Analysis of Algorithms

T. M. Murali

April 25, May 2, 2011

Problem Example

FIND MINIMUM

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

SOLUTION: Pair (i, x_i) such that $x_i = \min\{x_j \mid 1 \le j \le n\}$.

Algorithm Example

```
FIND-MINIMUM(x_1, x_2, ..., x_n)
    i ← 1
    for j ← 2 to n
        do if x_j < x_i
            then i ← j
    return (i, x_i)
```

Running Time of Algorithm

```
FIND-MINIMUM(x_1, x_2, ..., x_n)
    i ← 1
    for j ← 2 to n
        do if x_j < x_i
            then i ← j
    return (i, x_i)
```

Running Time of Algorithm

```
FIND-MINIMUM(x_1, x_2, ..., x_n)
    i ← 1
    for j ← 2 to n
        do if x_j < x_i
            then i ← j
    return (i, x_i)
```

- At most $2n - 1$ assignments and $n - 1$ comparisons.

Correctness of Algorithm: Proof 1

```
FIND-MINIMUM(x_1, x_2, ..., x_n)
    i ← 1
    for j ← 2 to n
        do if x_j < x_i
            then i ← j
    return (i, x_i)
```

Correctness of Algorithm: Proof 1

```
FIND-MINIMUM(x_1, x_2, ..., x_n)
    i ← 1
    for j ← 2 to n
        do if x_j < x_i
              then i ← j
    return (i, x_i)
```

- Proof by contradiction:

Correctness of Algorithm: Proof 1

```
FIND-MINIMUM(x_1, x_2, ..., x_n)
    i ← 1
    for j ← 2 to n
        do if x_j < x_i
               then i ← j
    return (i, x_i)
```

- Proof by contradiction: Suppose algorithm returns (k, x_k) but there exists $1 \leq l \leq n$ such that $x_l < x_k$ and $x_l = \min\{x_j \mid 1 \leq j \leq n\}$.

Correctness of Algorithm: Proof 1

```
FIND-MINIMUM(x_1, x_2, ..., x_n)
    i ← 1
    for j ← 2 to n
        do if x_j < x_i
               then i ← j
    return (i, x_i)
```

- Proof by contradiction: Suppose algorithm returns (k, x_k) but there exists $1 \leq l \leq n$ such that $x_l < x_k$ and $x_l = \min\{x_j \mid 1 \leq j \leq n\}$.
- Is $k < l$?

Correctness of Algorithm: Proof 1

Find-Minimum$(x_1, x_2, \ldots, x_n)$

$i \leftarrow 1$
for $j \leftarrow 2$ **to** n
    do if $x_j < x_i$
        then $i \leftarrow j$
return (i, x_i)

- Proof by contradiction: Suppose algorithm returns (k, x_k) but there exists $1 \leq l \leq n$ such that $x_l < x_k$ and $x_l = \min\{x_j \mid 1 \leq j \leq n\}$.
- Is $k < l$? No. Since the algorithm returns (k, x_k), $x_k \leq x_j$, for all $k < j \leq n$. Therefore $l < k$.

Correctness of Algorithm: Proof 1

```
FIND-MINIMUM(x_1, x_2, ..., x_n)
    i ← 1
    for j ← 2 to n
        do if x_j < x_i
              then i ← j
    return (i, x_i)
```

- Proof by contradiction: Suppose algorithm returns (k, x_k) but there exists $1 \leq l \leq n$ such that $x_l < x_k$ and $x_l = \min\{x_j \mid 1 \leq j \leq n\}$.
- Is $k < l$? No. Since the algorithm returns (k, x_k), $x_k \leq x_j$, for all $k < j \leq n$. Therefore $l < k$.
- What does the algorithm do when $j = l$? *It must set i to l*, since we have been told that x_l is the smallest element.
- What does the algorithm do when $j = k$ (which happens after $j = l$)? Since $x_l < x_k$, the value of i does not change.
- Therefore, the algorithm does not return (k, x_k) yielding a contradiction.

What is Algorithm Analysis?

- Measure resource requirements: how do the amount of time and space that an algorithm uses scale with increasing input size?
- How do we put this notion on a concrete footing?
- What does it mean for one function to grow faster or slower than another?

What is Algorithm Analysis?

- Measure resource requirements: how do the amount of time and space that an algorithm uses scale with increasing input size?
- How do we put this notion on a concrete footing?
- What does it mean for one function to grow faster or slower than another?
- Goal: Develop algorithms that provably run quickly and use low amounts of space.

Worst-case Running Time

- We will measure worst-case running time of an algorithm.
  - Avoid depending on test cases or sample runs.
- Bound the largest possible running time the algorithm over all inputs of size n, as a function of n.

Worst-case Running Time

- We will measure worst-case running time of an algorithm.
  - Avoid depending on test cases or sample runs.
- Bound the largest possible running time the algorithm over all inputs of size n, as a function of n.
- Why worst-case? Why not average-case or on random inputs?

Worst-case Running Time

- We will measure worst-case running time of an algorithm.
  - Avoid depending on test cases or sample runs.
- Bound the largest possible running time the algorithm over all inputs of size n, as a function of n.
- Why worst-case? Why not average-case or on random inputs?
- *Input size* = number of elements in the input.

Worst-case Running Time

- We will measure worst-case running time of an algorithm.
  - Avoid depending on test cases or sample runs.
- Bound the largest possible running time the algorithm over all inputs of size n, as a function of n.
- Why worst-case? Why not average-case or on random inputs?
- *Input size* = number of elements in the input. Values in the input do not matter.
- Assume all elementary operations take unit time: assignment, arithmetic on a fixed-size number, comparisons, array lookup, following a pointer, etc.
  - Make analysis independent of hardware and software.

Polynomial Time

- Brute force algorithm: Check every possible solution.

Polynomial Time

- Brute force algorithm: Check every possible solution.
- What is a brute force algorithm for sorting: given n numbers, permute them so that they appear in increasing order?

Polynomial Time

- Brute force algorithm: Check every possible solution.
- What is a brute force algorithm for sorting: given n numbers, permute them so that they appear in increasing order?
  - Try all possible $n!$ permutations of the numbers.
  - For each permutation, check if it is sorted.

Polynomial Time

- Brute force algorithm: Check every possible solution.
- What is a brute force algorithm for sorting: given n numbers, permute them so that they appear in increasing order?
  - Try all possible $n!$ permutations of the numbers.
  - For each permutation, check if it is sorted.
  - Running time is $nn!$. Unacceptable in practice!

Polynomial Time

- Brute force algorithm: Check every possible solution.
- What is a brute force algorithm for sorting: given n numbers, permute them so that they appear in increasing order?
  - Try all possible $n!$ permutations of the numbers.
  - For each permutation, check if it is sorted.
  - Running time is $nn!$. Unacceptable in practice!
- Desirable scaling property: when the input size doubles, the algorithm should only slow down by some constant factor k.

Polynomial Time

- Brute force algorithm: Check every possible solution.
- What is a brute force algorithm for sorting: given n numbers, permute them so that they appear in increasing order?
  - Try all possible $n!$ permutations of the numbers.
  - For each permutation, check if it is sorted.
  - Running time is $nn!$. Unacceptable in practice!
- Desirable scaling property: when the input size doubles, the algorithm should only slow down by some constant factor k.
- An algorithm has a *polynomial* running time if there exist constants $c > 0$ and $d > 0$ such that on every input of size n, the running time of the algorithm is bounded by cn^d steps.

Polynomial Time

- Brute force algorithm: Check every possible solution.
- What is a brute force algorithm for sorting: given n numbers, permute them so that they appear in increasing order?
  - Try all possible $n!$ permutations of the numbers.
  - For each permutation, check if it is sorted.
  - Running time is $nn!$. Unacceptable in practice!
- Desirable scaling property: when the input size doubles, the algorithm should only slow down by some constant factor k.
- An algorithm has a *polynomial* running time if there exist constants $c > 0$ and $d > 0$ such that on every input of size n, the running time of the algorithm is bounded by cn^d steps.

Definition

An algorithm is *efficient* if it has a polynomial running time.

Upper and Lower Bounds

- Express "$4n^2 + 100$ does not grow faster than n^2."
- Express "$n^2/4$ grows faster than $n + 1,000,000$."

Upper and Lower Bounds

- Express "$4n^2 + 100$ does not grow faster than n^2."
- Express "$n^2/4$ grows faster than $n + 1,000,000$."

Definition
Asymptotic upper bound: A function $f(n)$ is $O(g(n))$ if
we have $f(n) \leq \ \ g(n)$.

Upper and Lower Bounds

- Express "$4n^2 + 100$ does not grow faster than n^2."
- Express "$n^2/4$ grows faster than $n + 1,000,000$."

Definition

Asymptotic upper bound: A function $f(n)$ is $O(g(n))$ if there exists constant $c > 0$ such that we have $f(n) \leq cg(n)$.

Upper and Lower Bounds

- Express "$4n^2 + 100$ does not grow faster than n^2."
- Express "$n^2/4$ grows faster than $n + 1,000,000$."

Definition

Asymptotic upper bound: A function $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $f(n) \leq cg(n)$.

Upper and Lower Bounds

- Express "$4n^2 + 100$ does not grow faster than n^2."
- Express "$n^2/4$ grows faster than $n + 1,000,000$."

Definition

Asymptotic upper bound: A function $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $f(n) \leq cg(n)$.

Definition

Asymptotic lower bound: A function $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $f(n) \geq cg(n)$.

Upper and Lower Bounds

- Express "$4n^2 + 100$ does not grow faster than n^2."
- Express "$n^2/4$ grows faster than $n + 1,000,000$."

Definition

Asymptotic upper bound: A function $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $f(n) \leq cg(n)$.

Definition

Asymptotic lower bound: A function $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $f(n) \geq cg(n)$.

Definition

Asymptotic tight bound: A function $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Upper and Lower Bounds

- Express "$4n^2 + 100$ does not grow faster than n^2."
- Express "$n^2/4$ grows faster than $n + 1,000,000$."

Definition
Asymptotic upper bound: A function $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $f(n) \leq cg(n)$.

Definition
Asymptotic lower bound: A function $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $f(n) \geq cg(n)$.

Definition
Asymptotic tight bound: A function $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

- In these definitions, c is a constant independent of n.

Upper and Lower Bounds

- Express "$4n^2 + 100$ does not grow faster than n^2."
- Express "$n^2/4$ grows faster than $n + 1,000,000$."

Definition
Asymptotic upper bound: A function $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $f(n) \leq cg(n)$.

Definition
Asymptotic lower bound: A function $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $f(n) \geq cg(n)$.

Definition
Asymptotic tight bound: A function $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

- In these definitions, c is a constant independent of n.
- Abuse of notation: say $g(n) = O(f(n))$, $g(n) = \Omega(f(n))$, $g(n) = \Theta(f(n))$.

Examples

- $f(n) = pn^2 + qn + r$ is

Examples

- $f(n) = pn^2 + qn + r$ is $\theta(n^2)$. Can ignore lower order terms.

Examples

- $f(n) = pn^2 + qn + r$ is $\theta(n^2)$. Can ignore lower order terms.
- Is $f(n) = pn^2 + qn + r = O(n^3)$?

Examples

- $f(n) = pn^2 + qn + r$ is $\theta(n^2)$. Can ignore lower order terms.
- Is $f(n) = pn^2 + qn + r = O(n^3)$?
- $f(n) = \sum_{0 \leq i \leq d} a_i n^i =$

Examples

- $f(n) = pn^2 + qn + r$ is $\theta(n^2)$. Can ignore lower order terms.
- Is $f(n) = pn^2 + qn + r = O(n^3)$?
- $f(n) = \sum_{0 \leq i \leq d} a_i n^i = O(n^d)$, if $d > 0$ is an integer constant and $a_d > 0$.
  - $O(n^d)$ is the definition of polynomial time.

Examples

- $f(n) = pn^2 + qn + r$ is $\theta(n^2)$. Can ignore lower order terms.
- Is $f(n) = pn^2 + qn + r = O(n^3)$?
- $f(n) = \sum_{0 \leq i \leq d} a_i n^i = O(n^d)$, if $d > 0$ is an integer constant and $a_d > 0$.
  - $O(n^d)$ is the definition of polynomial time.
- Is an algorithm with running time $O(n^{1.59})$ a polynomial-time algorithm?

Examples

- $f(n) = pn^2 + qn + r$ is $\theta(n^2)$. Can ignore lower order terms.
- Is $f(n) = pn^2 + qn + r = O(n^3)$?
- $f(n) = \sum_{0 \leq i \leq d} a_i n^i = O(n^d)$, if $d > 0$ is an integer constant and $a_d > 0$.
  - $O(n^d)$ is the definition of polynomial time.
- Is an algorithm with running time $O(n^{1.59})$ a polynomial-time algorithm?
- $O(\log_a n) = O(\log_b n)$ for any pair of constants $a, b > 1$.
- For every $x > 0$, $\log n = O(n^x)$.

Examples

- $f(n) = pn^2 + qn + r$ is $\theta(n^2)$. Can ignore lower order terms.
- Is $f(n) = pn^2 + qn + r = O(n^3)$?
- $f(n) = \sum_{0 \leq i \leq d} a_i n^i = O(n^d)$, if $d > 0$ is an integer constant and $a_d > 0$.
  - $O(n^d)$ is the definition of polynomial time.
- Is an algorithm with running time $O(n^{1.59})$ a polynomial-time algorithm?
- $O(\log_a n) = O(\log_b n)$ for any pair of constants $a, b > 1$.
- For every $x > 0$, $\log n = O(n^x)$.
- For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$.

Properties of Asymptotic Growth Rates

Transitivity

- If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.

Properties of Asymptotic Growth Rates

Transitivity

- If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.

Additivity

- If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$.
- Similar statements hold for lower and tight bounds.

Properties of Asymptotic Growth Rates

Transitivity

- If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.

Additivity

- If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$.
- Similar statements hold for lower and tight bounds.
- If k is a constant and there are k functions $f_i = O(h), 1 \leq i \leq k$,

Properties of Asymptotic Growth Rates

Transitivity

- If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.

Additivity

- If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$.
- Similar statements hold for lower and tight bounds.
- If k is a constant and there are k functions $f_i = O(h), 1 \leq i \leq k$, then $f_1 + f_2 + \ldots + f_k = O(h)$.

Properties of Asymptotic Growth Rates

Transitivity

- If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.

Additivity

- If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$.
- Similar statements hold for lower and tight bounds.
- If k is a constant and there are k functions $f_i = O(h), 1 \leq i \leq k$, then $f_1 + f_2 + \ldots + f_k = O(h)$.
- If $f = O(g)$, then $f + g =$

Properties of Asymptotic Growth Rates

Transitivity

- If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.

Additivity

- If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$.
- Similar statements hold for lower and tight bounds.
- If k is a constant and there are k functions $f_i = O(h), 1 \leq i \leq k$, then $f_1 + f_2 + \ldots + f_k = O(h)$.
- If $f = O(g)$, then $f + g = \Theta(g)$.

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 $B = b_1, b_2, \ldots b_l$.
  1. Maintain a *current* pointer for each list.
  2. Initialise each pointer to the front of its list.
  3. While both lists are nonempty:
     - 3.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     - 3.2 Append the smaller of the two to the output list.
     - 3.3 Advance the current pointer in the list that the smaller element belonged to.
  4. 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 $B = b_1, b_2, \ldots b_l$.
  1. Maintain a *current* pointer for each list.
  2. Initialise each pointer to the front of its list.
  3. While both lists are nonempty:
     3.1 Let a_i and b_j be the elements pointed to by the *current* pointers.
     3.2 Append the smaller of the two to the output list.
     3.3 Advance the current pointer in the list that the smaller element belonged to.
  4. 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 ($T(n)$) $\leq$

   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.

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 $(T(n)) \leq$

   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.

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

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 $(T(n)) \leq$

   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.

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

- Three 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

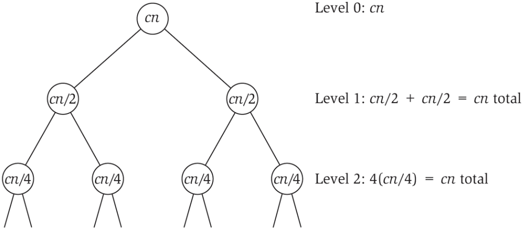

Figure 5.1 Unrolling the recurrence $T(n) \leq 2T(n/2) + O(n)$.

Unrolling the recurrence

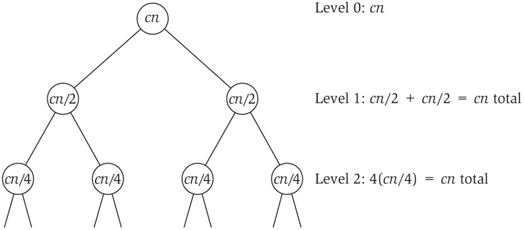

Figure 5.1 Unrolling the recurrence $T(n) \leq 2T(n/2) + O(n)$.

- 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.
- 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.
- 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.
- 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.
- 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 doesn't an attempt to prove $T(n) \leq kn$, for some $k > 0$ work?
- Why is $T(n) \leq kn^2$ a "loose" bound?

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 $m \leq 2n$.