# Divide and Conquer Algorithms

T. M. Murali

March 1, 2016

# Divide and Conquer

- Break up a problem into several parts.
- Solve each part recursively.
- Solve base cases by brute force.
- Efficiently combine solutions for sub-problems into final solution.

# Divide and Conquer

- Break up a problem into several parts.
- Solve each part recursively.
- Solve base cases by brute force.
- Efficiently combine solutions for sub-problems into final solution.
- Common use:
  - Partition problem into two equal sub-problems of size $n/2$.
  - Solve each part recursively.
  - Combine the two solutions in $O(n)$ time.
  - Resulting running time is $O(n \log n)$.

# Mergesort

SORT

**INSTANCE:** Nonempty list $L = x_1, x_2, \ldots, x_n$ of integers.

**SOLUTION:** A permutation $y_1, y_2, \ldots, y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$, for all $1 \leq i < n$.

- Mergesort is a divide-and-conquer algorithm for sorting.
  1. Partition $L$ into two lists $A$ and $B$ of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively.
  2. Recursively sort $A$.
  3. Recursively sort $B$.
  4. Merge the sorted lists $A$ and $B$ into a single sorted list.

# Merging Two Sorted Lists

- Merge two sorted lists $A = a_1, a_2, \ldots, a_k$ and $B = b_1, b_2, \ldots b_l$.

  Maintain a *current* pointer for each list.
  Initialise each pointer to the front of the list.
  While both lists are nonempty:

    Let $a_i$ and $b_j$ be the elements pointed to by the *current* pointers.
    Append the smaller of the two to the output list.
    Advance the current pointer in the list that the smaller element belonged to.

  EndWhile
  Append the rest of the non-empty list to the output.

# Merging Two Sorted Lists

- Merge two sorted lists $A = a_1, a_2, \ldots, a_k$ and $B = b_1, b_2, \ldots b_l$.

    Maintain a *current* pointer for each list.
    Initialise each pointer to the front of the list.
    While both lists are nonempty:

        Let $a_i$ and $b_j$ be the elements pointed to by the *current* pointers.
        Append the smaller of the two to the output list.
        Advance the current pointer in the list that the smaller element belonged to.

    EndWhile
    Append the rest of the non-empty list to the output.

- Running time of this algorithm is $O(k + l)$.

# Analysing Mergesort

1. Partition $L$ into two lists $A$ and $B$ of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively.
2. Recursively sort $A$.
3. Recursively sort $B$.
4. Merge the sorted lists $A$ and $B$ into a single sorted list.

# Analysing Mergesort

1. Partition $L$ into two lists $A$ and $B$ of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively.
2. Recursively sort $A$.
3. Recursively sort $B$.
4. Merge the sorted lists $A$ and $B$ into a single sorted list.

Worst-case running time for $n$ elements $\leq$

Worst-case running time for $\lfloor n/2 \rfloor$ elements $+$
Worst-case running time for $\lceil n/2 \rceil$ elements $+$
Time to split the input into two lists $+$
Time to merge two sorted lists.

# Analysing Mergesort

1. Partition $L$ into two lists $A$ and $B$ of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively.
2. Recursively sort $A$.
3. Recursively sort $B$.
4. Merge the sorted lists $A$ and $B$ into a single sorted list.

Worst-case running time for $n$ elements $\leq$

Worst-case running time for $\lfloor n/2 \rfloor$ elements $+$
Worst-case running time for $\lceil n/2 \rceil$ elements $+$
Time to split the input into two lists $+$
Time to merge two sorted lists.

- Assume $n$ is a power of 2.
- Define $T(n) \equiv$ Worst-case running time for $n$ elements, for every $n \geq 1$.

# Analysing Mergesort

1. Partition $L$ into two lists $A$ and $B$ of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively.
2. Recursively sort $A$.
3. Recursively sort $B$.
4. Merge the sorted lists $A$ and $B$ into a single sorted list.

   Worst-case running time for $n$ elements $\leq$
   
   Worst-case running time for $\lfloor n/2 \rfloor$ elements $+$
   
   Worst-case running time for $\lceil n/2 \rceil$ elements $+$
   
   Time to split the input into two lists $+$
   
   Time to merge two sorted lists.

- Assume $n$ is a power of 2.
- Define $T(n) \equiv$ Worst-case running time for $n$ elements, for every $n \geq 1$.

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn, n > 2 \\ T(2) &\leq c \end{aligned}$$

# Analysing Mergesort

1. Partition $L$ into two lists $A$ and $B$ of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively.
2. Recursively sort $A$.
3. Recursively sort $B$.
4. Merge the sorted lists $A$ and $B$ into a single sorted list.

Worst-case running time for $n$ elements $\leq$
   Worst-case running time for $\lfloor n/2 \rfloor$ elements $+$
   Worst-case running time for $\lceil n/2 \rceil$ elements $+$
   Time to split the input into two lists $+$
   Time to merge two sorted lists.

- Assume $n$ is a power of 2.
- Define $T(n) \equiv$ Worst-case running time for $n$ elements, for every $n \geq 1$.

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn, n > 2 \\ T(2) &\leq c \end{aligned}$$

- Three basic ways of solving this recurrence relation:
  1. "Unroll" the recurrence (somewhat informal method).
  2. Guess a solution and substitute into recurrence to check.
  3. Guess solution in $O()$ form and substitute into recurrence to determine the constants.

# Unrolling the recurrence

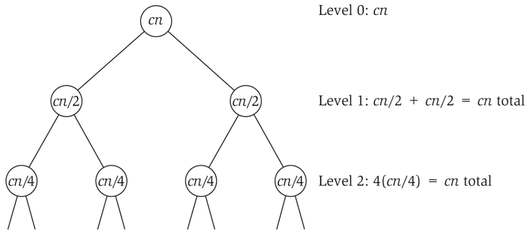


**Figure 5.1** Unrolling the recurrence $T(n) \leq 2T(n/2) + O(n)$.

# Unrolling the recurrence

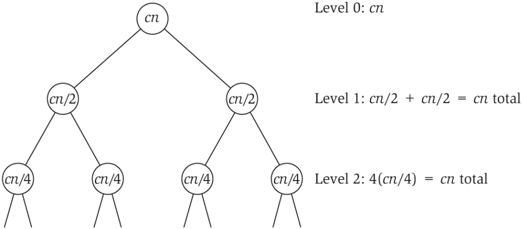


**Figure 5.1** Unrolling the recurrence $T(n) \le 2T(n/2) + O(n)$.

- Recursion tree has $\log n$ levels.
- Total work done at each level is $cn$.
- Running time of the algorithm is $cn \log n$.
- Use this method only to get an idea of the solution.

# Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.

# Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
- Base case: $n = 2$. Is $T(2) = c \leq 2c \log 2$? Yes.

# Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
- Base case: $n = 2$. Is $T(2) = c \leq 2c \log 2$? Yes.
- (Strong) Inductive hypothesis: assume $T(m) \leq cm \log_2 m$ for all $m < n$.

# Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
- Base case: $n = 2$. Is $T(2) = c \leq 2c \log 2$? Yes.
- (Strong) Inductive hypothesis: assume $T(m) \leq cm \log_2 m$ for all $m < n$. Therefore, $T(n/2) \leq (cn/2) \log(n/2)$.

# Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
- Base case: $n = 2$. Is $T(2) = c \leq 2c \log 2$? Yes.
- (Strong) Inductive hypothesis: assume $T(m) \leq cm \log_2 m$ for all $m < n$. Therefore, $T(n/2) \leq (cn/2) \log(n/2)$.
- Inductive step: Prove $T(n) \leq cn \log n$.

# Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
- Base case: $n = 2$. Is $T(2) = c \leq 2c \log 2$? Yes.
- (Strong) Inductive hypothesis: assume $T(m) \leq cm \log_2 m$ for all $m < n$. Therefore, $T(n/2) \leq (cn/2) \log(n/2)$.
- Inductive step: Prove $T(n) \leq cn \log n$.

$$
\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\
&\leq 2\left(\frac{cn}{2} \log\left(\frac{n}{2}\right)\right) + cn, \text{ by the inductive hypothesis} \\
&= cn \log\left(\frac{n}{2}\right) + cn \\
&= cn \log n - cn + cn \\
&= cn \log n.
\end{aligned}
$$

# Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \leq cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
- Base case: $n = 2$. Is $T(2) = c \leq 2c \log 2$? Yes.
- (Strong) Inductive hypothesis: assume $T(m) \leq cm \log_2 m$ for all $m < n$. Therefore, $T(n/2) \leq (cn/2) \log(n/2)$.
- Inductive step: Prove $T(n) \leq cn \log n$.

$$
\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\
&\leq 2\left(\frac{cn}{2} \log\left(\frac{n}{2}\right)\right) + cn, \text{ by the inductive hypothesis} \\
&= cn \log\left(\frac{n}{2}\right) + cn \\
&= cn \log n - cn + cn \\
&= cn \log n.
\end{aligned}
$$

- Why is $T(n) \leq kn^2$ a "loose" bound?

# Substituting a Solution into the Recurrence

- Guess that the solution is $T(n) \le cn \log n$ (logarithm to the base 2).
- Use induction to check if the solution satisfies the recurrence relation.
- Base case: $n = 2$. Is $T(2) = c \le 2c \log 2$? Yes.
- (Strong) Inductive hypothesis: assume $T(m) \le cm \log_2 m$ for all $m < n$. Therefore, $T(n/2) \le (cn/2) \log(n/2)$.
- Inductive step: Prove $T(n) \le cn \log n$.

$$\begin{aligned} T(n) &\le 2T\left(\frac{n}{2}\right) + cn \\ &\le 2\left(\frac{cn}{2} \log\left(\frac{n}{2}\right)\right) + cn, \text{ by the inductive hypothesis} \\ &= cn \log\left(\frac{n}{2}\right) + cn \\ &= cn \log n - cn + cn \\ &= cn \log n. \end{aligned}$$

- Why is $T(n) \le kn^2$ a “loose” bound?
- Why doesn’t an attempt to prove $T(n) \le kn$, for some $k > 0$ work?

# Partial Substitution

- Guess that the solution is $kn \log n$ (logarithm to the base 2).
- Substitute guess into the recurrence relation to check what value of $k$ will satisfy the recurrence relation.

# Partial Substitution

- Guess that the solution is $kn \log n$ (logarithm to the base 2).
- Substitute guess into the recurrence relation to check what value of $k$ will satisfy the recurrence relation.
- $k \geq c$ will work.

# Proof for All Values of $n$

- We assumed $n$ is a power of 2.
- How do we generalise the proof?

# Proof for All Values of $n$

- We assumed $n$ is a power of 2.
- How do we generalise the proof?
- Basic axiom: $T(n) \leq T(n+1)$, for all $n$: worst case running time increases as input size increases.
- Let $m$ be the smallest power of 2 larger than $n$.
- $T(n) \leq T(m) = O(m \log m)$

# Proof for All Values of $n$

- We assumed $n$ is a power of 2.
- How do we generalise the proof?
- Basic axiom: $T(n) \leq T(n+1)$, for all $n$: worst case running time increases as input size increases.
- Let $m$ be the smallest power of 2 larger than $n$.
- $T(n) \leq T(m) = O(m \log m) = O(n \log n)$, because $m \leq 2n$.

# Other Recurrence Relations

- Divide into $q$ sub-problems of size $n/2$ and merge in $O(n)$ time. Two distinct cases: $q = 1$ and $q > 2$.
- Divide into two sub-problems of size $n/2$ and merge in $O(n^2)$ time.

# $T(n) = qT(n/2) + cn, q = 1$

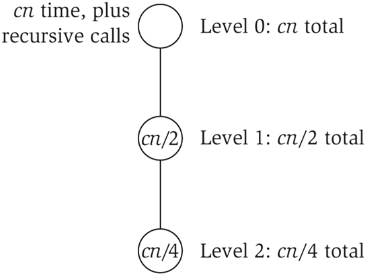


**Figure 5.3** Unrolling the recurrence $T(n) \leq T(n/2) + O(n)$.

# $T(n) = qT(n/2) + cn, q = 1$

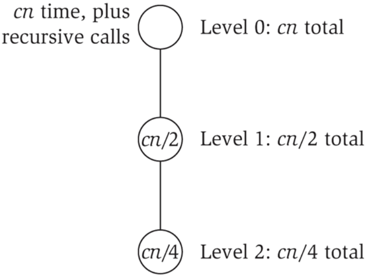


**Figure 5.3** Unrolling the recurrence $T(n) \leq T(n/2) + O(n)$.

- Each invocation reduces the problem size by a factor of 2 $\Rightarrow$ there are $\log n$ levels in the recursion tree.
- At level $i$ of the tree, the problem size is $n/2^i$ and the work done is $cn/2^i$.
- Therefore, the total work done is

$$\sum_{i=0}^{i=\log n} \frac{cn}{2^i} = O(n).$$

# $T(n) = qT(n/2) + cn, q > 2$

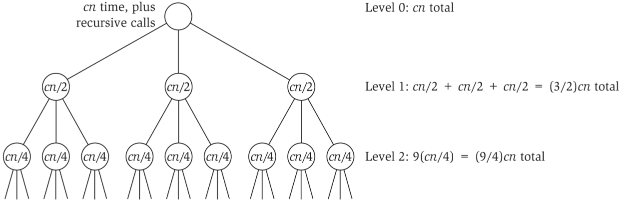


**Figure 5.2** Unrolling the recurrence $T(n) \leq 3T(n/2) + O(n)$.

# $T(n) = qT(n/2) + cn, q > 2$

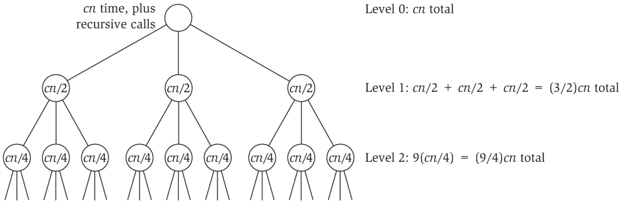


**Figure 5.2** Unrolling the recurrence $T(n) \leq 3T(n/2) + O(n)$.

- There are $\log n$ levels in the recursion tree.
- At level $i$ of the tree, there are $q^i$ sub-problems, each of size $n/2^i$.
- The total work done at level $i$ is $q^i cn/2^i$. Therefore, the total work done is

$$T(n) \quad \leq \quad \sum_{i=0}^{i=\log_2 n} q^i \frac{cn}{2^i} \leq$$

# $T(n) = qT(n/2) + cn, q > 2$

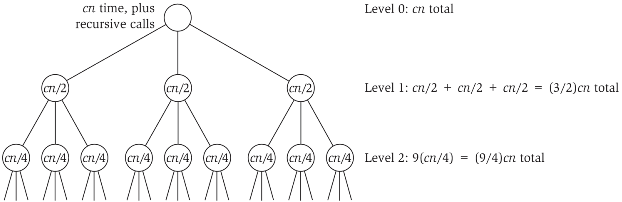


**Figure 5.2** Unrolling the recurrence $T(n) \leq 3T(n/2) + O(n)$.

- There are $\log n$ levels in the recursion tree.
- At level $i$ of the tree, there are $q^i$ sub-problems, each of size $n/2^i$.
- The total work done at level $i$ is $q^i cn/2^i$. Therefore, the total work done is

$$
\begin{aligned}
T(n) &\leq \sum_{i=0}^{i=\log_2 n} q^i \frac{cn}{2^i} \leq cn \sum_{i=0}^{i=\log_2 n} \left(\frac{q}{2}\right)^i \\
&= O\left(cn \left(\frac{q}{2}\right)^{\log_2 n}\right) = O\left(cn \left(\frac{q}{2}\right)^{(\log_{q/2} n)(\log_2 q/2)}\right) \\
&= O(cn\, n^{\log_2 q/2}) = O(n^{\log_2 q}).
\end{aligned}
$$

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

- Total work done is

$$\sum_{i=0}^{i=\log n} 2^i \left(\frac{cn}{2^i}\right)^2 \leq$$

## $T(n) = 2T(n/2) + cn^2$

- Total work done is

$$\sum_{i=0}^{i=\log n} 2^i \left(\frac{cn}{2^i}\right)^2 \leq O(n^2).$$