Analysis of Algorithms

August 28, 2014

Problem Example

Find Minimum

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

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

Algorithm Example

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)

Running Time of Algorithm

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

```
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 (a, x_a) but there exists $1 \leq c \leq n$ such that $x_c < x_a$ and $x_c = \min\{x_j \mid 1 \leq j \leq n\}$.

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 (a, x_a) but there exists $1 \leq c \leq n$ such that $x_c < x_a$ and $x_c = \min\{x_j \mid 1 \leq j \leq n\}$.
- Is $a < c$?

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 (a, x_a) but there exists $1 \leq c \leq n$ such that $x_c < x_a$ and $x_c = \min\{x_j \mid 1 \leq j \leq n\}$.
- Is $a < c$? No. Since the algorithm returns (a, x_a), $x_a \leq x_j$, for all $a < j \leq n$. Therefore $c < a$.

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

Correctness of Algorithm: Proof 2

```
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 induction: What is true at the end of each iteration?

Correctness of Algorithm: Proof 2

```
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 induction: What is true at the end of each iteration?
- Claim: At the end of iteration j, $x_i = \min\{x_m \mid 1 \leq m \leq j\}$, for all $1 \leq j \leq n$.
- Claim is true

Correctness of Algorithm: Proof 2

```
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 induction: What is true at the end of each iteration?
- Claim: At the end of iteration j, $x_i = \min\{x_m \mid 1 \leq m \leq j\}$, for all $1 \leq j \leq n$.
- Claim is true $\Rightarrow$ algorithm is correct (set $j = n$).

Correctness of Algorithm: Proof 2

```
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 induction: What is true at the end of each iteration?
- Claim: At the end of iteration j, $x_i = \min\{x_m \mid 1 \leq m \leq j\}$, for all $1 \leq j \leq n$.
- Claim is true $\Rightarrow$ algorithm is correct (set $j = n$).
- Proof of the claim involves three steps.

1. Base case: $j = 1$ (before loop). $x_i = \min\{x_m \mid 1 \leq m \leq 1\}$ is trivially true.
2. Inductive hypothesis: Assume $x_i = \min\{x_m \mid 1 \leq m \leq j\}$.
3. Inductive step: Prove $x_i = \min\{x_m \mid 1 \leq m \leq j + 1\}$.
   - In the loop, i is set to be $j + 1$ if and only if $x_{j+1} < x_i$.
   - Therefore, x_i is the smallest of $x_1, x_2, \ldots, x_{j+1}$ when the loop ends.

Format of Proof by Induction

- Goal: prove some proposition $P(n)$ is true for all n.
- Strategy: prove base case, assume inductive hypothesis, prove inductive step.

Format of Proof by Induction

- Goal: prove some proposition $P(n)$ is true for all n.
- Strategy: prove base case, assume inductive hypothesis, prove inductive step.
- Base case: prove that $P(1)$ or $P(2)$ (or $P(\text{small number})$) is true.
- Inductive hypothesis: assume $P(k-1)$ is true.
- Inductive step: prove that $P(k-1) \Rightarrow P(k)$.

Format of Proof by Induction

- Goal: prove some proposition $P(n)$ is true for all n.
- Strategy: prove base case, assume inductive hypothesis, prove inductive step.
- Base case: prove that $P(1)$ or $P(2)$ (or $P(\text{small number})$) is true.
- Inductive hypothesis: assume $P(k-1)$ is true.
- Inductive step: prove that $P(k-1) \Rightarrow P(k)$.
- Why does this strategy work?

Sum of first n natural numbers

$$P(n) = \sum_{i=1}^{n} i =$$

Sum of first n natural numbers

$$P(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Sum of first n natural numbers

$$P(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Proof by Induction:

- Base case:

Sum of first n natural numbers

$$P(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Proof by Induction:

- Base case: $k = 1$: $P(1) = 1 = 1 \times 2/2$.

Sum of first n natural numbers

$$P(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Proof by Induction:

- Base case: $k = 1$: $P(1) = 1 = 1 \times 2/2$.
- Inductive hypothesis:

Sum of first n natural numbers

$$P(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Proof by Induction:

- Base case: $k = 1$: $P(1) = 1 = 1 \times 2/2$.
- Inductive hypothesis: assume $P(k) = k(k+1)/2$.

Sum of first n natural numbers

$$P(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Proof by Induction:

- Base case: $k = 1$: $P(1) = 1 = 1 \times 2/2$.
- Inductive hypothesis: assume $P(k) = k(k+1)/2$.
- Inductive step: Assuming $P(k) = k(k+1)/2$, prove that $P(k+1) = (k+1)(k+2)/2$

Sum of first n natural numbers

$$P(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Proof by Induction:

- Base case: $k = 1$: $P(1) = 1 = 1 \times 2/2$.
- Inductive hypothesis: assume $P(k) = k(k+1)/2$.
- Inductive step: Assuming $P(k) = k(k+1)/2$, prove that $P(k+1) = (k+1)(k+2)/2$

$$P(k+1) \quad = \quad \sum_{i=1}^{k+1} i =$$

Sum of first n natural numbers

$$P(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Proof by Induction:

- Base case: $k = 1$: $P(1) = 1 = 1 \times 2/2$.
- Inductive hypothesis: assume $P(k) = k(k+1)/2$.
- Inductive step: Assuming $P(k) = k(k+1)/2$, prove that $P(k+1) = (k+1)(k+2)/2$

$$P(k+1) = \sum_{i=1}^{k+1} i = \sum_{i=1}^{k} i + (k+1)$$

Sum of first n natural numbers

$$P(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Proof by Induction:

- Base case: $k = 1$: $P(1) = 1 = 1 \times 2/2$.
- Inductive hypothesis: assume $P(k) = k(k+1)/2$.
- Inductive step: Assuming $P(k) = k(k+1)/2$, prove that $P(k+1) = (k+1)(k+2)/2$

$$P(k+1) = \sum_{i=1}^{k+1} i = \sum_{i=1}^{k} i + (k+1) = \frac{k(k+1)}{2} + (k+1)$$

Sum of first n natural numbers

$$P(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Proof by Induction:

- Base case: $k = 1$: $P(1) = 1 = 1 \times 2/2$.
- Inductive hypothesis: assume $P(k) = k(k+1)/2$.
- Inductive step: Assuming $P(k) = k(k+1)/2$, prove that $P(k+1) = (k+1)(k+2)/2$

$$\begin{aligned} P(k+1) &= \sum_{i=1}^{k+1} i = \sum_{i=1}^{k} i + (k+1) = \frac{k(k+1)}{2} + (k+1) \\ &= (k+1)(\frac{k}{2} + 2) = \frac{(k+1)(k+2)}{2}. \end{aligned}$$

Recurrence Relation

Given

$$P(n) = \begin{cases} P(\lfloor \frac{n}{2} \rfloor) + 1 & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

prove that

$$P(n) \leq$$

Recurrence Relation

Given

$$P(n) = \begin{cases} P(\lfloor \frac{n}{2} \rfloor) + 1 & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

prove that

$$P(n) \leq 1 + \log_2 n.$$

Recurrence Relation

Given

$$P(n) = \begin{cases} P(\lfloor \frac{n}{2} \rfloor) + 1 & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

prove that

$$P(n) \leq 1 + \log_2 n.$$

- Basis: $k = 1$: $P(1) = 1 \leq 1 + \log_2 1$.

Recurrence Relation

Given

$$P(n) = \begin{cases} P(\lfloor \frac{n}{2} \rfloor) + 1 & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

prove that

$$P(n) \leq 1 + \log_2 n.$$

- Basis: $k = 1$: $P(1) = 1 \leq 1 + \log_2 1$.
- Inductive hypothesis: Assume $P(k) \leq 1 + \log_2 k$. Prove $P(k+1) \leq 1 + \log_2(k+1)$.

Recurrence Relation

Given

$$P(n) = \begin{cases} P(\lfloor \frac{n}{2} \rfloor) + 1 & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

prove that

$$P(n) \leq 1 + \log_2 n.$$

- Basis: $k = 1$: $P(1) = 1 \leq 1 + \log_2 1$.
- Inductive hypothesis: Assume $P(k) \leq 1 + \log_2 k$. Prove $P(k+1) \leq 1 + \log_2(k+1)$.
- Inductive step: $P(k+1) =$

Recurrence Relation

Given

$$P(n) = \begin{cases} P(\lfloor \frac{n}{2} \rfloor) + 1 & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

prove that

$$P(n) \leq 1 + \log_2 n.$$

- Basis: $k = 1$: $P(1) = 1 \leq 1 + \log_2 1$.
- Inductive hypothesis: Assume $P(k) \leq 1 + \log_2 k$. Prove $P(k+1) \leq 1 + \log_2(k+1)$.
- Inductive step: $P(k+1) = P(\lfloor \frac{k+1}{2} \rfloor) + 1$.

Recurrence Relation

Given

$$P(n) = \begin{cases} P(\lfloor \frac{n}{2} \rfloor) + 1 & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

prove that

$$P(n) \leq 1 + \log_2 n.$$

- Basis: $k = 1$: $P(1) = 1 \leq 1 + \log_2 1$.
- Inductive hypothesis: Assume $P(k) \leq 1 + \log_2 k$. Prove $P(k+1) \leq 1 + \log_2(k+1)$.
- Inductive step: $P(k+1) = P(\lfloor \frac{k+1}{2} \rfloor) + 1$.
- We are stuck since inductive hypothesis does not say anything about $P(\lfloor \frac{k+1}{2} \rfloor)$.

Strong Induction

- Use strong induction: In the inductive hypothesis, assume that $P(i)$ is true for all $i \leq k$.

$$P(k+1) = P(\lfloor \frac{k+1}{2} \rfloor) + 1$$

Strong Induction

- Use strong induction: In the inductive hypothesis, assume that $P(i)$ is true for all $i \leq k$.

$$
\begin{aligned}
P(k+1) &= P(\lfloor \frac{k+1}{2} \rfloor) + 1 \\
&\leq 1 + \log_2(\lfloor \frac{k+1}{2} \rfloor) + 1 \\
&\leq 1 + \log_2(k+1) - 1 + 1 = 1 + \log_2(k+1)
\end{aligned}
$$

Efficiency

- Measure resource requirements: how does the amount of time and space 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?

Efficiency

- Measure resource requirements: how does the amount of time and space 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/or use low amounts of space.

Worst-case Running Time

- We will measure worst-case running time of an algorithm.
- 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.
- 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.
- 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.
- 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, except for specific algorithms.
- Assume all elementary operations take unit time: assignment, arithmetic on a fixed-size number, comparisons, array lookup, following a pointer, etc.

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 c.

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 c.
- 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 c.
- 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 *tractable* if it has a polynomial running time.

Comparing Functions

- Assume all functions take only positive values.
- Different algorithms for the same problem may have different (worst-case) running times.
- Example of sorting:

Comparing Functions

- Assume all functions take only positive values.
- Different algorithms for the same problem may have different (worst-case) running times.
- Example of sorting: bubble sort, insertion sort, quick sort, merge sort, etc.

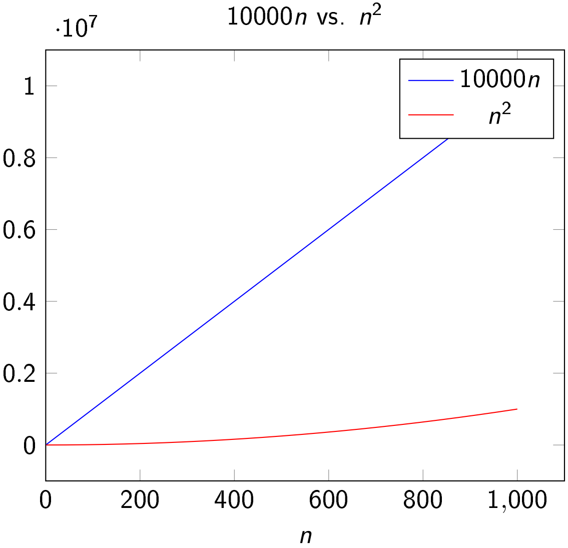
Comparing Functions

- Assume all functions take only positive values.
- Different algorithms for the same problem may have different (worst-case) running times.
- Example of sorting: bubble sort, insertion sort, quick sort, merge sort, etc.
- Bubble sort and insertion sort take roughly n^2 comparisons while quick sort and merge sort take roughly $n \log_2 n$ comparisons.
  - "Roughly" hides potentially large constants, e.g., running time of merge sort may in reality be $100n \log_2 n$.

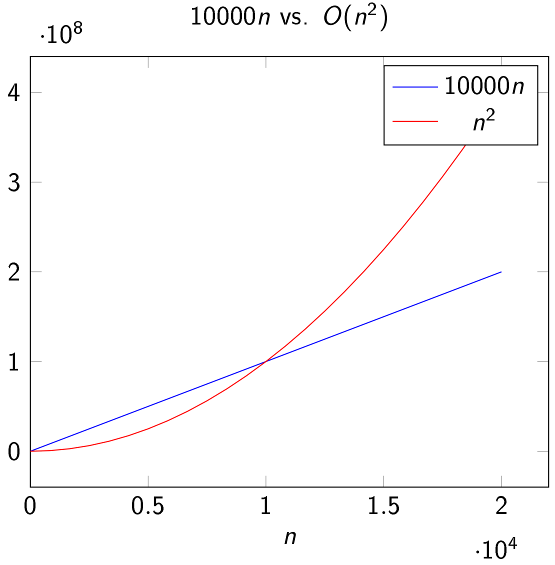
Comparing Functions

- Assume all functions take only positive values.
- Different algorithms for the same problem may have different (worst-case) running times.
- Example of sorting: bubble sort, insertion sort, quick sort, merge sort, etc.
- Bubble sort and insertion sort take roughly n^2 comparisons while quick sort and merge sort take roughly $n \log_2 n$ comparisons.
  - "Roughly" hides potentially large constants, e.g., running time of merge sort may in reality be $100n \log_2 n$.
- How can make statements such as the following?
  - "$100n \log_2 n \leq n^2$"
  - "$10000n \leq n^2$"
  - "$5n^2 - 4n \geq 1000n \log n$"

"$10000n \leq n^2$"



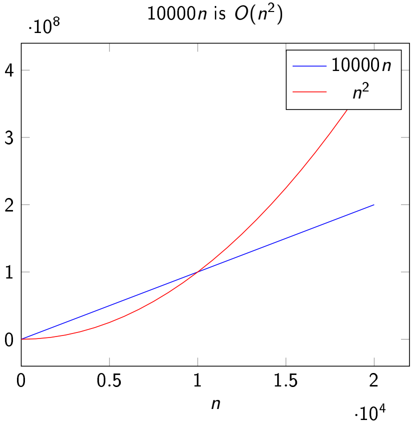
"$10000n \leq n^2$"



Upper Bound

Definition

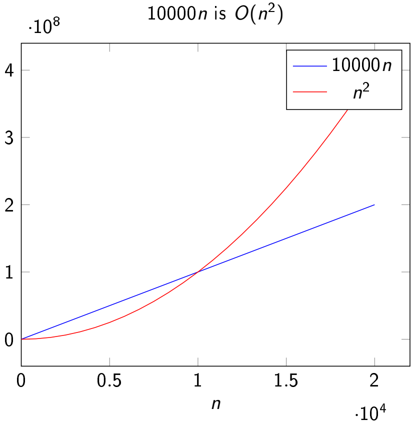
Asymptotic upper bound: A function $f(n)$ is $O(g(n))$ if
for all n , we have $f(n) \leq \ g(n)$.



Upper Bound

Definition

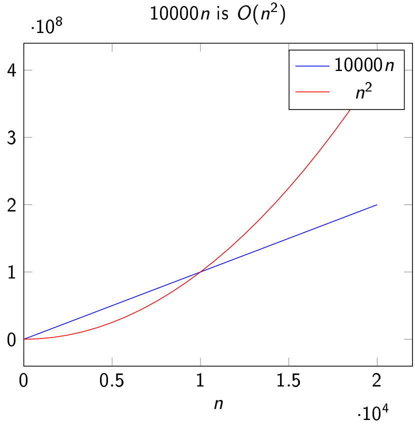
Asymptotic upper bound: A function $f(n)$ is $O(g(n))$ if there exists constant $c > 0$ such that for all n, we have $f(n) \leq cg(n)$.



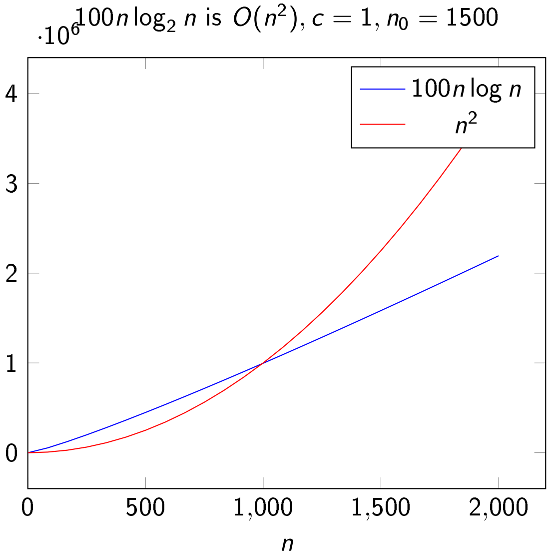
Upper Bound

Definition

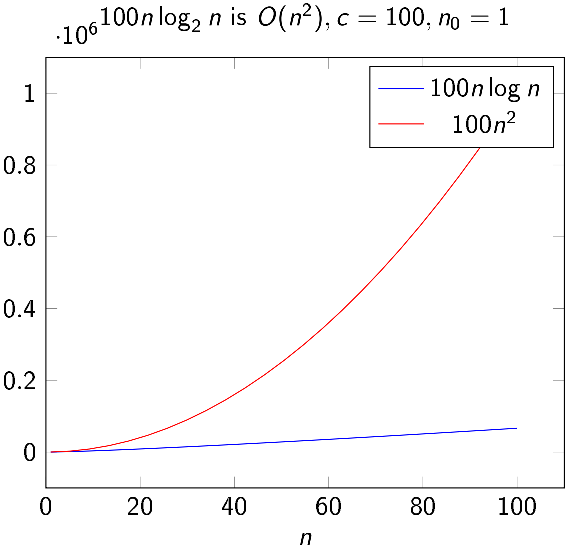
Asymptotic upper bound: A function $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $f(n) \leq cg(n)$.



$100n \log_2 n$ and n^2



$100n \log_2 n$ and n^2



Lower Bound

Definition

Asymptotic lower bound: A function $f(n)$ is $\Omega(g(n))$ if
for all n , we have $f(n) \geq \ g(n)$.

Lower Bound

Definition

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

Lower Bound

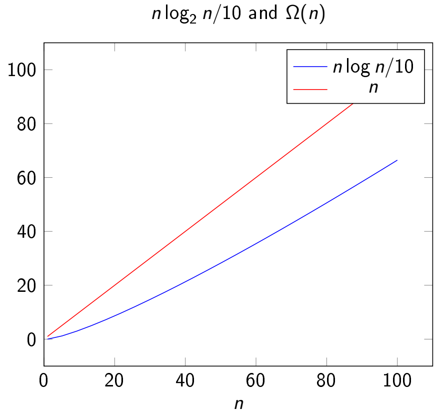
Definition

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 $f(n) \geq cg(n)$.

Lower Bound

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 $f(n) \geq cg(n)$.

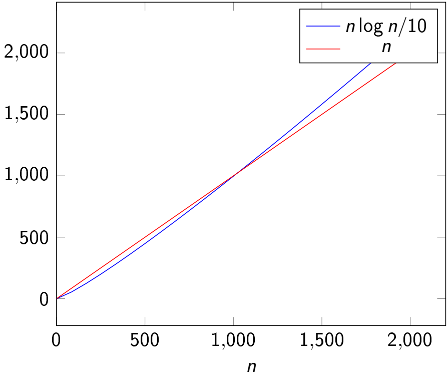


Lower Bound

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 $f(n) \geq cg(n)$.

$n \log_2 n/10$ is $\Omega(n)$, $c = 1$, $n_0 = 1024$



Meaning of “Lower Bound” in Different Contexts

- Functions:

Meaning of "Lower Bound" in Different Contexts

- Functions: n is a lower bound for $n \log n/10$, i.e., $n \log n/10 = \Omega(n)$.

Meaning of "Lower Bound" in Different Contexts

- Functions: n is a lower bound for $n \log n/10$, i.e., $n \log n/10 = \Omega(n)$. This statement is purely about these two mathematical functions without relevance to any algorithm or problem.

Meaning of "Lower Bound" in Different Contexts

- Functions: n is a lower bound for $n \log n/10$, i.e., $n \log n/10 = \Omega(n)$. This statement is purely about these two mathematical functions without relevance to any algorithm or problem.
- Algorithms: The lower bound on the running time of bubble sort is $\Omega(n^2)$.

Meaning of "Lower Bound" in Different Contexts

- Functions: n is a lower bound for $n \log n/10$, i.e., $n \log n/10 = \Omega(n)$. This statement is purely about these two mathematical functions without relevance to any algorithm or problem.
- Algorithms: The lower bound on the running time of bubble sort is $\Omega(n^2)$. There is some input of n numbers that will cause bubble sort to take at least $\Omega(n^2)$ time, e.g.,

Meaning of "Lower Bound" in Different Contexts

- Functions: n is a lower bound for $n \log n/10$, i.e., $n \log n/10 = \Omega(n)$. This statement is purely about these two mathematical functions without relevance to any algorithm or problem.
- Algorithms: The lower bound on the running time of bubble sort is $\Omega(n^2)$. There is some input of n numbers that will cause bubble sort to take at least $\Omega(n^2)$ time, e.g., input the numbers in decreasing order.

Meaning of "Lower Bound" in Different Contexts

- Functions: n is a lower bound for $n \log n/10$, i.e., $n \log n/10 = \Omega(n)$. This statement is purely about these two mathematical functions without relevance to any algorithm or problem.
- Algorithms: The lower bound on the running time of bubble sort is $\Omega(n^2)$. There is some input of n numbers that will cause bubble sort to take at least $\Omega(n^2)$ time, e.g., input the numbers in decreasing order.
- Problems: The problem of sorting n numbers has a lower bound of $\Omega(n \log n)$.

Meaning of "Lower Bound" in Different Contexts

- Functions: n is a lower bound for $n \log n/10$, i.e., $n \log n/10 = \Omega(n)$. This statement is purely about these two mathematical functions without relevance to any algorithm or problem.
- Algorithms: The lower bound on the running time of bubble sort is $\Omega(n^2)$. There is some input of n numbers that will cause bubble sort to take at least $\Omega(n^2)$ time, e.g., input the numbers in decreasing order.
- Problems: The problem of sorting n numbers has a lower bound of $\Omega(n \log n)$. For *any* comparison-based sorting algorithm, there is at least one input for which that algorithm will take $\Omega(n \log n)$ steps.

Tight Bound

Definition

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

Tight Bound

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 all these definitions, c and n_0 are constants independent of n.
- Abuse of notation: say $g(n) = O(f(n))$, $g(n) = \Omega(f(n))$, $g(n) = \Theta(f(n))$.

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 $f + g = \Theta(g)$.

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

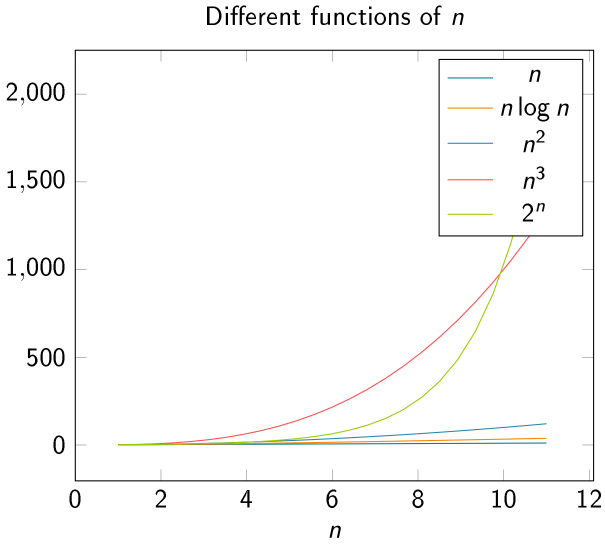
- $f(n) = pn^2 + qn + r$ is $\theta(n^2)$. Can ignore lower order terms.
- Is $f(n) = pn^2 + qn + r = O(n^3)$?
- $f(n) = \sum_{0 \leq i \leq d} a_i n^i = O(n^d)$, if $d > 0$ is an integer constant and $a_d > 0$.
  - $O(n^d)$ is the definition of *polynomial time*.
- Is an algorithm with running time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

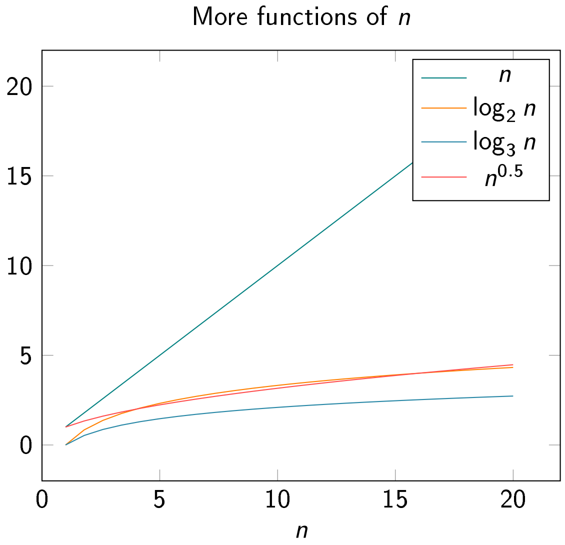
Examples

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

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 constant $x > 0$, $\log n = O(n^x)$.
- For every constant $r > 1$ and every constant $d > 0$, $n^d = O(r^n)$.


Different functions of n
2,000
1,500
1,000
500
0
0
2
4
6
8
10
12
n
n
n log n
n2
n3
2n


More functions of n
n
log2 n
log3 n
n0.5
0
5
10
15
20
0
5
10
15
20
n

Linear Time

- Running time is at most a constant factor times the size of the input.

Linear Time

- Running time is at most a constant factor times the size of the input.
- Finding the minimum, merging two sorted lists.

Linear Time

- Running time is at most a constant factor times the size of the input.
- Finding the minimum, merging two sorted lists.
- Sub-linear time.

Linear Time

- Running time is at most a constant factor times the size of the input.
- Finding the minimum, merging two sorted lists.
- Sub-linear time. Binary search in a sorted array of n numbers takes $O(\log n)$ time.

$O(n \log n)$ Time

- Any algorithm where the costliest step is sorting.

Quadratic Time

- Enumerate all pairs of elements.

Quadratic Time

- Enumerate all pairs of elements.
- Given a set of n points in the plane, find the pair that are the closest.

Quadratic Time

- Enumerate all pairs of elements.
- Given a set of n points in the plane, find the pair that are the closest. Surprising fact: will solve this problem in $O(n \log n)$ time later in the semester.

$O(n^k)$ Time

- Does a graph have an independent set of size k, where k is a constant, i.e. there are k nodes such that no two are joined by an edge?

$O(n^k)$ Time

- Does a graph have an independent set of size k, where k is a constant, i.e. there are k nodes such that no two are joined by an edge?
- Algorithm: For each subset S of k nodes, check if S is an independent set. If the answer is yes, report it.

$O(n^k)$ Time

- Does a graph have an independent set of size k, where k is a constant, i.e. there are k nodes such that no two are joined by an edge?
- Algorithm: For each subset S of k nodes, check if S is an independent set. If the answer is yes, report it.
- Running time is

$O(n^k)$ Time

- Does a graph have an independent set of size k, where k is a constant, i.e. there are k nodes such that no two are joined by an edge?
- Algorithm: For each subset S of k nodes, check if S is an independent set. If the answer is yes, report it.
- Running time is $O(k^2 \binom{n}{k}) = O(n^k)$.

Beyond Polynomial Time

- What is the largest size of an independent set in a graph with n nodes?

Beyond Polynomial Time

- What is the largest size of an independent set in a graph with n nodes?
- Algorithm: For each $1 \leq i \leq n$, check if the graph has an independent size of size i. Output largest independent set found.

Beyond Polynomial Time

- What is the largest size of an independent set in a graph with n nodes?
- Algorithm: For each $1 \leq i \leq n$, check if the graph has an independent size of size i. Output largest independent set found.
- What is the running time?

Beyond Polynomial Time

- What is the largest size of an independent set in a graph with n nodes?
- Algorithm: For each $1 \leq i \leq n$, check if the graph has an independent size of size i. Output largest independent set found.
- What is the running time? $O(n^2 2^n)$.