ b_R + a_R\ b_L)\ 10^{n/2} + a_R\ b_R$$

And we already reason that this last is equal to the product of *a* and *b*.

The divide-and-conquer algorithm in action

Function	Value	Computational Complexity
X_L	6,143	Given
X_R	8,521	Given
Y_L	9,473	Given
Y_R	6,407	Given
$D_1 = X_L - X_R$	−2,378	$O(N)$
$D_2 = Y_R - Y_L$	−3,066	$O(N)$
$X_L Y_L$	58,192,639	$T(N/2)$
$X_R Y_R$	54,594,047	$T(N/2)$
$D_1 D_2$	7,290,948	$T(N/2)$
$D_3 = D_1 D_2 + X_L Y_L + X_R Y_R$	120,077,634	$O(N)$
$X_R Y_R$	54,594,047	Computed above
$D_3 10^4$	1,200,776,340,000	$O(N)$
$X_L Y_L 10^8$	5,819,263,900,000,000	$O(N)$
$X_L Y_L 10^8 + D_3 10^4 + X_R Y_R$	5,820,464,730,934,047	$O(N)$

Multiplying Two n Bit Numbers

$$T(n) = 3T(\frac{n}{2}) + O(n)$$

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_2 n} 3^i A \frac{n}{2^i} \\
&= An \sum_{i=0}^{\log_2 n} (\frac{3}{2})^i \\
&= An \frac{(3/2)^{\log_2 n + 1} - 1}{3/2 - 1} \\
&= O(n \frac{3}{2}^{\log_2 n}) \\
&= O(n * n^{\log_2 3/2}) \\
&= O(n^{\log_2 3}) \\
&= O(n^{1.585})
\end{aligned}$$

Multiplying Two n Bit Numbers

- How long does it take to multiply two n-bit numbers? Grade school algorithm: Runtime **$O(n^2)$**
- Recursive algorithm: How to solve in terms of sub-problem solutions?

$(a\,2^{n/2} + b)(c\,2^{n/2} + d) = ac\,2^n + (ad + bc)2^{n/2} + bd$ ------(1)

$\Rightarrow T(n) = 4T(n/2) + c\,n =$ **$O(n^2)$**

Improvement: substitute **$(a+b)(c+d)$** = $ac + ad + bc + bd$ in equation (1)

$\Rightarrow ac\,2^n + (ad + bc)2^{n/2} + bd$

$= ac\,2^n + (ac + ad + bc + bd)2^{n/2} + bd$

$\Rightarrow T(n) = 3T(n/2) + c\,n =$ **$O(n^{\lg 3})$**

Infinite Wall Problem

- You have an infinite wall on both sides where you are standing and it has a gate somewhere in one direction, you have to find out the gate,

- No design-Infinite time
- Incremental design- you go one step in one direction , come back go to other direction one step, come back and then go 2 steps in other direction.

Infinite Wall Problem: Quadratic time solution

- 1+2.1+1
- 2+2.2+2
- 3+2.3+3
- : : :
- n-1+2.(n-1)+n-1
- n+2n
- $4\sum i+3n$
- $4n(n-1)/2+3n=2n^2+n$

Infinite Wall Problem: Linear time solution

- You go 2^0 step in one direction, come back go 1 step in other then 2^1direction in one way then come back and then up to 2^k
- $2^0+2.2^0+2^0$
- $2^1+2.\ 2^1+2^1$
- …
- $2^{K-1}+2.2^{K-1}+2^{K-1}$
- 3.2^K
- $4(2^{K-1}+2^{K-2}+\ldots\ldots 1)+\ 3.2^K$
- $4(2^{K-1})+\ 3.2^K = 7.2^K -4 = 7N-4$

Questions, Comments and Suggestions

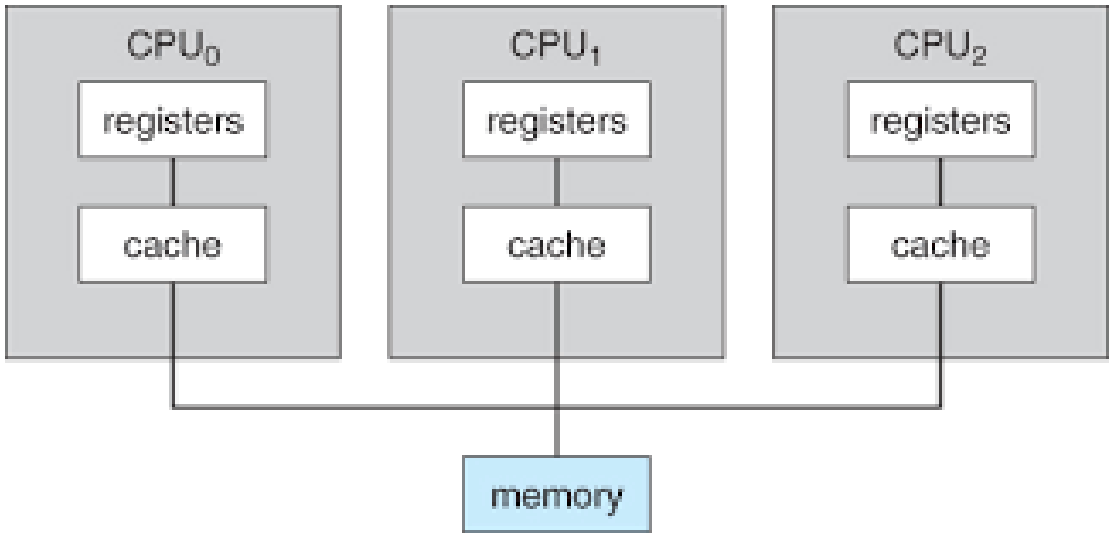
- Divide step is the dominating operation and Combine step is the dominant operation respectively in following
- A) Merge Sort, Quick Sort
- B) Quick Sort, Merge Sort
- C) Bubble Sort, Counting Sort
- D) Radix Sort, Selection Sort

Advantages of Divide and Conquer

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.
- It efficiently uses **cache memory** without occupying much space because it solves simple sub-problems within the cache memory instead of accessing the slower main memory.
- It is more proficient than that of its counterpart **Brute Force** technique.
- Divide & Conquer algorithms are adapted for execution in **multi-processor machines**.
- Since these algorithms **inhibit parallelism**, it does not involve any modification and is handled by systems incorporating parallel processing.

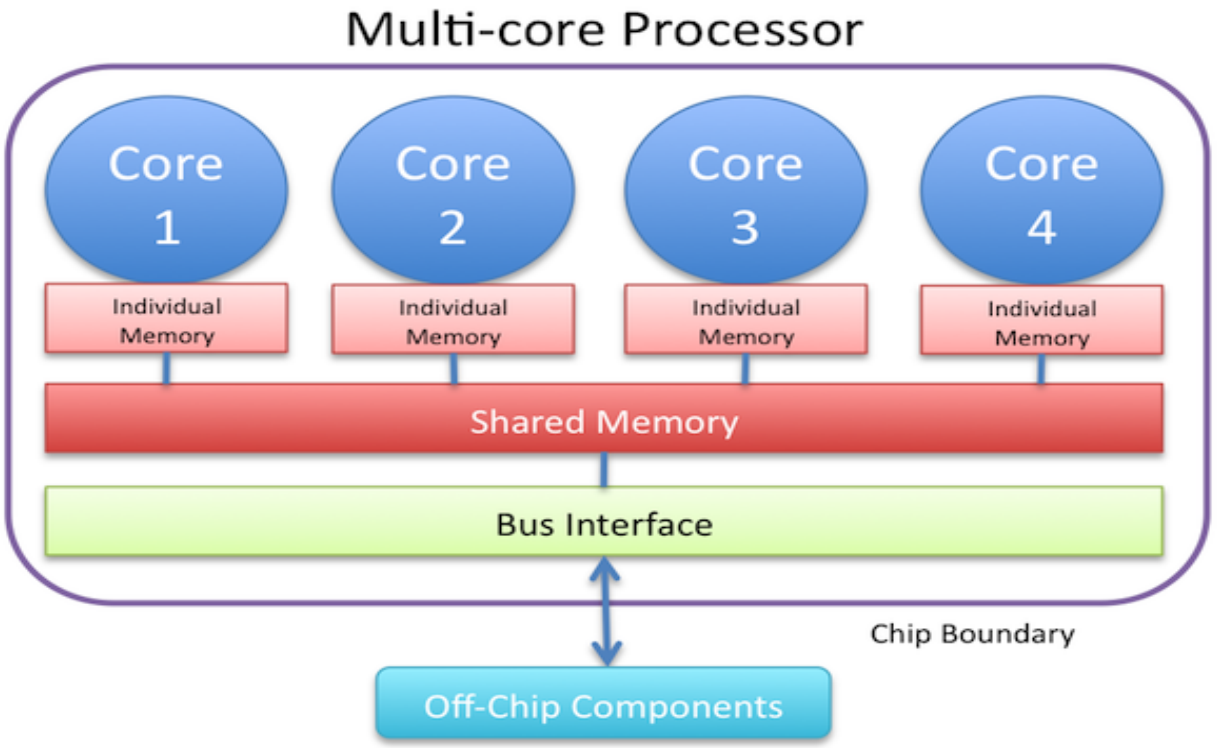
Multi-processor machines.

- A **Multiprocessor** is a computer system with two or more central processing units (CPUs) share full access to a common RAM.
- The main objective of using a **multiprocessor** is to boost the system's execution speed, with other objectives being fault tolerance and application matching



Multi-core processor

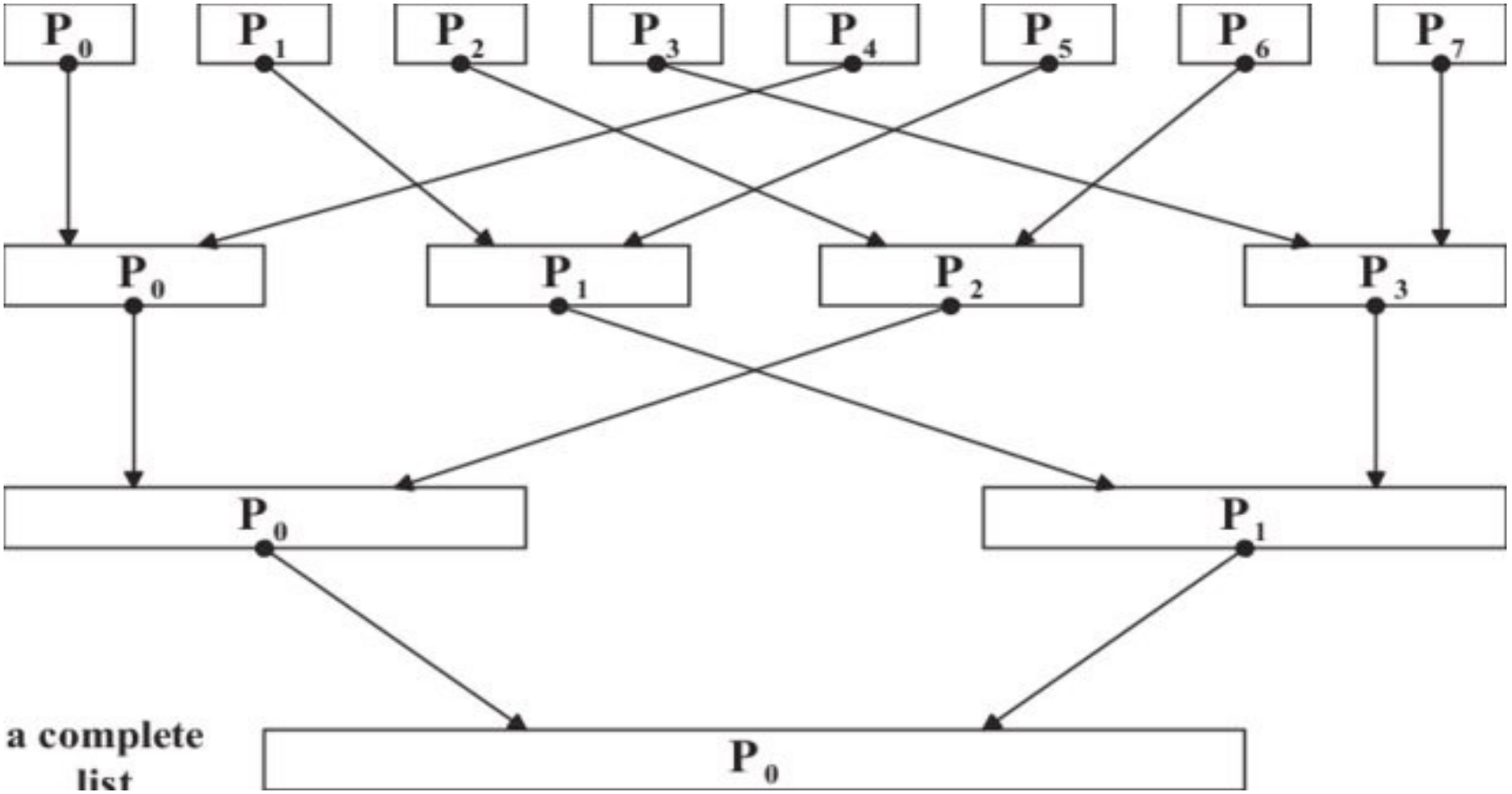
- A **multi-core processor** is a computer **processor** on a single integrated circuit with two or more separate **processing** units, called **cores**, each of which reads and executes program instructions.



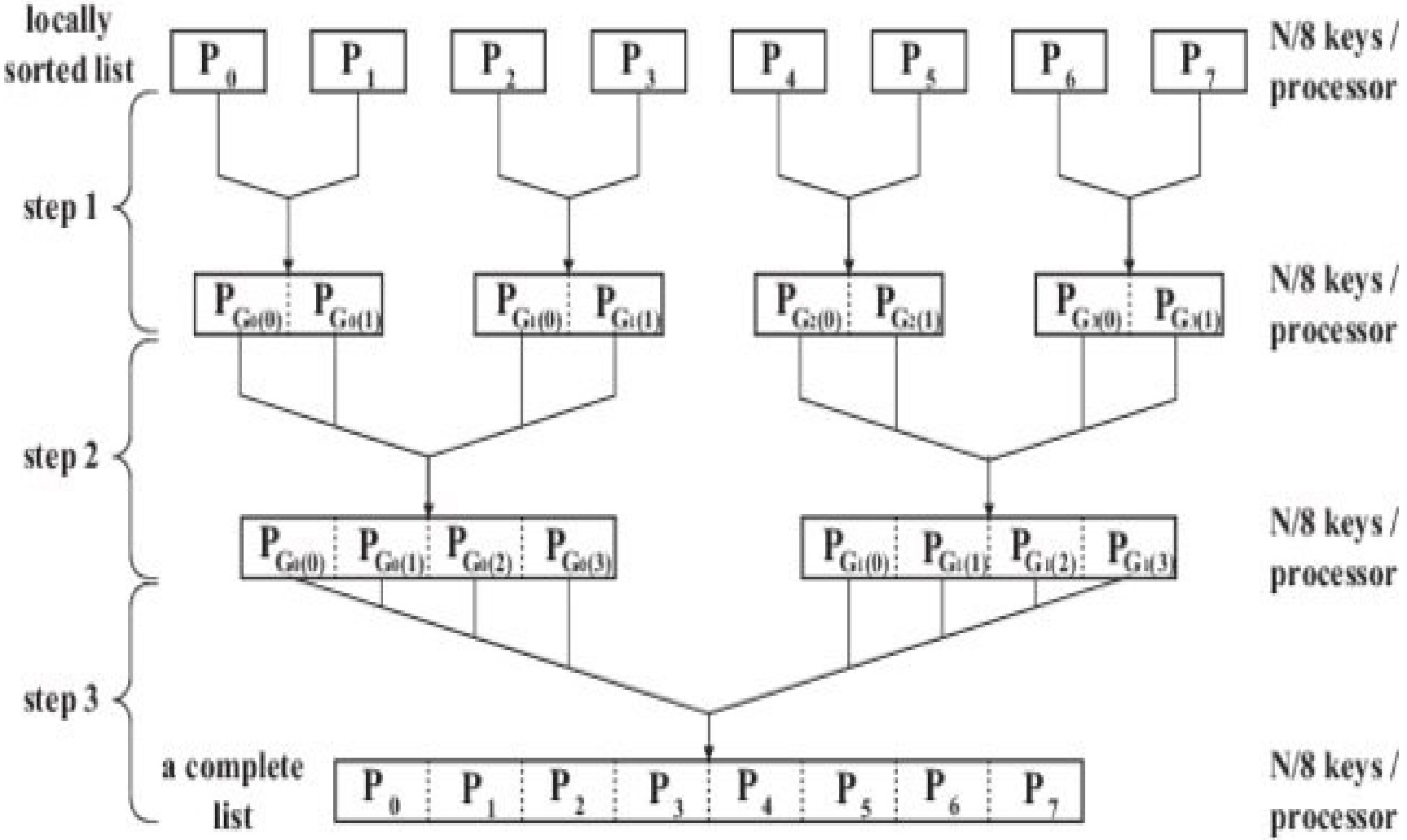
Multi-threading

- In computer architecture, multithreading is the ability of a central processing unit (**CPU**) (or a single **core** in a **multi-core processor**) to provide **multiple threads** of execution concurrently, supported by the operating system.
- **Multi-threading** is way to improve parallelism by running the threads simultaneously in different cores of your processor.
- When multiple **parallel tasks** are executed by a processor, it is known as **multitasking**.
- A CPU scheduler, handles the tasks that execute in parallel.
- The CPU implements tasks using operating system **threads**. So that tasks can execute independently but have some data transfer between them, such as data transfer between a data acquisition module and controller for the system. Data transfer occurs when there is a data dependency.

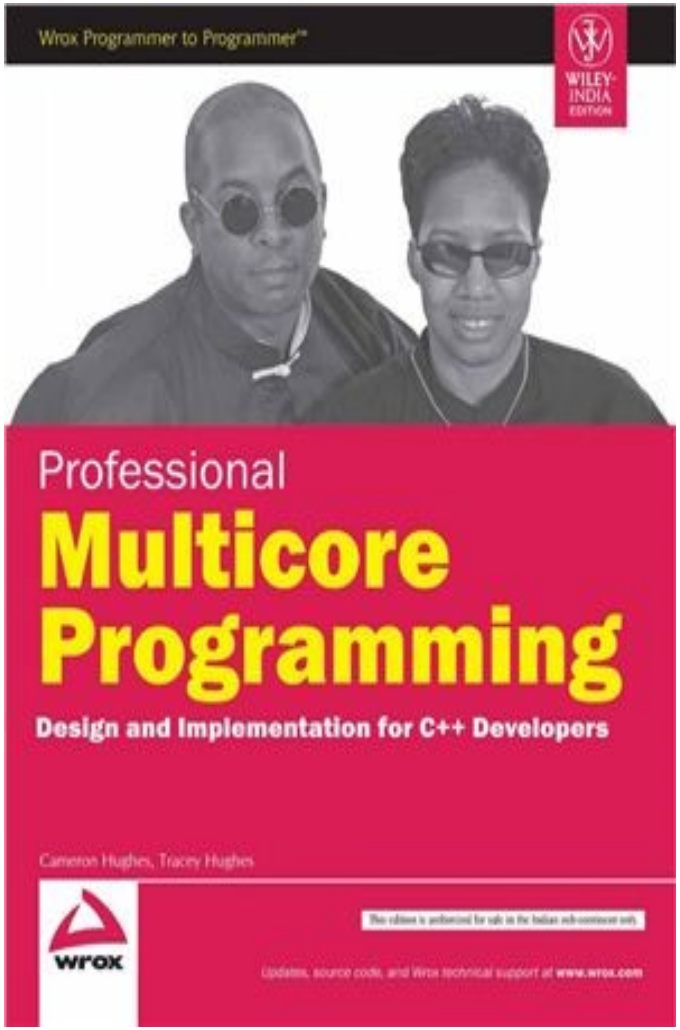
Conventional parallel merge sort with 8 processors.



Conventional parallel merge sort with 8 processors.



Multicore Programming



Professional **Multicore Programming**: Design and Implementation for **C++** Developers by **Hughes & Hughes,** Wrox **Publication**

Disadvantages of Divide and Conquer

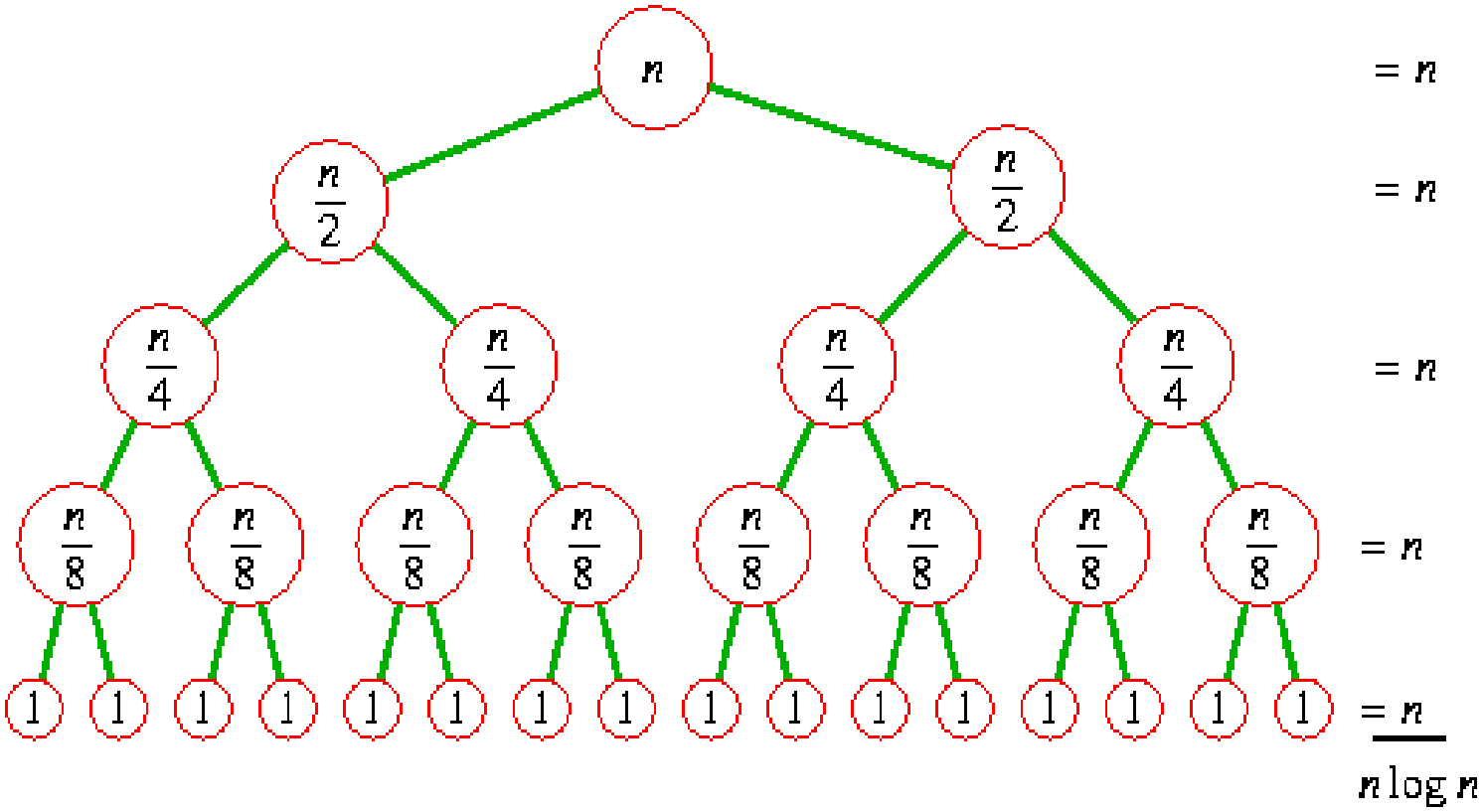
- Recursion is slow. Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
- Very simple problem may be more complicated than an **iterative approach**. Example: adding n numbers etc.
- An explicit stack may overuse the space.
- It may even crash the system if the recursion is performed rigorously greater than the **stack** present in the CPU.

Conclusions

- Divide and conquer means that when you are faced with a large problem, you recursively divide it into smaller versions of the same problem. When you break up a problem in this way, the complexity often decreases exponentially, leading to O(n log n) instead of O(n) time complexity.
- Divide and conquer is just one of several powerful techniques for algorithm design.
- Divide-and-conquer algorithms can be analyzed using recurrences and the master method.
- Can lead to more efficient algorithms.

Divide and Conquer Algorithms

Divide and conquer is a way to break complex problems into smaller problems that are easier to solve, and then combine the answers to solve the original problem.



Thanks for Your Attention!

Home work

1. Given a sorted array of distinct integers A[1…n], you want to find out whether there is an index i for which A[i] = i. Given a divide-and-conquer algorithm that runs in time O(logn).
2. Why Selection Sort can't be converted to Divide and Conquer Mechanism ?
3. What are the deciding factors in determining the termination of further divisions of the problem into sub problems. (Give two at least) .
4. You are given two pan fair balance. You have 12 identically looking coins out of which one coin may be lighter or heavier. How can you find odd coin, if any, in minimum trials, also determine whether defective coin is lighter or heavier, in the worst case?

Home work

5. Finding the Non-Duplicate: You are given a sorted array of numbers where every value except one appears exactly twice; the remaining value appears only once. Design an efficient algorithm for finding which value appears only once.
6. The problem is to tile a board of size $2^k \times 2^k$ with one single tile and $2^{2k} - 1$ L-shaped groups of 3 tiles. A divide-and-conquer approach can recursively divide the board into four, and place a L-grouped set of 3 tiles in the center at the parts that have no extra tile.
7. Find the n^{th} smallest value among 2^n numbers?

Home work

Running Time of a Code Fragment. Consider the following code fragment:

```
b = 1
for i = 1 to n + 1
{
    a = 0
    for k = 2 to ⌊i / 2⌋
    {
        a = a + 1
    }
    b = b * (a + 5)
}
```

In terms of *n, how many additions and multiplications (total) are performed when this code is executed with an* ***even*** input of *n? Simplify your result to a polynomial.*

References

- https://ocw.tudelft.nl/courses/algoritmiek/?view=lectures