Introduction to Algorithms

Arijit Bishnu
(arijit@isical.ac.in)

Advanced Computing and Microelectronics Unit
Indian Statistical Institute
Kolkata 700108, India.

Talk at Indian Institute of Technology, Guwahati,
February 15, 2016.

Organization

Outline

O, Upper bound

Any function considered here is $f : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$.

O, Upper bound

Any function considered here is $f : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$.

Big-Oh, O

$T(n)$ is order $f(n)$, $T(n) = O(f(n))$, if for sufficiently large n, the function $T(n)$ is bounded above by a constant multiple of $f(n)$.

O, Upper bound

Any function considered here is $f : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$.

Big-Oh, O

$T(n)$ is order $f(n)$, $T(n) = O(f(n))$, if for sufficiently large n, the function $T(n)$ is bounded above by a constant multiple of $f(n)$.

Formal definition

$T(n) = O(f(n))$, if there exist constants $c > 0$ and $n_0 \geq 0$ so that $\forall n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.
Note that the definition requires a constant c to exist that works for all n; c cannot depend on n.

O, Upper bound

Any function considered here is $f : \mathbb{N} \to \mathbb{R}^+ \cup \{0\}$.

Big-Oh, O

$T(n)$ is order $f(n)$, $T(n) = O(f(n))$, if for sufficiently large n, the function $T(n)$ is bounded above by a constant multiple of $f(n)$.

Formal definition

$T(n) = O(f(n))$, if there exist constants $c > 0$ and $n_0 \geq 0$ so that $\forall n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.

Note that the definition requires a constant c to exist that works for all n; c cannot depend on n.

Exercise

O, Upper bound

Any function considered here is $f : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\}$.

Big-Oh, O

$T(n)$ is order $f(n)$, $T(n) = O(f(n))$, if for sufficiently large n, the function $T(n)$ is bounded above by a constant multiple of $f(n)$.

Formal definition

$T(n) = O(f(n))$, if there exist constants $c > 0$ and $n_0 \geq 0$ so that $\forall n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.
Note that the definition requires a constant c to exist that works for all n; c cannot depend on n.

Exercise

- What is $O(1)$?

O, Upper bound

Any function considered here is $f : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\}$.

Big-Oh, O

$T(n)$ is order $f(n)$, $T(n) = O(f(n))$, if for sufficiently large n, the function $T(n)$ is bounded above by a constant multiple of $f(n)$.

Formal definition

$T(n) = O(f(n))$, if there exist constants $c > 0$ and $n_0 \geq 0$ so that $\forall n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.
Note that the definition requires a constant c to exist that works for all n; c cannot depend on n.

Exercise

- What is $O(1)$?
- Is $2^n = O(2^{n-1})$?

O, Upper bound

Any function considered here is $f : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\}$.

Big-Oh, O

$T(n)$ is order $f(n)$, $T(n) = O(f(n))$, if for sufficiently large n, the function $T(n)$ is bounded above by a constant multiple of $f(n)$.

Formal definition

$T(n) = O(f(n))$, if there exist constants $c > 0$ and $n_0 \geq 0$ so that $\forall n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.
Note that the definition requires a constant c to exist that works for all n; c cannot depend on n.

Exercise

- What is $O(1)$?
- Is $2^n = O(2^{n-1})$?
- What is $1 + r + r^2 + \ldots + r^n$ in terms of $O(r^{??})$?

Some basic idea

Geometric series sum

Some basic idea

Geometric series sum

- What is $a + a \cdot r + a \cdot r^2 + \ldots + a \cdot r^n$ in terms of $O(??)$ when $r < 1$, or $r > 1$?

Some basic idea

Geometric series sum

- What is $a + a \cdot r + a \cdot r^2 + \ldots + a \cdot r^n$ in terms of $O(??)$ when $r < 1$, or $r > 1$?
- If $r = 1$, then the sum is $a(n+1)$; else if $r > 1$, the last term dominates; and if $r < 1$, the first term dominates.

Geometric series sum and throwing away a constant fraction

- An algorithm discards $1/3$ of whatever objects it is processing and recurses on the remaining $2/3$rd and suppose that in each call, the algorithm does not spend more than a constant time for each element. What is the overall time?

Geometric series sum and throwing away a constant fraction

- An algorithm discards $1/3$ of whatever objects it is processing and recurses on the remaining $2/3$rd and suppose that in each call, the algorithm does not spend more than a constant time for each element. What is the overall time?
- $$cn + (2/3)cn + (2/3)^2 cn + \ldots + (2/3)^i cn + \ldots + 1$$
$$= \sum_{i=0}^{\lceil \log_{3/2} n \rceil} (2/3)^i cn \leq \sum_{i=0}^{\infty} (2/3)^i cn = 3cn = \theta(n)$$

Geometric series sum and throwing away a constant fraction

- An algorithm discards $1/3$ of whatever objects it is processing and recurses on the remaining $2/3$rd and suppose that in each call, the algorithm does not spend more than a constant time for each element. What is the overall time?
- $$cn + (2/3)cn + (2/3)^2 cn + \ldots + (2/3)^i cn + \ldots + 1$$
$$= \sum_{i=0}^{\lceil \log_{3/2} n \rceil} (2/3)^i cn \leq \sum_{i=0}^{\infty} (2/3)^i cn = 3cn = \theta(n)$$

The guiding divide and conquer recurrence

The recursion $T(n) = T(c_1 n) + T(c_2 n) + O(n)$ where c_1 and c_2 are constants and $c_1 + c_2 < 1$ solves to $T(n) = O(n)$.

Ω, Lower bound

Omega, Ω

$T(n)$ is omega $g(n)$, $T(n) = \Omega(g(n))$, if for sufficiently large n, the function $T(n)$ is bounded from below by a constant multiple of $g(n)$.

Ω, Lower bound

Omega, Ω

$T(n)$ is omega $g(n)$, $T(n) = \Omega(g(n))$, if for sufficiently large n, the function $T(n)$ is bounded from below by a constant multiple of $g(n)$.

Formal definition

$T(n) = \Omega(g(n))$, if there exist constants $d > 0$ and $n_1 \geq 0$ so that $\forall n \geq n_1$, we have $T(n) \geq d \cdot g(n)$.
Note that the definition requires a constant d to exist that works for all n; d cannot depend on n.

Ω, Lower bound

Omega, Ω

$T(n)$ is omega $g(n)$, $T(n) = \Omega(g(n))$, if for sufficiently large n, the function $T(n)$ is bounded from below by a constant multiple of $g(n)$.

Formal definition

$T(n) = \Omega(g(n))$, if there exist constants $d > 0$ and $n_1 \geq 0$ so that $\forall n \geq n_1$, we have $T(n) \geq d \cdot g(n)$.
Note that the definition requires a constant d to exist that works for all n; d cannot depend on n.

Relation between O and Ω

$f(n) = \Omega(g(n))$ if and only if $g(n) = O(f(n))$.

Exercises on O and Ω

Exercise on O

$$\log(n!) = \sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = O(n \log n)$$

Exercises on O and Ω

Exercise on O

$$\log(n!) = \sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = O(n \log n)$$

Exercise on Ω

$\log(n!) = ?$

Exercises on O and Ω

Exercise on O

$$\log(n!) = \sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = O(n \log n)$$

Exercise on Ω

$\log(n!) = ?$

Exercise on Ω

$$\log(n!) = \sum_{i=1}^{n} \log i \geq \sum_{i=1}^{\lceil \frac{n}{2} \rceil} \log\left(\frac{n}{2}\right) = \left\lceil \frac{n}{2} \right\rceil \log\left(\frac{n}{2}\right) = \Omega(n \log n)$$

Θ, Asymptotically tight bound

Theta, Θ

$T(n)$ is theta $h(n)$, $T(n) = \Theta(h(n))$, if for sufficiently large n, the function $T(n)$ is bounded from both above and below by a constant multiple of $h(n)$.

Θ, Asymptotically tight bound

Theta, Θ

$T(n)$ is theta $h(n)$, $T(n) = \Theta(h(n))$, if for sufficiently large n, the function $T(n)$ is bounded from both above and below by a constant multiple of $h(n)$.

Formal definition

$T(n) = \Theta(h(n))$, if and only if $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$.

Θ, Asymptotically tight bound

Theta, Θ

$T(n)$ is theta $h(n)$, $T(n) = \Theta(h(n))$, if for sufficiently large n, the function $T(n)$ is bounded from both above and below by a constant multiple of $h(n)$.

Formal definition

$T(n) = \Theta(h(n))$, if and only if $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$.

Exercise

Θ, Asymptotically tight bound

Theta, Θ

$T(n)$ is theta $h(n)$, $T(n) = \Theta(h(n))$, if for sufficiently large n, the function $T(n)$ is bounded from both above and below by a constant multiple of $h(n)$.

Formal definition

$T(n) = \Theta(h(n))$, if and only if $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$.

Exercise

- How many constants you need for Θ?

Θ, Asymptotically tight bound

Theta, Θ

$T(n)$ is theta $h(n)$, $T(n) = \Theta(h(n))$, if for sufficiently large n, the function $T(n)$ is bounded from both above and below by a constant multiple of $h(n)$.

Formal definition

$T(n) = \Theta(h(n))$, if and only if $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$.

Exercise

- How many constants you need for Θ?
- Any constant function is $O(1)$, $\Omega(1)$ and $\Theta(1)$.

Properties of asymptotic growth rates

Transitivity

Properties of asymptotic growth rates

Transitivity

- If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

Properties of asymptotic growth rates

Transitivity

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

Properties of asymptotic growth rates

Transitivity

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

Properties of asymptotic growth rates

Transitivity

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

Exercise

Is $2^n = O(1)$?

Properties of asymptotic growth rates

Transitivity

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

Exercise

Is $2^n = O(1)$?

Sums of functions

Let k be a fixed constant, and let $f_1, f_2, \ldots, f_k$ and h be functions such that $f_i = O(h)$, $\forall i$. Then $f_1 + f_2 + \cdots + f_k = O(h)$.

o, Asymptotically tight bound

Small-Oh, o

$T(n) = o(f(n))$, if for sufficiently large n, the function $T(n)$ becomes insignificant relative to $f(n)$.

o, Asymptotically tight bound

Small-Oh, o

$T(n) = o(f(n))$, if for sufficiently large n, the function $T(n)$ becomes insignificant relative to $f(n)$.

Formal definition

$T(n) = O(f(n))$, if for every constant $c > 0$, there exists a $n_0 \geq 0$ so that $\forall n \geq n_0$, we have $T(n) < c \cdot f(n)$.
Note that the definition is for all constants $c > 0$.

o, Asymptotically tight bound

Small-Oh, o

$T(n) = o(f(n))$, if for sufficiently large n, the function $T(n)$ becomes insignificant relative to $f(n)$.

Formal definition

$T(n) = O(f(n))$, if for every constant $c > 0$, there exists a $n_0 \geq 0$ so that $\forall n \geq n_0$, we have $T(n) < c \cdot f(n)$.
Note that the definition is for all constants $c > 0$.

Exercise

Suggest a function that is $o(1)$.

Outline

Binary search

The guiding recurrence

Binary search

The guiding recurrence

- Probe at the middle and based on the comparison, recurse on one half. The guiding recurrence is

Binary search

The guiding recurrence

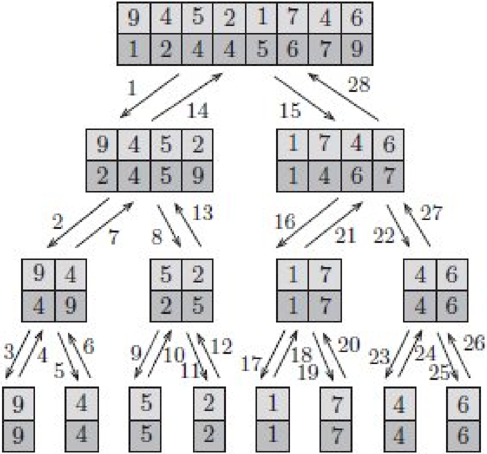
- Probe at the middle and based on the comparison, recurse on one half. The guiding recurrence is
- $T(n) \leq \begin{cases} 1 & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + 1 & \text{if } n \geq 2 \end{cases}.$

Binary search

The guiding recurrence

- Probe at the middle and based on the comparison, recurse on one half. The guiding recurrence is
- $T(n) \leq \begin{cases} 1 & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + 1 & \text{if } n \geq 2 \end{cases}$.
- This recurrence solves to ?

Merge Sort


9 4 5 2 1 7 4 6
1 2 4 4 5 6 7 9
9 4 5 2
2 4 5 9
1 7 4 6
1 4 6 7
9 4
4 9
5 2
2 5
1 7
1 7
4 6
4 6
9
9
4
4
5
5
2
2
1
1
7
7
4
4
6
6
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28

Mergesort

What is the guiding recurrence?

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n - 1 & \text{if } n \geq 2 \end{cases}.$$

Integer multiplication

- Let x and y be two n-bit strings. We want to find $x \times y = xy$.

Integer multiplication

- Let x and y be two n-bit strings. We want to find $x \times y = xy$.
- Let $x = x_L \circ x_R$ and $y = y_L \circ y_R$, where x_L, x_R, y_L, y_R be $n/2$-bit strings.

Integer multiplication

- Let x and y be two n-bit strings. We want to find $x \times y = xy$.
- Let $x = x_L \circ x_R$ and $y = y_L \circ y_R$, where x_L, x_R, y_L, y_R be $n/2$-bit strings.
- $xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R)$
  $= 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$
  (Significant operations are 4 $n/2$-bit multiplications and $O(n)$ left shifts and additions; recurrence is $T(n) \leq 4T(n/2) + O(n)$.)

Integer multiplication

- Let x and y be two n-bit strings. We want to find $x \times y = xy$.
- Let $x = x_L \circ x_R$ and $y = y_L \circ y_R$, where x_L, x_R, y_L, y_R be $n/2$-bit strings.
- $xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R)$
  $= 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$
  (Significant operations are 4 $n/2$-bit multiplications and $O(n)$ left shifts and additions; recurrence is $T(n) \leq 4T(n/2) + O(n)$.)
- $= 2^n x_L y_L + 2^{n/2}\{(x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R\} + x_R y_R$
  (Significant operations are 3 $n/2$-bit multiplications and $O(n)$ left shifts and additions; recurrence is $T(n) \leq 3T(n/2) + O(n)$.)

A few recursions

A divide and conquer recursion

The recursion $T(n) = 2T(n/2) + O(n)$ solves to $T(n) = O(n \log n)$.

A few recursions

A divide and conquer recursion

The recursion $T(n) = 2T(n/2) + O(n)$ solves to $T(n) = O(n \log n)$.

Another divide and conquer recursion

The recursion $T(n) = T(c_1 n) + T(c_2 n) + O(n)$ where c_1 and c_2 are constants and $c_1 + c_2 < 1$ solves to $T(n) = O(n)$.

Solving the previous recurrence $T(n) \leq 3T(n/2) + O(n)$

- Let us look at the recursion tree formed by the pattern of the recursive calls.

Solving the previous recurrence $T(n) \leq 3T(n/2) + O(n)$

- Let us look at the recursion tree formed by the pattern of the recursive calls.
- At each successive level, the size of the subproblems get halved.

Solving the previous recurrence $T(n) \leq 3T(n/2) + O(n)$

- Let us look at the recursion tree formed by the pattern of the recursive calls.
- At each successive level, the size of the subproblems get halved.
- At $(\log_2 n)$-th level, the subproblem size is 1 and the recursion ends. The height of the tree is $\log_2 n$.

Solving the previous recurrence $T(n) \leq 3T(n/2) + O(n)$

- Let us look at the recursion tree formed by the pattern of the recursive calls.
- At each successive level, the size of the subproblems get halved.
- At $(\log_2 n)$-th level, the subproblem size is 1 and the recursion ends. The height of the tree is $\log_2 n$.
- At depth k, there are 3^k subproblems, each of size $n/2^k$.

Solving the previous recurrence $T(n) \leq 3T(n/2) + O(n)$

- Let us look at the recursion tree formed by the pattern of the recursive calls.
- At each successive level, the size of the subproblems get halved.
- At $(\log_2 n)$-th level, the subproblem size is 1 and the recursion ends. The height of the tree is $\log_2 n$.
- At depth k, there are 3^k subproblems, each of size $n/2^k$.
- Total time spent at depth k is $3^k O(\frac{n}{2^k}) = (\frac{3}{2})^k O(n)$.

Solving the previous recurrence $T(n) \leq 3T(n/2) + O(n)$

- Let us look at the recursion tree formed by the pattern of the recursive calls.
- At each successive level, the size of the subproblems get halved.
- At $(\log_2 n)$-th level, the subproblem size is 1 and the recursion ends. The height of the tree is $\log_2 n$.
- At depth k, there are 3^k subproblems, each of size $n/2^k$.
- Total time spent at depth k is $3^k O(\frac{n}{2^k}) = (\frac{3}{2})^k O(n)$.
- The above is a geometric series with common ratio greater than 1. So, the last term $(k = \log_2 n)$ should matter. The last term is $O(3^{\log_2 n}) \approx O(n^{\log_2 3}) \approx O(n^{1.59})$.

Recurrence

Master theorem

If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants $a > 0$, $b > 1$, and $d \geq 0$, then

$$T(n) \leq \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}.$$

Recurrence

Master theorem

If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants $a > 0$, $b > 1$, and $d \geq 0$, then

$$T(n) \leq \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}.$$

Proof idea

The k-th level of the tree is made up of a^k subproblems, each of size n/b^k. The total work done is $a^k \cdot O(\frac{n}{b^k})^d = O(n^d) \cdot (\frac{a}{b^d})^k$. The three cases now follow from the idea of the sum of a geometric series.

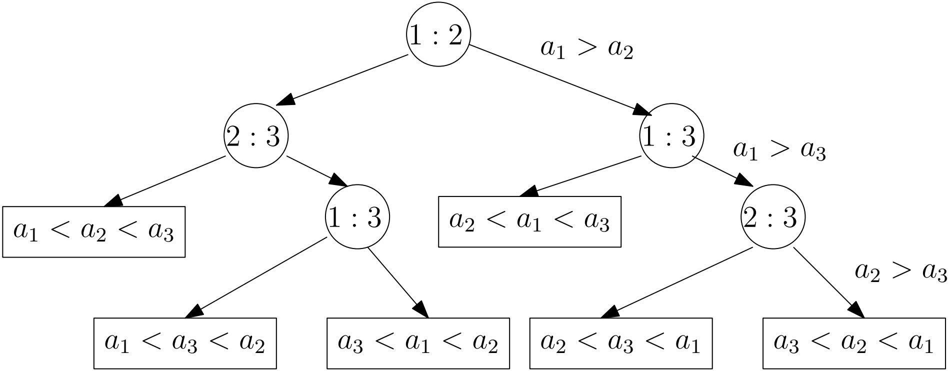
Outline

Lower bound on comparison based sorting

The model of computation is the decision tree model.

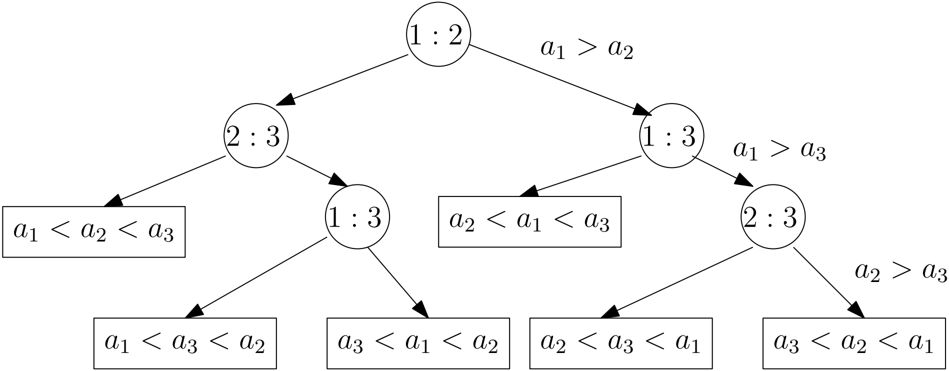
Lower bound on comparison based sorting

The model of computation is the decision tree model.



Lower bound on comparison based sorting

The model of computation is the decision tree model.



Observation

Time complexity in the worst case is the length of a longest path from the root to a leaf, which is the height of the decision tree.

Lower bound on worst case of comparison based sorting

Lower bound on worst case of comparison based sorting is $\Omega(n \log n)$

- Let ℓ be the number of leaves in T and let h be its height.

Lower bound on worst case of comparison based sorting

Lower bound on worst case of comparison based sorting is $\Omega(n \log n)$

- Let ℓ be the number of leaves in T and let h be its height.
- The number of vertices at level h, which are leaves, is at most 2^h.

Lower bound on worst case of comparison based sorting

Lower bound on worst case of comparison based sorting is $\Omega(n \log n)$

- Let ℓ be the number of leaves in T and let h be its height.
- The number of vertices at level h, which are leaves, is at most 2^h.
- Since, $\ell \geq n!$, $n! \leq \ell \leq 2^h$. Therefore, $h \geq \log n!$.

Lower bound on worst case of comparison based sorting

Lower bound on worst case of comparison based sorting is $\Omega(n \log n)$

- Let ℓ be the number of leaves in T and let h be its height.
- The number of vertices at level h, which are leaves, is at most 2^h.
- Since, $\ell \geq n!$, $n! \leq \ell \leq 2^h$. Therefore, $h \geq \log n!$.
- $h \geq \log n! = \sum_{i=1}^{n} \log i = \Omega(n \log n)$.

Lower bound on worst case of comparison based sorting

Lower bound on worst case of comparison based sorting is $\Omega(n \log n)$

- Let ℓ be the number of leaves in T and let h be its height.
- The number of vertices at level h, which are leaves, is at most 2^h.
- Since, $\ell \geq n!$, $n! \leq \ell \leq 2^h$. Therefore, $h \geq \log n!$.
- $h \geq \log n! = \sum_{i=1}^{n} \log i = \Omega(n \log n)$.

Lower bound on average case of comparison based sorting

Lower bound on average case of comparison based sorting is also $\Omega(n \log n)$

Quicksort and heapsort

Quicksort

The worst case time complexity of quicksort is $O(n^2)$.

Heapsort

The worst case time complexity of heapsort is $O(n \log n)$.

Average case analysis of quicksort

- We will assume that the input elements are distinct and w.l.o.g. we also assume that the numbers to be sorted are $\mathcal{A} = \{1, 2, \ldots, n\}$.

Average case analysis of quicksort

- We will assume that the input elements are distinct and w.l.o.g. we also assume that the numbers to be sorted are $\mathcal{A} = \{1, 2, \ldots, n\}$.
- All permutations are equally likely. This ensures that any number in $\mathcal{A}$ is equally likely to be the pivot.

Average case analysis of quicksort

- We will assume that the input elements are distinct and w.l.o.g. we also assume that the numbers to be sorted are $\mathcal{A} = \{1, 2, \ldots, n\}$.
- All permutations are equally likely. This ensures that any number in $\mathcal{A}$ is equally likely to be the pivot.
- Let $T(n)$ denote the number of comparisons done by the algorithm on an average on $\mathcal{A}$.

Average case analysis of quicksort

- We will assume that the input elements are distinct and w.l.o.g. we also assume that the numbers to be sorted are $\mathcal{A} = \{1, 2, \ldots, n\}$.
- All permutations are equally likely. This ensures that any number in $\mathcal{A}$ is equally likely to be the pivot.
- Let $T(n)$ denote the number of comparisons done by the algorithm on an average on $\mathcal{A}$.
- We average over all possible inputs and the expression is

Average case analysis of quicksort

$$T(n) = (n-1) + \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i))$$

Since, $\sum_{i=1}^{n} T(n-i) = T(n-1) + T(n-2) + \cdots + T(0) = \sum_{i=1}^{n} T(i-1)$

We have $T(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n} T(i-1)$.

This recurrence solves to $O(n \log n)$

Average case time complexity of quicksort

Average case time complexity of quicksort is $O(n \log n)$.

Linear time sort(!)

What happened to the lower bound of $\Omega(n \log n)$?

Counting sort

Linear time sort(!)

What happened to the lower bound of $\Omega(n \log n)$?

Counting sort

- Let $\mathcal{L} = \{a_1, \ldots, a_n\}$ be a list of n numbers where each a_i is an integer in the range 0 to k for some integer k.

Linear time sort(!)

What happened to the lower bound of $\Omega(n \log n)$?

Counting sort

- Let $\mathcal{L} = \{a_1, \ldots, a_n\}$ be a list of n numbers where each a_i is an integer in the range 0 to k for some integer k.
- The sorting can be done in $O(n + k)$ time. When $k = O(n)$, then it is $\Theta(n)$.

Linear time sort(!)

What happened to the lower bound of $\Omega(n \log n)$?

Counting sort

- Let $\mathcal{L} = \{a_1, \ldots, a_n\}$ be a list of n numbers where each a_i is an integer in the range 0 to k for some integer k.
- The sorting can be done in $O(n+k)$ time. When $k = O(n)$, then it is $\Theta(n)$.
- Counting sort is a stable sort.

Counting sort

- Input: $A = \{2, 5, 3, 0, 2, 3, 0, 3\}$

Counting sort

- Input: $A = \{2, 5, 3, 0, 2, 3, 0, 3\}$
- Do a hashing type operation on an auxiliary array C of size k to find frequency. $C = \{2, 0, 2, 3, 0, 1\}$

Counting sort

- Input: $A = \{2, 5, 3, 0, 2, 3, 0, 3\}$
- Do a hashing type operation on an auxiliary array C of size k to find frequency. $C = \{2, 0, 2, 3, 0, 1\}$
- Find cumulative frequency of C. $C = \{2, 2, 4, 7, 7, 8\}$

Counting sort

- Input: $A = \{2, 5, 3, 0, 2, 3, 0, 3\}$
- Do a hashing type operation on an auxiliary array C of size k to find frequency. $C = \{2, 0, 2, 3, 0, 1\}$
- Find cumulative frequency of C. $C = \{2, 2, 4, 7, 7, 8\}$
- Start from the end of A; use the number to hash to C; use that number of C to hash to a location of B and write the number you picked up from A. (Indexing is from 0.) Reduce that location of C by 1.

Counting sort

- Input: $A = \{2, 5, 3, 0, 2, 3, 0, 3\}$
- Do a hashing type operation on an auxiliary array C of size k to find frequency. $C = \{2, 0, 2, 3, 0, 1\}$
- Find cumulative frequency of C. $C = \{2, 2, 4, 7, 7, 8\}$
- Start from the end of A; use the number to hash to C; use that number of C to hash to a location of B and write the number you picked up from A. (Indexing is from 0.) Reduce that location of C by 1.
- Working on the example, pick up 3 from A; go to 3rd location of C; there is 7; go to 7th location of B, write 3. B and C looks like $B = \{x, x, x, x, x, x, 3, x\}$ and $C = \{2, 2, 4, 6, 7, 8\}$.

Outline

Lower bounds by reduction

Convex hull problem

Given a set of points $P = \{p_1, \ldots, p_n\}$, compute the smallest convex set that contains P.

Lower bounds by reduction

Convex hull problem

Given a set of points $P = \{p_1, \ldots, p_n\}$, compute the smallest convex set that contains P.

Lower bound for convex hull

Lower bounds by reduction

Convex hull problem

Given a set of points $P = \{p_1, \ldots, p_n\}$, compute the smallest convex set that contains P.

Lower bound for convex hull

- We are to sort the real number $X = \{x_1, \ldots, x_i, \ldots, x_n\}$.

Lower bounds by reduction

Convex hull problem

Given a set of points $P = \{p_1, \ldots, p_n\}$, compute the smallest convex set that contains P.

Lower bound for convex hull

- We are to sort the real number $X = \{x_1, \ldots, x_i, \ldots, x_n\}$.
- With each real number x_i, we associate the point (x_i, x_i^2) in the 2D plane.

Lower bounds by reduction

Convex hull problem

Given a set of points $P = \{p_1, \ldots, p_n\}$, compute the smallest convex set that contains P.

Lower bound for convex hull

- We are to sort the real number $X = \{x_1, \ldots, x_i, \ldots, x_n\}$.
- With each real number x_i, we associate the point (x_i, x_i^2) in the 2D plane.
- Any algorithm for finding convex hull will give the output sorted by their x-coordinates.

Lower bounds by reduction

Convex hull problem

Given a set of points $P = \{p_1, \ldots, p_n\}$, compute the smallest convex set that contains P.

Lower bound for convex hull

- We are to sort the real number $X = \{x_1, \ldots, x_i, \ldots, x_n\}$.
- With each real number x_i, we associate the point (x_i, x_i^2) in the 2D plane.
- Any algorithm for finding convex hull will give the output sorted by their x-coordinates.
- Read the x-coordinates of the points on the convex hull to get the sorted order of X.

Lower bounds by reduction

Convex hull problem

Given a set of points $P = \{p_1, \ldots, p_n\}$, compute the smallest convex set that contains P.

Lower bound for convex hull

- We are to sort the real number $X = \{x_1, \ldots, x_i, \ldots, x_n\}$.
- With each real number x_i, we associate the point (x_i, x_i^2) in the 2D plane.
- Any algorithm for finding convex hull will give the output sorted by their x-coordinates.
- Read the x-coordinates of the points on the convex hull to get the sorted order of X.
- Thus, convex hull has a lower bound of $\Omega(n \log n)$.

Outline

Selection

Median finding

We can find the median or any k-th largest element of a set of elements in $\Theta(n)$ time.

Deterministic selection

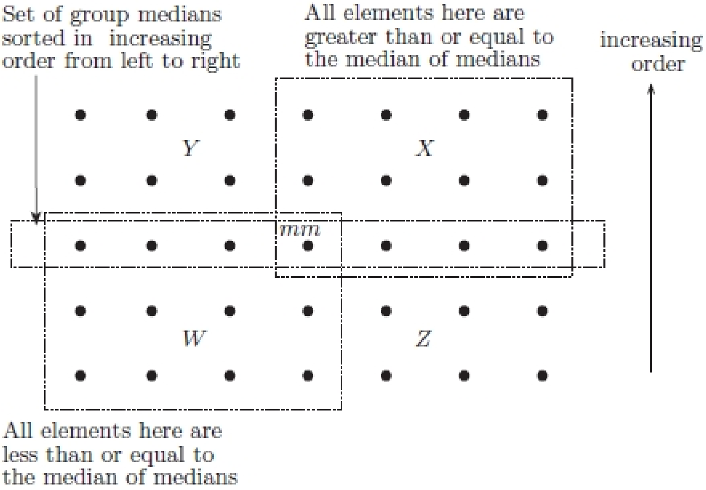


Figure: The schematic for selection algorithm.

A digression into Probability

Expectation

Random variable

A function defined on a sample space is called a random variable. Given a random variable X, $Pr[X = j]$ means X's probability of taking the value j.

Expectation – "the average value"

The expectation of a random variable X is defined as: $E[X] = \sum_{j=0}^{\infty} j \cdot Pr[X = j]$

Waiting for the first success

- Let p be the probability of success and $1 - p$ be the probability of failure of a random experiment.

Waiting for the first success

- Let p be the probability of success and $1 - p$ be the probability of failure of a random experiment.
- If we continue the random experiment till we get success, what is the expected number of experiments we need to perform?

Waiting for the first success

- Let p be the probability of success and $1 - p$ be the probability of failure of a random experiment.
- If we continue the random experiment till we get success, what is the expected number of experiments we need to perform?
- Let X: random variable that equals the number of experiments performed.

Waiting for the first success

- Let p be the probability of success and $1 - p$ be the probability of failure of a random experiment.
- If we continue the random experiment till we get success, what is the expected number of experiments we need to perform?
- Let X: random variable that equals the number of experiments performed.
- For the process to perform exactly j experiments, the first $j - 1$ experiments should be failures and the j-th one should be a success. So, we have $Pr[X = j] = (1 - p)^{(j-1)} \cdot p$.

Waiting for the first success

- Let p be the probability of success and $1 - p$ be the probability of failure of a random experiment.
- If we continue the random experiment till we get success, what is the expected number of experiments we need to perform?
- Let X: random variable that equals the number of experiments performed.
- For the process to perform exactly j experiments, the first $j - 1$ experiments should be failures and the j-th one should be a success. So, we have $Pr[X = j] = (1 - p)^{(j-1)} \cdot p$.
- So, the expectation of X, $E[X] = \sum_{j=0}^{\infty} j \cdot Pr[X = j] = \frac{1}{p}$.

Randomized Quick Sort

RandQSORT(A, p, q)

1: If $p \geq q$, then EXIT.

2: While no central splitter has been found, execute the following steps:

   2.1: Choose uniformly at random a number $r \in \{p, p+1, \ldots, q\}$.

   2.2: Compute s = number of elements in A that are less than $A[r]$, and
   t = number of elements in A that are greater than $A[r]$.

   2.3: If $s \geq \frac{q-p}{4}$ and $t \geq \frac{q-p}{4}$, then $A[r]$ is a central splitter.

3: Position $A[r]$ in $A[s+1]$, put the members in A that are smaller than the central splitter in $A[p \ldots s]$ and the members in A that are larger than the central splitter in $A[s+2 \ldots q]$.

4: RandQSORT(A, p, s);

5: RandQSORT(A, $s+2$, q).

Analysis of RandQSORT

Fact: One execution of Step 2 needs $O(q - p)$ time.

Question: How many times Step 2 is executed for finding a central splitter ?

Result:

The probability that the randomly chosen element is a central splitter is $\frac{1}{2}$.

Recall "Waiting for success"

If p be the probability of success of a random experiment, and we continue the random experiment till we get success, the expected number of experiments we need to perform is $\frac{1}{p}$.

Implication in Our Case

- The expected number of times Step 2 needs to be repeated to get a central splitter (success) is 2 as the corresponding success probability is $\frac{1}{2}$.
- Thus, the expected time complexity of Step 2 is $O(n)$

Analysis of RandQSORT

Time Complexity

- The expected running time for the algorithm on a set A, excluding the time spent on recursive calls, is $O(|A|)$.

Analysis of RandQSORT

Time Complexity

- The expected running time for the algorithm on a set A, excluding the time spent on recursive calls, is $O(|A|)$.
- Worst case size of each partition in j-th level of recursion is $n \cdot (\frac{3}{4})^j$, So, the expected time spent excluding recursive calls is $O(n \cdot (\frac{3}{4})^j)$ for each partition.

Analysis of RandQSORT

Time Complexity

- The expected running time for the algorithm on a set A, excluding the time spent on recursive calls, is $O(|A|)$.
- Worst case size of each partition in j-th level of recursion is $n \cdot (\frac{3}{4})^j$, So, the expected time spent excluding recursive calls is $O(n \cdot (\frac{3}{4})^j)$ for each partition.
- The number of partitions of size $n \cdot (\frac{3}{4})^j$ is $O((\frac{4}{3})^j)$.

Analysis of RandQSORT

Time Complexity

- The expected running time for the algorithm on a set A, excluding the time spent on recursive calls, is $O(|A|)$.
- Worst case size of each partition in j-th level of recursion is $n \cdot (\frac{3}{4})^j$, So, the expected time spent excluding recursive calls is $O(n \cdot (\frac{3}{4})^j)$ for each partition.
- The number of partitions of size $n \cdot (\frac{3}{4})^j$ is $O((\frac{4}{3})^j)$.
- By linearity of expectations, the expected time for all partitions of size $n \cdot (\frac{3}{4})^j$ is $O(n)$.

Analysis of RandQSORT

Time Complexity

- The expected running time for the algorithm on a set A, excluding the time spent on recursive calls, is $O(|A|)$.
- Worst case size of each partition in j-th level of recursion is $n \cdot (\frac{3}{4})^j$, So, the expected time spent excluding recursive calls is $O(n \cdot (\frac{3}{4})^j)$ for each partition.
- The number of partitions of size $n \cdot (\frac{3}{4})^j$ is $O((\frac{4}{3})^j)$.
- By linearity of expectations, the expected time for all partitions of size $n \cdot (\frac{3}{4})^j$ is $O(n)$.
- Number of levels of recursion $= \log_{\frac{4}{3}} n = O(\log n)$.

Analysis of RandQSORT

Time Complexity

- The expected running time for the algorithm on a set A, excluding the time spent on recursive calls, is $O(|A|)$.
- Worst case size of each partition in j-th level of recursion is $n \cdot (\frac{3}{4})^j$, So, the expected time spent excluding recursive calls is $O(n \cdot (\frac{3}{4})^j)$ for each partition.
- The number of partitions of size $n \cdot (\frac{3}{4})^j$ is $O((\frac{4}{3})^j)$.
- By linearity of expectations, the expected time for all partitions of size $n \cdot (\frac{3}{4})^j$ is $O(n)$.
- Number of levels of recursion $= \log_{\frac{4}{3}} n = O(\log n)$.
- Thus, the expected running time is $O(n \log n)$.

Finding the k-th largest

Median Finding

Similar ideas of getting a central splitter and waiting for success bound applies for finding the median in $O(n)$ time.

Outline

Longest common subsequence

Given two strings A and B of lengths n and m respectively over an alphabet set Σ, determine the length of the longest subsequence that is common to both A and B.

Let $A = zxyxyz$ and $B = xyyzx$ and $\Sigma = \{x, y, z\}$. The LCS is $xyyz$ and the length is 4.

Longest common subsequence

Given two strings A and B of lengths n and m respectively over an alphabet set Σ, determine the length of the longest subsequence that is common to both A and B.
Let $A = zxyxyz$ and $B = xyyzx$ and $\Sigma = \{x, y, z\}$. The LCS is $xyyz$ and the length is 4.

- Let $A = a_1 a_2 \cdots a_n$ and $B = b_1 b_2 \cdots b_m$. Let $L[i, j]$ denote the LCS of $a_1 a_2 \cdots a_i$ and $b_1 b_2 \cdots b_j$.

Longest common subsequence

Given two strings A and B of lengths n and m respectively over an alphabet set Σ, determine the length of the longest subsequence that is common to both A and B.
Let $A = zxyxyz$ and $B = xyyzx$ and $\Sigma = \{x, y, z\}$. The LCS is $xyyz$ and the length is 4.

- Let $A = a_1 a_2 \cdots a_n$ and $B = b_1 b_2 \cdots b_m$. Let $L[i, j]$ denote the LCS of $a_1 a_2 \cdots a_i$ and $b_1 b_2 \cdots b_j$.
- If $i = 0$ or $j = 0$, then $L[i, j] = 0$.

Longest common subsequence

Given two strings A and B of lengths n and m respectively over an alphabet set Σ, determine the length of the longest subsequence that is common to both A and B.
Let $A = zxyxyz$ and $B = xyyzx$ and $\Sigma = \{x, y, z\}$. The LCS is $xyyz$ and the length is 4.

- Let $A = a_1 a_2 \cdots a_n$ and $B = b_1 b_2 \cdots b_m$. Let $L[i, j]$ denote the LCS of $a_1 a_2 \cdots a_i$ and $b_1 b_2 \cdots b_j$.
- If $i = 0$ or $j = 0$, then $L[i, j] = 0$.
- If $i, j > 0$ and $a_i = b_j$, then $L[i, j] = L[i - 1, j - 1] + 1$.

Longest common subsequence

Given two strings A and B of lengths n and m respectively over an alphabet set Σ, determine the length of the longest subsequence that is common to both A and B.
Let $A = zxyxyz$ and $B = xyyzx$ and $\Sigma = \{x, y, z\}$. The LCS is $xyyz$ and the length is 4.

- Let $A = a_1a_2 \cdots a_n$ and $B = b_1b_2 \cdots b_m$. Let $L[i, j]$ denote the LCS of $a_1a_2 \cdots a_i$ and $b_1b_2 \cdots b_j$.
- If $i = 0$ or $j = 0$, then $L[i, j] = 0$.
- If $i, j > 0$ and $a_i = b_j$, then $L[i, j] = L[i - 1, j - 1] + 1$.
- If $i, j > 0$ and $a_i \neq b_j$, then $L[i, j] = \max\{L[i, j - 1], L[i - 1, j]\}$.

Longest common subsequence

Given two strings A and B of lengths n and m respectively over an alphabet set Σ, determine the length of the longest subsequence that is common to both A and B.

Let $A = zxyxyz$ and $B = xyyzx$ and $\Sigma = \{x, y, z\}$. The LCS is $xyyz$ and the length is 4.

- Let $A = a_1 a_2 \cdots a_n$ and $B = b_1 b_2 \cdots b_m$. Let $L[i, j]$ denote the LCS of $a_1 a_2 \cdots a_i$ and $b_1 b_2 \cdots b_j$.
- If $i = 0$ or $j = 0$, then $L[i, j] = 0$.
- If $i, j > 0$ and $a_i = b_j$, then $L[i, j] = L[i - 1, j - 1] + 1$.
- If $i, j > 0$ and $a_i \neq b_j$, then $L[i, j] = \max\{L[i, j - 1], L[i - 1, j]\}$.
- The algorithm takes $O(nm)$ time by filling up a table of size nm.