Lecture Note (Part 2)

CSCI 4470/6470 Algorithms, Fall 2023

Liming Cai
Department of Computer Science, UGA

September 25, 2023

Part 2. Elementary algorithms (Chapter2)

Topics to be discussed:

- Divide-and-conquer
- More on time complexity for recursive algorithms
- Quick Sort, order statisics, and randomized versions
- Complexity lower bounds

1. Power of divide-and-conquer

Examples (of recursive algorithms)

1. Power of divide-and-conquer

Examples (of recursive algorithms)

- Fibonacci sequence: returning $(F_1, F_2, \ldots, F_n)$, given n;

1. Power of divide-and-conquer

Examples (of recursive algorithms)

- Fibonacci sequence: returning $(F_1, F_2, \ldots, F_n)$, given n;
- ADD two (long) numbers: $x + y$, given x, y, $|x| = |y| = n$;

1. Power of divide-and-conquer

Examples (of recursive algorithms)

- Fibonacci sequence: returning $(F_1, F_2, \ldots, F_n)$, given n;
- ADD two (long) numbers: $x + y$, given x, y, $|x| = |y| = n$;

  how to add recursively?

1. Power of divide-and-conquer

Examples (of recursive algorithms)

- Fibonacci sequence: returning $(F_1, F_2, \ldots, F_n)$, given n;
- ADD two (long) numbers: $x + y$, given x, y, $|x| = |y| = n$;

  how to add recursively?
- MULTIPLE two numbers: $x \times y$, given x, y, $|x| = |y| = n$;

1. Power of divide-and-conquer

Examples (of recursive algorithms)

- Fibonacci sequence: returning $(F_1, F_2, \ldots, F_n)$, given n;
- ADD two (long) numbers: $x + y$, given x, y, $|x| = |y| = n$;

  how to add recursively?
- MULTIPLE two numbers: $x \times y$, given x, y, $|x| = |y| = n$;

  how to multiple recursively?

1. Power of divide-and-conquer

1. Power of divide-and-conquer

- A not-so-good recursive idea for addition

1. Power of divide-and-conquer

- A not-so-good recursive idea for addition

  Let $n = |x| = |y|$ be the number of bits in x and y.

1. Power of divide-and-conquer

- A not-so-good recursive idea for addition

  Let $n = |x| = |y|$ be the number of bits in x and y.

  $n = 1$, $Add(x, y) = x + y$

1. Power of divide-and-conquer

- A not-so-good recursive idea for addition

  Let $n = |x| = |y|$ be the number of bits in x and y.

  $n = 1,\ Add(x, y) = x + y$

  $n \geq 2$

$$Add(x, y) = \begin{cases} 2 \times Add(\frac{x}{2}, \frac{y}{2}) & x \text{ and } y \text{ are even} \\ 2 \times Add(\lfloor \frac{x}{2} \rfloor, \lfloor \frac{y}{2} \rfloor) + 2 & x \text{ and } y \text{ are odd} \\ 2 \times Add(\lfloor \frac{x}{2} \rfloor, \lfloor \frac{y}{2} \rfloor) + 1 & \text{otherwise} \end{cases}$$

1. Power of divide-and-conquer

1. Power of divide-and-conquer

- time complexity $T(n)$, in terms of bit-wise operations:
  $n = 1$, $T(n) = a$;

1. Power of divide-and-conquer

- time complexity $T(n)$, in terms of bit-wise operations:
  $n = 1$, $T(n) = a$;
  $n \geq 2$

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

1. Power of divide-and-conquer

- time complexity $T(n)$, in terms of bit-wise operations:
  $n = 1$, $T(n) = a$;
  $n \geq 2$

$$T(n) = T(n-1) + bn \quad \longleftarrow \text{ why?}$$

1. Power of divide-and-conquer

- time complexity $T(n)$, in terms of bit-wise operations:
  $n = 1$, $T(n) = a$;
  $n \geq 2$

$$T(n) = T(n-1) + bn \longleftarrow \text{why?}$$

`Exercise`: Prove that $T(n) = O(n^2)$.

1. Power of divide-and-conquer

- time complexity $T(n)$, in terms of bit-wise operations:
  $n = 1$, $T(n) = a$;
  $n \geq 2$

$$T(n) = T(n-1) + bn \longleftarrow \text{why?}$$

`Exercise`: Prove that $T(n) = O(n^2)$.
By the definition of big-O, $T(n) = O(n^2)$ is equivalent to

(1) $T(n) \leq cn^2$ for some $c > 0, k > 0$ when $n \geq k$

1. Power of divide-and-conquer

- time complexity $T(n)$, in terms of bit-wise operations:
  $n = 1$, $T(n) = a$;
  $n \geq 2$

$$T(n) = T(n-1) + bn \quad \longleftarrow \text{why?}$$

`Exercise`: Prove that $T(n) = O(n^2)$.
By the definition of big-O, $T(n) = O(n^2)$ is equivalent to

(1) $T(n) \leq cn^2$ for some $c > 0, k > 0$ when $n \geq k$

So this exercise is to prove (1) using induction.

1. Power of divide-and-conquer

Multiplication of two n-bits numbers:

1. Power of divide-and-conquer

Multiplication of two n-bits numbers:

- multiplication as repeat additions;

1. Power of divide-and-conquer

Multiplication of two n-bits numbers:

- multiplication as repeat additions;
- a different strategy: recursive

$$Mult(x,y) = \begin{cases} 2 \times Mult(x, \frac{y}{2}) & \text{if } y \text{ is even} \\ x + 2 \times Mult(x, \lfloor \frac{y}{2} \rfloor) & \text{if } y \text{ is odd} \end{cases}$$

e.g., $x = 7$, $y = 13$;

1. Power of divide-and-conquer

Multiplication of two n-bits numbers:

- multiplication as repeat additions;
- a different strategy: recursive

$$Mult(x, y) = \begin{cases} 2 \times Mult(x, \frac{y}{2}) & \text{if } y \text{ is even} \\ x + 2 \times Mult(x, \lfloor \frac{y}{2} \rfloor) & \text{if } y \text{ is odd} \end{cases}$$

e.g., $x = 7$, $y = 13$;

base case: ?;

1. Power of divide-and-conquer

Multiplication of two n-bits numbers:

- multiplication as repeat additions;
- a different strategy: recursive

$$Mult(x, y) = \begin{cases} 2 \times Mult(x, \frac{y}{2}) & \text{if } y \text{ is even} \\ x + 2 \times Mult(x, \lfloor \frac{y}{2} \rfloor) & \text{if } y \text{ is odd} \end{cases}$$

  e.g., $x = 7$, $y = 13$;

  base case: ?;

- time complexity of $Mult(x, y)$, assuming $|x| = |y| = n$?

1. Power of divide-and-conquer

Let $T(n)$ be the worst case time complexity for $Mult(x, y)$ when $y = n$.

1. Power of divide-and-conquer

Let $T(n)$ be the worst case time complexity for $Mult(x, y)$ when $y = n$.

Then

$$T(n) = \begin{cases} an & n = 1 \\ T(n-1) + bn & n \geq 2 \end{cases}$$

for constants $a, b > 0$.

1. Power of divide-and-conquer

Let $T(n)$ be the worst case time complexity for $Mult(x, y)$ when $y = n$.

Then

$$T(n) = \begin{cases} an & n = 1 \\ T(n-1) + bn & n \geq 2 \end{cases}$$

for constants $a, b > 0$.

Claim: $T(n) = O(n^2)$. That is, there is $c > 0$, $T(n) \leq cn^2$ for all $n \geq 1$.

in class exercise Prove the claim by induction.

1. Power of divide-and-conquer

Multiplication (revisited)

Input: two n-bits long binary numbers x and y, $|x| = |y| = n$,

Output: $x \times y$.

1. Power of divide-and-conquer

Multiplication (revisited)

Input: two n-bits long binary numbers x and y, $|x| = |y| = n$,
Output: $x \times y$.

Divide-and-conquer:

- split x and y into high and low segments, each with $\frac{n}{2}$ bits;
- $x \times y$ uses 4 $\times$'s on segments plus some additions;

$$\begin{aligned} xy &= (x_h 2^{\frac{n}{2}} + x_l)(y_h 2^{\frac{n}{2}} + y_l) \\ &= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l \end{aligned}$$

1. Power of divide-and-conquer

Multiplication (revisited)

Input: two n-bits long binary numbers x and y, $|x| = |y| = n$,

Output: $x \times y$.

Divide-and-conquer:

- split x and y into high and low segments, each with $\frac{n}{2}$ bits;
- $x \times y$ uses 4 $\times$'s on segments plus some additions;

$$\begin{aligned} xy &= (x_h 2^{\frac{n}{2}} + x_l)(y_h 2^{\frac{n}{2}} + y_l) \\ &= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l \end{aligned}$$

Time complexity

$$T(n) = \begin{cases} a & n = 1 \\ 4T(\frac{n}{2}) + bn & n \geq 2 \end{cases}$$

1. Power of divide-and-conquer

Multiplication (revisited)

Input: two n-bits long binary numbers x and y, $|x| = |y| = n$,
Output: $x \times y$.

Divide-and-conquer:

- split x and y into high and low segments, each with $\frac{n}{2}$ bits;
- $x \times y$ uses 4 $\times$'s on segments plus some additions;

$$\begin{aligned} xy &= (x_h 2^{\frac{n}{2}} + x_l)(y_h 2^{\frac{n}{2}} + y_l) \\ &= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l \end{aligned}$$

Time complexity

$$T(n) = \begin{cases} a & n = 1 \\ 4T(\frac{n}{2}) + bn & n \geq 2 \quad \longleftarrow \text{why?} \end{cases}$$

Prove $T(n) = O(n^2)$ (homework)

1. Power of divide-and-conquer

Multiplication (a better solution)

$$
\begin{aligned}
xy &= (x_h 2^{\frac{n}{2}} + x_l)(y_h 2^{\frac{n}{2}} + y_l) \\
&= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l
\end{aligned}
$$

1. Power of divide-and-conquer

Multiplication (a better solution)

$$
\begin{aligned}
xy &= (x_h 2^{\frac{n}{2}} + x_l)(y_h 2^{\frac{n}{2}} + y_l) \\
&= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l
\end{aligned}
$$

where with 1 '$\times$' operation

1. Power of divide-and-conquer

Multiplication (a better solution)

$$\begin{aligned} xy &= (x_h 2^{\frac{n}{2}} + x_l)(y_h 2^{\frac{n}{2}} + y_l) \\ &= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l \end{aligned}$$

where with 1 '$\times$' operation

$$(x_h y_h + x_l y_h) = (x_h + x_l)(y_h + y_l) - x_h y_h - x_l y_l$$

1. Power of divide-and-conquer

Multiplication (a better solution)

$$
\begin{aligned}
xy &= (x_h 2^{\frac{n}{2}} + x_l)(y_h 2^{\frac{n}{2}} + y_l) \\
&= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l
\end{aligned}
$$

where with 1 '$\times$' operation

$$(x_h y_h + x_l y_h) = (x_h + x_l)(y_h + y_l) - x_h y_h - x_l y_l$$

$T(n) = 3T(n/2) + O(n)$ leading to

$T(n) = O(n^{1.6})$ (proof, homework question)

1. Power of divide-and-conquer

Matrix multiplication

1. Power of divide-and-conquer

Matrix multiplication

1. Power of divide-and-conquer

Matrix multiplication

- for 2×2 matrices: $A_{(2\times2)} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$

$$A_{(2\times2)} \times B_{(2\times2)} = C_{(2\times2)}$$

1. Power of divide-and-conquer

Matrix multiplication

- for 2×2 matrices: $A_{(2\times 2)} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$

$$A_{(2\times 2)} \times B_{(2\times 2)} = C_{(2\times 2)}$$

conventional way: 8 scalar multiplications needed

1. Power of divide-and-conquer

Matrix multiplication

- for 2×2 matrices: $A_{(2\times2)} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$

$$A_{(2\times2)} \times B_{(2\times2)} = C_{(2\times2)}$$

  conventional way: 8 scalar multiplications needed

- for $n \times n$ matrices: $A_{n\times n} \times B_{n\times n} = C_{n\times n}$

1. Power of divide-and-conquer

Matrix multiplication

- for 2×2 matrices: $A_{(2\times2)} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$

$$A_{(2\times2)} \times B_{(2\times2)} = C_{(2\times2)}$$

  conventional way: 8 scalar multiplications needed

- for $n \times n$ matrices: $A_{n\times n} \times B_{n\times n} = C_{n\times n}$

$$A_{(n\times n)} = \begin{bmatrix} X_{(\frac{n}{2}\times\frac{n}{2})} & Y_{(\frac{n}{2}\times\frac{n}{2})} \\ Z_{(\frac{n}{2}\times\frac{n}{2})} & W_{(\frac{n}{2}\times\frac{n}{2})} \end{bmatrix}$$

1. Power of divide-and-conquer

Matrix multiplication

- for 2×2 matrices: $A_{(2\times 2)} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$

$$A_{(2\times 2)} \times B_{(2\times 2)} = C_{(2\times 2)}$$

conventional way: 8 scalar multiplications needed

- for $n \times n$ matrices: $A_{n\times n} \times B_{n\times n} = C_{n\times n}$

$$A_{(n\times n)} = \begin{bmatrix} X_{(\frac{n}{2}\times\frac{n}{2})} & Y_{(\frac{n}{2}\times\frac{n}{2})} \\ Z_{(\frac{n}{2}\times\frac{n}{2})} & W_{(\frac{n}{2}\times\frac{n}{2})} \end{bmatrix}$$

conventional way: 8 dim$_{(\frac{n}{2}\times\frac{n}{2})}$-matrix multiplications needed

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

1. Power of divide-and-conquer

Matrix multiplication

- for 2×2 matrices: $A_{(2\times 2)} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$

$$A_{(2\times 2)} \times B_{(2\times 2)} = C_{(2\times 2)}$$

conventional way: 8 scalar multiplications needed

- for $n \times n$ matrices: $A_{n\times n} \times B_{n\times n} = C_{n\times n}$

$$A_{(n\times n)} = \begin{bmatrix} X_{(\frac{n}{2}\times\frac{n}{2})} & Y_{(\frac{n}{2}\times\frac{n}{2})} \\ Z_{(\frac{n}{2}\times\frac{n}{2})} & W_{(\frac{n}{2}\times\frac{n}{2})} \end{bmatrix}$$

conventional way: 8 dim$_{(\frac{n}{2}\times\frac{n}{2})}$-matrix multiplications needed

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

then $T(n) = O(?)$,

1. Power of divide-and-conquer

Matrix multiplication

- for 2×2 matrices: $A_{(2\times2)} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$

$$A_{(2\times2)} \times B_{(2\times2)} = C_{(2\times2)}$$

conventional way: 8 scalar multiplications needed

- for $n \times n$ matrices: $A_{n\times n} \times B_{n\times n} = C_{n\times n}$

$$A_{(n\times n)} = \begin{bmatrix} X_{(\frac{n}{2}\times\frac{n}{2})} & Y_{(\frac{n}{2}\times\frac{n}{2})} \\ Z_{(\frac{n}{2}\times\frac{n}{2})} & W_{(\frac{n}{2}\times\frac{n}{2})} \end{bmatrix}$$

conventional way: 8 dim$_{(\frac{n}{2}\times\frac{n}{2})}$-matrix multiplications needed

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

then $T(n) = O(?)$, prove by induction

1. Power of divide-and-conquer

Example 2: Matrix multiplication (a better solution)

1. Power of divide-and-conquer

Example 2: Matrix multiplication (a better solution)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} x & y \\ z & w \end{bmatrix} = \begin{bmatrix} ax + bz & ay + bw \\ cx + dz & cy + dw \end{bmatrix}$$

1. Power of divide-and-conquer

Example 2: Matrix multiplication (a better solution)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} x & y \\ z & w \end{bmatrix} = \begin{bmatrix} ax+bz & ay+bw \\ cx+dz & cy+dw \end{bmatrix}$$

more clever algebra, with

$$\begin{cases} s_1 = a(y-w) & s_5 = (a+d)(x+w) \\ s_2 = (a+b)w & s_6 = (b-d)(z+w) \\ s_3 = (c+d)x & s_7 = (a-c)(x+y) \\ s_4 = d(z-x) & \end{cases}$$

1. Power of divide-and-conquer

Example 2: Matrix multiplication (a better solution)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} x & y \\ z & w \end{bmatrix} = \begin{bmatrix} ax + bz & ay + bw \\ cx + dz & cy + dw \end{bmatrix}$$

more clever algebra, with

$$\begin{cases} s_1 = a(y - w) & s_5 = (a + d)(x + w) \\ s_2 = (a + b)w & s_6 = (b - d)(z + w) \\ s_3 = (c + d)x & s_7 = (a - c)(x + y) \\ s_4 = d(z - x) & \end{cases}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} x & y \\ z & w \end{bmatrix} = \begin{bmatrix} s_4 + s_5 + s_6 - s_2 & s_1 + s_2 \\ s_3 + s_4 & s_1 + s_5 - s_3 - s_7 \end{bmatrix}$$

1. Power of divide-and-conquer

Example 2: Matrix multiplication (a better solution)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} x & y \\ z & w \end{bmatrix} = \begin{bmatrix} ax+bz & ay+bw \\ cx+dz & cy+dw \end{bmatrix}$$

more clever algebra, with

$$\begin{cases} s_1 = a(y-w) & s_5 = (a+d)(x+w) \\ s_2 = (a+b)w & s_6 = (b-d)(z+w) \\ s_3 = (c+d)x & s_7 = (a-c)(x+y) \\ s_4 = d(z-x) & \end{cases}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} x & y \\ z & w \end{bmatrix} = \begin{bmatrix} s_4+s_5+s_6-s_2 & s_1+s_2 \\ s_3+s_4 & s_1+s_5-s_3-s_7 \end{bmatrix}$$

7 multiplications and 18 additions/subtractions !

1. Power of divide-and-conquer

Example 2: Matrix multiplication (a better solution)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} x & y \\ z & w \end{bmatrix} = \begin{bmatrix} ax+bz & ay+bw \\ cx+dz & cy+dw \end{bmatrix}$$

more clever algebra, with

$$\begin{cases} s_1 = a(y-w) & s_5 = (a+d)(x+w) \\ s_2 = (a+b)w & s_6 = (b-d)(z+w) \\ s_3 = (c+d)x & s_7 = (a-c)(x+y) \\ s_4 = d(z-x) & \end{cases}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} x & y \\ z & w \end{bmatrix} = \begin{bmatrix} s_4+s_5+s_6-s_2 & s_1+s_2 \\ s_3+s_4 & s_1+s_5-s_3-s_7 \end{bmatrix}$$

7 multiplications and 18 additions/subtractions !

$T(n) = 7T(n/2) + O(n^2)$, leading to $T(n) = O(n^{2.81})$.

1. Power of divide-and-conquer

Example: Merge Sort

Input: a list $L[1..n]$ of n elements;
Output: list $\bar{L}[1..n]$, a permutation of L, such that

$$\forall i,\ 1 \leq i < n,\ \bar{L}[i] \leq \bar{L}[i+1]$$

1. Power of divide-and-conquer

Example: Merge Sort

Input: a list $L[1..n]$ of n elements;
Output: list $\bar{L}[1..n]$, a permutation of L, such that

$$\forall i,\ 1 \leq i < n,\ \bar{L}[i] \leq \bar{L}[i+1]$$

Merge Sort algorithm:

1. Power of divide-and-conquer

Example: Merge Sort

Input: a list $L[1..n]$ of n elements;
Output: list $\bar{L}[1..n]$, a permutation of L, such that

$$\forall i,\ 1 \leq i < n,\ \bar{L}[i] \leq \bar{L}[i+1]$$

Merge Sort algorithm:

- idea: partition L into two sublists of equal sizes;
  sort the two sublists;
  merge the sorted two sublists into one;

1. Power of divide-and-conquer

1. Power of divide-and-conquer

- iterative algorithm (exercise at home)

1. Power of divide-and-conquer

- iterative algorithm (exercise at home)
- recursive algorithm:

recursive case ($l < h$):

$$\mathrm{MergeSort}(L, l, h) =$$
$$\mathrm{MergeTwo}\Big(\mathrm{MergeSort}(L, l, \lfloor \frac{l+h}{2} \rfloor), \mathrm{MergeSort}(L, \lfloor \frac{l+h}{2} \rfloor + 1, h)\Big)$$

1. Power of divide-and-conquer

- iterative algorithm (exercise at home)
- recursive algorithm:

recursive case ($l < h$):

$$\begin{aligned}&\mathrm{MergeSort}(L, l, h) = \\ &\quad \mathrm{MergeTwo}\Big(\mathrm{MergeSort}(L, l, \lfloor\frac{l+h}{2}\rfloor), \mathrm{MergeSort}(L, \lfloor\frac{l+h}{2}\rfloor + 1, h)\Big)\end{aligned}$$

base case?

1. Power of divide-and-conquer

Time complexity for MERGESORT:

1. Power of divide-and-conquer

Time complexity for MergeSort:

- what does n correspond to in the algorithm?

1. Power of divide-and-conquer

Time complexity for MERGESORT:

- what does n correspond to in the algorithm?
- base case $T(?) = ?$

1. Power of divide-and-conquer

Time complexity for MERGESORT:

- what does n correspond to in the algorithm?
- base case $T(?) = ?$
- recursive cases:

$$T(n) = 2T(n/2) + T_{\text{MergeTwo}}(n)$$

1. Power of divide-and-conquer

Time complexity for MERGESORT:

- what does n correspond to in the algorithm?
- base case $T(?) = ?$
- recursive cases:

$$T(n) = 2T(n/2) + T_{\text{MergeTwo}}(n)$$

$T_{\text{MergeTwo}}(n) = O(n)$ or $\leq bn$.

1. Power of divide-and-conquer

Time complexity for MergeSort:

- what does n correspond to in the algorithm?
- base case $T(?) = ?$
- recursive cases:

$$T(n) = 2T(n/2) + T_{\text{MergeTwo}}(n)$$

$T_{\text{MergeTwo}}(n) = O(n)$ or $\leq bn$. So

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

1. Power of divide-and-conquer

Time complexity for MergeSort:

- what does n correspond to in the algorithm?
- base case $T(?) = ?$
- recursive cases:

$$T(n) = 2T(n/2) + T_{\text{MergeTwo}}(n)$$

$T_{\text{MergeTwo}}(n) = O(n)$ or $\leq bn$. So

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

- Prove that $T(n) = O(n \log_2 n)$

1. Power of divide-and-conquer

Time complexity for MergeSort:

- what does n correspond to in the algorithm?
- base case $T(?) = ?$
- recursive cases:

$$T(n) = 2T(n/2) + T_{\text{MergeTwo}}(n)$$

$T_{\text{MergeTwo}}(n) = O(n)$ or $\leq bn$. So

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

- Prove that $T(n) = O(n \log_2 n)$ (using ?)

1. Power of divide-and-conquer

Time complexity for MergeSort:

- what does n correspond to in the algorithm?
- base case $T(?) = ?$
- recursive cases:

$$T(n) = 2T(n/2) + T_{\text{MergeTwo}}(n)$$

$T_{\text{MergeTwo}}(n) = O(n)$ or $\leq bn$. So

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

- Prove that $T(n) = O(n \log_2 n)$ (using ?)

  - using the unfolding method
  - using the recursive tree method
  - using the induction method

1. Power of divide-and-conquer

Prove that Merge Sort time complexity $T(n) = O(n \log_2 n)$

Proof with unfolding

1. Power of divide-and-conquer

Prove that Merge Sort time complexity $T(n) = O(n \log_2 n)$

Proof with unfolding

- by the recursive formula of $T(n)$ derived from Merge Sort algorithm

$$T(n) = \begin{cases} a & n = 1 \\ 2T(n/2) + bn & n \geq 2 \end{cases}$$

1. Power of divide-and-conquer

Prove that Merge Sort time complexity $T(n) = O(n \log_2 n)$

Proof with unfolding

- by the recursive formula of $T(n)$ derived from Merge Sort algorithm

$$T(n) = \begin{cases} a & n = 1 \\ 2T(n/2) + bn & n \geq 2 \end{cases}$$

- unfold for all different values of n:

$$\begin{aligned} T(n) &= 2T(n/2) + bn \\ T(n/2) &= 2T(n/2^2) + bn/2 \\ T(n/2^2) &= 2T(n/2^3) + bn/2^2 \\ &\cdots\cdots \\ T(n/2^k) &= 2T(n/2^{k+1}) + bn/2^k \end{aligned}$$

1. Power of divide-and-conquer

1. Power of divide-and-conquer

- to cancel same terms on left and right sides, $\times$ power of 2 to each equation

$$\begin{aligned}
T(n) &= 2T(n/2) + bn \\
2T(n/2) &= 2\,2T(n/2^2) + 2bn/2 \\
2^2T(n/2^2) &= 2^2\,2T(n/2^3) + 2^2bn/2^2 \\
&\cdots\cdots \\
2^kT(n/2^k) &= 2^k\,2T(n/2^{k+1}) + 2^kbn/2^k \\
+\rule{10cm}{0.4pt} \\
T(n) &= 2^k\,2T(n/2^{k+1}) + bn \times (k+1)
\end{aligned}$$

where $n/2^{k+1} = 1$, so $k+1 = \log_2 n$.

1. Power of divide-and-conquer

- to cancel same terms on left and right sides, $\times$ power of 2 to each equation

$$
\begin{aligned}
T(n) &= 2T(n/2) + bn \\
2T(n/2) &= 2 2T(n/2^2) + 2bn/2 \\
2^2 T(n/2^2) &= 2^2 2T(n/2^3) + 2^2 bn/2^2 \\
&\cdots\cdots \\
2^k T(n/2^k) &= 2^k 2T(n/2^{k+1}) + 2^k bn/2^k \\
+\overline{\qquad\qquad} & \overline{\qquad\qquad\qquad\qquad\qquad\qquad} \\
T(n) &= 2^k 2T(n/2^{k+1}) + bn \times (k+1)
\end{aligned}
$$

where $n/2^{k+1} = 1$, so $k+1 = \log_2 n$.

- concluded:

$$T(n) = nT(1) + bn \log_2 n = an + bn \log_2 n = O(n \log_2 n)$$

1. Power of divide-and-conquer

- to cancel same terms on left and right sides, $\times$ power of 2 to each equation

$$
\begin{aligned}
T(n) &= 2T(n/2) + bn \\
2T(n/2) &= 2\,2T(n/2^2) + 2bn/2 \\
2^2T(n/2^2) &= 2^2\,2T(n/2^3) + 2^2bn/2^2 \\
&\cdots\cdots \\
2^kT(n/2^k) &= 2^k\,2T(n/2^{k+1}) + 2^kbn/2^k \\
+\rule{0pt}{0pt}&\overline{} \\
T(n) &= 2^k\,2T(n/2^{k+1}) + bn \times (k+1)
\end{aligned}
$$

where $n/2^{k+1} = 1$, so $k + 1 = \log_2 n$.

- concluded:

$$T(n) = nT(1) + bn\log_2 n = an + bn\log_2 n = O(n\log_2 n) \; \textit{why?}$$

1. Power of divide-and-conquer

Prove that Merge Sort time complexity $T(n) = O(n \log_2 n)$

Proof with induction

1. Power of divide-and-conquer

Prove that Merge Sort time complexity $T(n) = O(n \log_2 n)$

Proof with induction

- equivalent to proving that $\exists c > 0, n_0 > 0$, such that $T(n) \leq cn \log_2 n$, when $n \geq n_0$.

1. Power of divide-and-conquer

Prove that Merge Sort time complexity $T(n) = O(n \log_2 n)$

Proof with induction

- equivalent to proving that $\exists c > 0, n_0 > 0$, such that $T(n) \le cn \log_2 n$, when $n \ge n_0$.
- base case: $n = 1$, $T(1) \le c \times 1 \times \log_2 1$ – what happened? fix this!

1. Power of divide-and-conquer

Prove that Merge Sort time complexity $T(n) = O(n \log_2 n)$

Proof with induction

- equivalent to proving that $\exists c > 0, n_0 > 0$,
  such that $T(n) \leq cn \log_2 n$, when $n \geq n_0$.
- base case: $n = 1$, $T(1) \leq c \times 1 \times \log_2 1$ – what happened? fix this!
- assumption: when $n = k/2$, $T(k/2) \leq ck/2 \log_2 k/2$;

1. Power of divide-and-conquer

Prove that Merge Sort time complexity $T(n) = O(n \log_2 n)$

Proof with induction

- equivalent to proving that $\exists c > 0, n_0 > 0$,
  such that $T(n) \leq cn \log_2 n$, when $n \geq n_0$.
- base case: $n = 1$, $T(1) \leq c \times 1 \times \log_2 1$ – what happened? fix this!
- assumption: when $n = k/2$, $T(k/2) \leq ck/2 \log_2 k/2$;
- induction: when $n = k$,

$$
\begin{aligned}
T(k) &= 2T(k/2) + bk \quad \text{– by recursive formula} \\
&\leq 2ck/2 \log_2 k/2 + bk \quad \text{– by assumption} \\
&= ck \log_2 k/2 + bk \\
&= ck(\log_2 k - \log_2 2) + bk \\
&= ck \log_2 k - ck + bk \\
&\leq ck \log_2 k \quad \text{– if choose } c > b
\end{aligned}
$$

1. Power of divide-and-conquer

Example: Quick Sort

1. Power of divide-and-conquer

Example: Quick Sort

Idea:

- select a pivot element e from the input list L;
- partition list L into two sublists L_h and L_l such that
  $\forall x \in L_h, x > e$
  $\forall x \in L_l, x \leq e$
- recursively sort the two sublists L_h and L_l separately;

1. Power of divide-and-conquer

Example: Quick Sort

Idea:

- select a pivot element e from the input list L;
- partition list L into two sublists L_h and L_l such that
  $\forall x \in L_h, x > e$
  $\forall x \in L_l, x \leq e$
- recursively sort the two sublists L_h and L_l separately;

```
function quicksort (L, low, high);

if (low < high)
  k = partition (L, low, high);
  quicksort (L, low, k-1);
  quicksort (L, k+1, high);
return L;
```

1. Power of divide-and-conquer

how does `partition` work?

```
function partition(L, p, r);

e = L[r];
i = p-1;
for j = p to r-1
   if L[j] <= e
      i = i + 1;
      exchange (L[i], L[j]);
exchange (L[i+1], L[r]);
return i+1
```

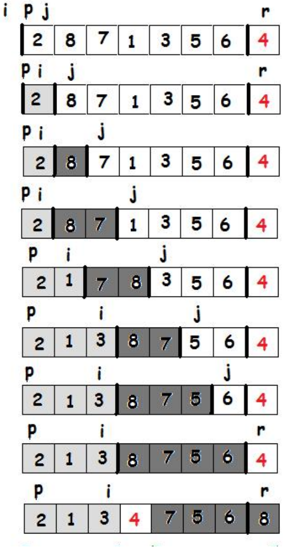
Single pass, dynamically 3 regions:

`L[p..i]`:

`L[i+1..j-1]`:

`L[j..r-1]`:

before pivot is in position $\longrightarrow$



1. Power of divide-and-conquer

Estimation of complexity for `quicksort`;

1. Power of divide-and-conquer

Estimation of complexity for `quicksort`;

- worst case situation $T(n) = T(n-1) + bn$

1. Power of divide-and-conquer

Estimation of complexity for `quicksort`;

- worst case situation $T(n) = T(n-1) + bn$ why?

1. Power of divide-and-conquer

Estimation of complexity for `quicksort`;

- worst case situation $T(n) = T(n-1) + bn$ why?
- ideal case situation $T(n) = 2T(\frac{n}{2}) + bn$

1. Power of divide-and-conquer

Estimation of complexity for `quicksort`;

- worst case situation $T(n) = T(n-1) + bn$ why?
- ideal case situation $T(n) = 2T(\frac{n}{2}) + bn$ why?

1. Power of divide-and-conquer

Estimation of complexity for `quicksort`;

- worst case situation $T(n) = T(n-1) + bn$ why?
- ideal case situation $T(n) = 2T(\frac{n}{2}) + bn$ why?
- not so ideal, but "balanced" cases, for some $\alpha + \beta = 1$,

$$T(n) = T(\alpha n) + T(\beta n) + bn$$

1. Power of divide-and-conquer

Estimation of complexity for `quicksort`;

- worst case situation $T(n) = T(n-1) + bn$ why?
- ideal case situation $T(n) = 2T(\frac{n}{2}) + bn$ why?
- not so ideal, but "balanced" cases, for some $\alpha + \beta = 1$,

$$T(n) = T(\alpha n) + T(\beta n) + bn$$

e.g., $\alpha = \frac{1}{5}, \beta = \frac{4}{5}$.

1. Power of divide-and-conquer

Estimation of complexity for `quicksort`;

- worst case situation $T(n) = T(n-1) + bn$ why?
- ideal case situation $T(n) = 2T(\frac{n}{2}) + bn$ why?
- not so ideal, but "balanced" cases, for some $\alpha + \beta = 1$,

$$T(n) = T(\alpha n) + T(\beta n) + bn$$

e.g., $\alpha = \frac{1}{5}, \beta = \frac{4}{5}$.

The final position of pivot e is crucial!
But it is hard to guarantee "balanced cases" .

2. Quick sort and randomized algorithms

2. Quick sort and randomized algorithms

- With high probability pivots are good,

2. Quick sort and randomized algorithms

- With high probability pivots are good, why?

2. Quick sort and randomized algorithms

- With high probability pivots are good, why?

  let e is a pivot, $P(rank(e) = k) = \frac{1}{n}$, $k = 1, 2, \ldots, n$;

2. Quick sort and randomized algorithms

- With high probability pivots are good, why?

  let e is a pivot, $P(rank(e) = k) = \frac{1}{n}$, $k = 1, 2, \ldots, n$;

  $P(\frac{n}{10} < rank(e) \leq \frac{9n}{10}) = \frac{8n}{10} \times \frac{1}{n} = 4/5$

2. Quick sort and randomized algorithms

- With high probability pivots are good, why?

  let e is a pivot, $P(rank(e) = k) = \frac{1}{n}$, $k = 1, 2, \ldots, n$;

  $P(\frac{n}{10} < rank(e) \leq \frac{9n}{10}) = \frac{8n}{10} \times \frac{1}{n} = 4/5$
- compute averaged time for QuickSort:

2. Quick sort and randomized algorithms

- With high probability pivots are good, why?

  let e is a pivot, $P(rank(e) = k) = \frac{1}{n}$, $k = 1, 2, \ldots, n$;

  $P(\frac{n}{10} < rank(e) \leq \frac{9n}{10}) = \frac{8n}{10} \times \frac{1}{n} = 4/5$

- compute averaged time for QuickSort:

  sum of all times on m runs $\div m$.

2. Quick sort and randomized algorithms

- With high probability pivots are good, why?

  let e is a pivot, $P(rank(e) = k) = \frac{1}{n}$, $k = 1, 2, \ldots, n$;

  $P(\frac{n}{10} < rank(e) \leq \frac{9n}{10}) = \frac{8n}{10} \times \frac{1}{n} = 4/5$
- compute averaged time for QuickSort:

  sum of all times on m runs $\div m$.
- instead, compute **expected time** directly!

2. Quick sort and randomized algorithms

- With high probability pivots are good, why?

  let e is a pivot, $P(rank(e) = k) = \frac{1}{n}$, $k = 1, 2, \dots, n$;

  $P(\frac{n}{10} < rank(e) \leq \frac{9n}{10}) = \frac{8n}{10} \times \frac{1}{n} = 4/5$

- compute averaged time for QuickSort:

  sum of all times on m runs $\div m$.

- instead, compute **expected time** directly!

> Claim: Searching for a key in an unsorted list of n elements uses $\frac{n}{2} + 1$ comparisons **in average**.

2. Quick sort and randomized algorithms

Prove averaged time complexity

2. Quick sort and randomized algorithms

Prove averaged time complexity

- deterministic algorithms on randomized data;

2. Quick sort and randomized algorithms

Prove averaged time complexity

- deterministic algorithms on randomized data;

  is the same as:

2. Quick sort and randomized algorithms

Prove averaged time complexity

- deterministic algorithms on randomized data;

  is the same as:

- randomized algorithms on deterministic data;

2. Quick sort and randomized algorithms

Prove averaged time complexity

- deterministic algorithms on randomized data;

  is the same as:

- randomized algorithms on deterministic data;

  randomized linear search ?,

2. Quick sort and randomized algorithms

Prove averaged time complexity

- deterministic algorithms on randomized data;

  is the same as:

- randomized algorithms on deterministic data;

  randomized linear search ?, shuffling data?

2. Quick sort and randomized algorithms

2. Quick sort and randomized algorithms

- define random variable X for number of comparisons,

$$X = k \text{ iff the } k^{\mathrm{th}} \text{ element is the key, } k = 1, 2, \ldots, n$$

2. Quick sort and randomized algorithms

- define random variable X for number of comparisons,

$$X = k \text{ iff the } k^{\text{th}} \text{ element is the key, } k = 1, 2, \ldots, n$$

- averaged number of comparisons

$$\begin{aligned} E[X] &= \sum_{k=1}^{n} P(X = k) \times k \\ &= \sum_{k=1}^{n} \frac{1}{n} k \\ &= \frac{1}{n} \sum_{k=1}^{n} k \\ &= \frac{1}{n} \frac{n}{2}(n+1) \\ &\leq \frac{n}{2} + 1 \end{aligned}$$

2. Quick sort and randomized algorithms

Idea for randomized quick sort:

2. Quick sort and randomized algorithms

Idea for randomized quick sort:

- randomly select a pivot element e from the input list;
  swap e with the last element in L;
- call `partition` to break L into sublists L_h and L_l such that
  $\forall x \in L_h, x > e$
  $\forall x \in L_l, x \leq e$
- recursively sort the two sublists L_h and L_l separately with randomized quick sort;

2. Quick sort and randomized algorithms

Idea for randomized quick sort:

- randomly select a pivot element e from the input list;
  swap e with the last element in L;
- call `partition` to break L into sublists L_h and L_l such that
  $\forall x \in L_h, x > e$
  $\forall x \in L_l, x \leq e$
- recursively sort the two sublists L_h and L_l separately with randomized quick sort;

We estimate averaged time for randomized quick sort

2. Quick sort and randomized algorithms

2. Quick sort and randomized algorithms

- randomly selecting an element from the list to be the pivot is the same as

  with a large probability, the pivot partitions the list into two "balanced" sublists;

2. Quick sort and randomized algorithms

- randomly selecting an element from the list to be the pivot is the same as

  with a large probability, the pivot partitions the list into two "balanced" sublists;

  e.g., with at least 60% probability, the list is partitioned into

  20:80 (80:20) or better

2. Quick sort and randomized algorithms

- randomly selecting an element from the list to be the pivot is the same as

  with a large probability, the pivot partitions the list into two "balanced" sublists;

  e.g., with at least 60% probability, the list is partitioned into
  20:80 (80:20) or better

- the above effect is on every round of recursive call to the random quick sort

2. Quick sort and randomized algorithms

- randomly selecting an element from the list to be the pivot is the same as

  with a large probability, the pivot partitions the list into two "balanced" sublists;

  e.g., with at least 60% probability, the list is partitioned into 20:80 (80:20) or better

- the above effect is on every round of recursive call to the random quick sort

- so the quick sort runs in time $O(n \log_2 n)$

2. Quick sort and randomized algorithms

Theorem: The expected number of comparisons between elements used by the random quick sort algorithm is $O(n \log_2 n)$ on input list of n elements.

2. Quick sort and randomized algorithms

Theorem: The expected number of comparisons between elements used by the random quick sort algorithm is $O(n \log_2 n)$ on input list of n elements.

Proof outline (proof-by-induction):

2. Quick sort and randomized algorithms

Theorem: The expected number of comparisons between elements used by the random quick sort algorithm is $O(n \log_2 n)$ on input list of n elements.

Proof outline (proof-by-induction):

- equivalent to proving $\exists c > 0, n_0 > 0, T(n) \leq cn \log_2 n$, for $n \geq n_0$.

2. Quick sort and randomized algorithms

Theorem: The expected number of comparisons between elements used by the random quick sort algorithm is $O(n \log_2 n)$ on input list of n elements.

Proof outline (proof-by-induction):

- equivalent to proving $\exists c > 0, n_0 > 0, T(n) \le cn \log_2 n$, for $n \ge n_0$.

  - proof for base case $n = ?$;

2. Quick sort and randomized algorithms

Theorem: The expected number of comparisons between elements used by the random quick sort algorithm is $O(n \log_2 n)$ on input list of n elements.

Proof outline (proof-by-induction):

- equivalent to proving $\exists c > 0, n_0 > 0, T(n) \leq cn \log_2 n$, for $n \geq n_0$.

  - proof for base case $n = ?$;

  - assume: for all $n = 1, 2, \ldots, k-1$, $T(n) \leq cn \log_2 n$;

2. Quick sort and randomized algorithms

Theorem: The expected number of comparisons between elements used by the random quick sort algorithm is $O(n \log_2 n)$ on input list of n elements.

Proof outline (proof-by-induction):

- equivalent to proving $\exists c > 0, n_0 > 0, T(n) \le cn \log_2 n$, for $n \ge n_0$.
  - proof for base case $n = ?$;
  - assume: for all $n = 1, 2, \ldots, k-1$, $T(n) \le cn \log_2 n$;
  - inductive step:

$$T(k) = (k-1) + \frac{1}{k} \sum_{i=0}^{k-1} \Big(T(i) + T(k-i-1) \Big)$$

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

$$\le (k-1) + \frac{2}{k} \sum_{i=0}^{k-1} c \times i \times \log_2 i \quad \text{by assumptions}$$

2. Quick sort and randomized algorithms

$$
\begin{aligned}
T(k) &= (k-1) + \frac{1}{k}\sum_{i=0}^{k-1}\Big(T(i) + T(k-i-1)\Big) \\
&= (k-1) + \frac{2}{k}\sum_{i=0}^{k-1} T(i) \\
&\leq (k-1) + \frac{2}{k}\sum_{i=0}^{k-1} c \times i \times \log_2 i \quad \text{by assumptions} \\
&= (k-1) + \frac{2}{k}\Big(\sum_{i=0}^{k/2} c \times i \times \log_2 i + \sum_{i=k/2+1}^{k-1} c \times i \times \log_2 i\Big) \\
&\leq (k-1) + \frac{2}{k}\Big(\sum_{i=0}^{k/2} c \times i \times \log_2 \frac{k}{2} + \sum_{i=k/2+1}^{k-1} c \times i \times \log_2 k\Big) \\
&\leq (k-1) + \frac{2}{k}\Big(\sum_{i=0}^{k/2} c \times i \times (\log_2 k - \log_2 2) + \sum_{i=k/2+1}^{k-1} c \times i \times \log_2 k\Big) \\
&= (k-1) + \frac{2}{k}\sum_{i=0}^{k-1} c \times i \times \log_2 k - \frac{2}{k}\sum_{i=0}^{k/2} c \times i \\
&= c\frac{2}{k}\frac{k-1}{2}(k-1+1)\log_2 k - c\frac{2}{k}\frac{k/2}{2}(k/2+1) + (k-1) \\
&= c(k-1)\log_2 k - c\frac{k/2+1}{2} + (k-1) \\
&\leq ck\log_2 k \quad \text{when } c \geq 4
\end{aligned}
$$

2. Quick sort and randomized algorithms

What algorithms have the following recurrences for running time?

2. Quick sort and randomized algorithms

What algorithms have the following recurrences for running time?

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

2. Quick sort and randomized algorithms

What algorithms have the following recurrences for running time?

- $T(n) = T(n-1) + b$;
- $T(n) = T(n-1) + bn$;

2. Quick sort and randomized algorithms

What algorithms have the following recurrences for running time?

- $T(n) = T(n-1) + b$;
- $T(n) = T(n-1) + bn$;
- $T(n) = T(\frac{n}{2}) + b$;

2. Quick sort and randomized algorithms

What algorithms have the following recurrences for running time?

- $T(n) = T(n-1) + b$;
- $T(n) = T(n-1) + bn$;
- $T(n) = T(\frac{n}{2}) + b$;
- $T(n) = dT(\frac{n}{2}) + bn$; $d = 2, 3, 4$;

2. Quick sort and randomized algorithms

What algorithms have the following recurrences for running time?

- $T(n) = T(n-1) + b$;
- $T(n) = T(n-1) + bn$;
- $T(n) = T(\frac{n}{2}) + b$;
- $T(n) = dT(\frac{n}{2}) + bn$; $d = 2, 3, 4$;
- $T(n) = dT(\frac{n}{2}) + bn^2$; $d = 7, 8$;

2. Quick sort and randomized algorithms

What algorithms have the following recurrences for running time?

- $T(n) = T(n-1) + b$;
- $T(n) = T(n-1) + bn$;
- $T(n) = T(\frac{n}{2}) + b$;
- $T(n) = dT(\frac{n}{2}) + bn$; $d = 2, 3, 4$;
- $T(n) = dT(\frac{n}{2}) + bn^2$; $d = 7, 8$;
- $T(n) = T(\alpha n) + T(\beta n) + bn$, for $\alpha + \beta = 1$

2. Quick sort and randomized algorithms

What algorithms have the following recurrences for running time?

- $T(n) = T(n-1) + b$;
- $T(n) = T(n-1) + bn$;
- $T(n) = T(\frac{n}{2}) + b$;
- $T(n) = dT(\frac{n}{2}) + bn$; $d = 2, 3, 4$;
- $T(n) = dT(\frac{n}{2}) + bn^2$; $d = 7, 8$;
- $T(n) = T(\alpha n) + T(\beta n) + bn$, for $\alpha + \beta = 1$
- what about: $T(n) = T(\alpha n) + T(\beta n) + T(\gamma n) + bn$, for $\alpha + \beta + \gamma = 1$

2. Quick sort and randomized algorithms

What algorithms have the following recurrences for running time?

- $T(n) = T(n-1) + b$;
- $T(n) = T(n-1) + bn$;
- $T(n) = T(\frac{n}{2}) + b$;
- $T(n) = dT(\frac{n}{2}) + bn$; $d = 2, 3, 4$;
- $T(n) = dT(\frac{n}{2}) + bn^2$; $d = 7, 8$;
- $T(n) = T(\alpha n) + T(\beta n) + bn$, for $\alpha + \beta = 1$
- what about: $T(n) = T(\alpha n) + T(\beta n) + T(\gamma n) + bn$, for $\alpha + \beta + \gamma = 1$
- what about $d = 1$, i.e., $T(n) = T(\frac{n}{2}) + bn$;

2. Quick sort and randomized algorithms

What algorithms have the following recurrences for running time?

- $T(n) = T(n-1) + b$;
- $T(n) = T(n-1) + bn$;
- $T(n) = T(\frac{n}{2}) + b$;
- $T(n) = dT(\frac{n}{2}) + bn$; $d = 2, 3, 4$;
- $T(n) = dT(\frac{n}{2}) + bn^2$; $d = 7, 8$;
- $T(n) = T(\alpha n) + T(\beta n) + bn$, for $\alpha + \beta = 1$
- what about: $T(n) = T(\alpha n) + T(\beta n) + T(\gamma n) + bn$, for $\alpha + \beta + \gamma = 1$
- what about $d = 1$, i.e., $T(n) = T(\frac{n}{2}) + bn$;
- more general $T(n) = T(\alpha n) + T(\beta n) + bn$, for $\alpha + \beta < 1$

3. Selection: deterministic and randomized

Problem Selection

Input: a list L and rank k;

Output: the k^{th} smallest element in L;

3. Selection: deterministic and randomized

Problem Selection

Input: a list L and rank k;

Output: the k^{th} smallest element in L;

- with sorting, Selection can be done in time $O(n \log_2 n)$;

3. Selection: deterministic and randomized

Problem Selection
Input: a list L and rank k;
Output: the k^{th} smallest element in L;

- with sorting, Selection can be done in time $O(n \log_2 n)$;
- can we do better?

3. Selection: deterministic and randomized

Problem Selection

Input: a list L and rank k;

Output: the k^{th} smallest element in L;

- with sorting, Selection can be done in time $O(n \log_2 n)$;
- can we do better? yes, in $O(n)$ time!

3. Selection: deterministic and randomized

Problem Selection

Input: a list L and rank k;

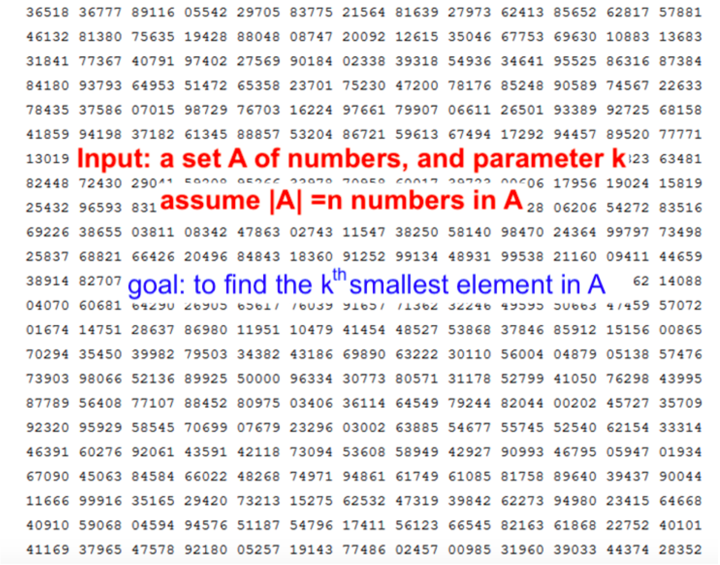
Output: the k^{th} smallest element in L;

- with sorting, Selection can be done in time $O(n \log_2 n)$;
- can we do better? yes, in $O(n)$ time!

  deterministic algorithm: worst case in linear time;
  randomized algorithm: averaged case in linear time;

3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:



3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:

36518	36777	89116	05542	29705	83775	21564	81639	27973	62413	85652	62817	57881
46132	81380	75635	19428	88048	08747	20092	12615	35046	67753	69630	10883	13683
31841	77367	40791	97402	27569	90184	02338	39318	54936	34641	95525	86316	87384
84180	93793	64953	51472	65358	23701	75230	47200	78176	85248	90589	74567	22633
78435	37586	07015	98729	76703	16224	97661	79907	06611	26501	93389	92725	68158
41859	94198	37182	61345	88857	53204	86721	59613	67494	17292	94457	89520	77771
13019	07274	51068	93129	40386	51731	44254	66685	72835	01270	42523	45323	63481
82448	72430	29041	59208	95266	33978	70958	60017	39723	00606	17956	19024	15819
25432	96593	83112	96997	55340	80312	78839	09815	16887	22228	06206	54272	83516
69226	38655	03811	08342	47863	02743	11547	38250	58140	98470	24364	99797	73498
25837	68821	66426	20496	84843	18360	91252	99134	48931	99538	21160	09411	44659
38914	82707	24769	72026	56813	49336	71767	04474	32909	74162	50404	68562	14088
04070	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	57072
01674	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	00865
70294	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	57476
73903	98066	52136	89925	50000	96334	30773	80571	31178	52799	41050	76298	43995
87789	56408	77107	88452	80975	03406	36114	64549	79244	82044	00202	45727	35709
92320	95929	58545	70699	07679	23296	03002	63885	54677	55745	52540	62154	33314
46391	60276	92061	43591	42118	73094	53608	58949	42927	90993	46795	05947	01934
67090	45063	84584	66022	48268	74971	94861	61749	61085	81758	89640	39437	90044
11666	99916	35165	29420	73213	15275	62532	47319	39842	62273	94980	23415	64668
40910	59068	04594	94576	51187	54796	17411	56123	66545	82163	61868	22752	40101
41169	37965	47578	92180	05257	19143	77486	02457	00985	31960	39033	44374	28352

Numbers in A are grouped into n/5 groups, with 5 numbers in each group

3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:

36518	36777	89116	05542	29705	83775	21564	81639	27973	62413	85652	62817	57881
46132	81380	75635	19428	88048	08747	20092	12615	35046	67753	69630	10883	13683
31841	77367	40791	97402	27569	90184	02338	39318	54936	34641	95525	86316	87384
84180	93793	64953	51472	65358	23701	75230	47200	78176	85248	90589	74567	22633
78435	37586	07015	98729	76703	16224	97661	79907	06611	26501	93389	92725	68158
41859	94198	37182	61345	88857	53204	86721	59613	67494	17292	94457	89520	77771
13019	07274	51068	93129	40386	51731	44254	66685	72835	01270	42523	45323	63481
82448	72430	29041	59208	95266	33978	70958	60017	39723	00606	17956	19024	15819
25432	96593	83112	96997	55340	80312	78839	09815	16887	22228	06206	54272	83516
69226	38655	03811	08342	47863	02743	11547	38250	58140	98470	24364	99797	73498
25837	68821	66426	20496	84843	18360	91252	99134	48931	99538	21160	09411	44659
38914	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
04070	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
01674	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
70294	35450	39982	79503	34382	43186	69890	63222	30110	56004	04879	05138	57476
73903	98066	52136	89925	50000	96334	30773	80571	31178	52799	41050	76298	43995
87789	56408	77107	88452	80975	03406	36114	64549	79244	82044	00202	45727	35709
92320	95929	58545	70699	07679	23296	03002	63885	54677	55745	52540	62154	33314
46391	60276	92061	43591	42118	73094	53608	58949	42927	90993	46795	05947	01934
67090	45063	84584	66022	48268	74971	94861	61749	61085	81758	89640	39437	90044
11666	99916	35165	29420	73213	15275	62532	47319	39842	62273	94980	23415	64668
40910	59068	04594	94576	51187	54796	17411	56123	66545	82163	61868	22752	40101
41169	37965	47578	92180	05257	19143	77486	02457	00985	31960	39033	44374	28352

The 3rd largest number in every group is chosen;
all such numbers are placed in new set M

3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:

$M =$

51731	44254	66685	72835	01270
33978	70958	60017	39723	00606
80312	78839	09815	16887	22228
02743	11547	38250	58140	98470
18360	91252	99134	48931	99538
49336	71767	04474	32909	74162
76039	91657	71362	32246	49595
10479	41454	48527	53868	37846

$|M| = n/5$ elements, e.g., 40

3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:

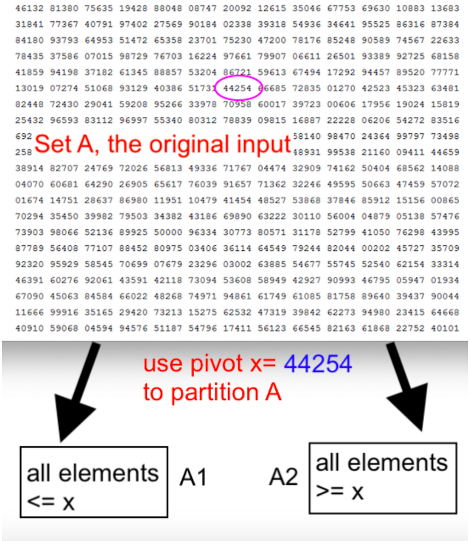
$M =$

51731	44254	66685	72835	01270
33978	70958	60017	39723	00606
80312	78839	09815	16887	22228
02743	11547	38250	58140	98470
18360	91252	99134	48931	99538
49336	71767	04474	32909	74162
76039	91657	71362	32246	49595
10479	41454	48527	53868	37846

Find $(\frac{n}{10})^{\text{th}}$ smallest element, e.g., the 20^{th} if $|M| = n/5 = 40$
let this element be 44254, name it x, the *pivot*

3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:



3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:

Summary of the steps described so far:

3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:

Summary of the steps described so far:

- **input**: list A of n elements, and parameter k;
  (goal: to find k^{th} smallest element from A)

3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:

Summary of the steps described so far:

- **input**: list A of n elements, and parameter k;
  (goal: to find k^{th} smallest element from A)
- group elements in A into groups of 5, resulting in $n/5$ groups;

3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:

Summary of the steps described so far:

- **input**: list A of n elements, and parameter k;
  (goal: to find k^{th} smallest element from A)
- group elements in A into groups of 5, resulting in $n/5$ groups;
- for each group, pick the third largest element;
  put such elements from all groups in M;

3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:

Summary of the steps described so far:

- **input**: list A of n elements, and parameter k;
  (goal: to find k^{th} smallest element from A)
- group elements in A into groups of 5, resulting in $n/5$ groups;
- for each group, pick the third largest element;
  put such elements from all groups in M;
- let $x =$ `selection(M, n/5, n/10)`, a recursive call to `selection`;
  x is the $(\frac{n}{10})^{\text{th}}$ smallest element in M, or median in M;

3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:

Summary of the steps described so far:

- **input**: list A of n elements, and parameter k;
  (goal: to find k^{th} smallest element from A)
- group elements in A into groups of 5, resulting in $n/5$ groups;
- for each group, pick the third largest element;
  put such elements from all groups in M;
- let $x =$ `selection(M, n/5, n/10)`, a recursive call to `selection`;
  x is the $(\frac{n}{10})^{\mathrm{th}}$ smallest element in M, or median in M;
- use x as pivot to partition A into A_1 and A_2, such that
  $\forall y \in A_1, y \leq x$, and $\forall z \in A_2, x < z$

3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:

Continue the idea:

3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:

Continue the idea:

- let $r = rank(x)$ in A, i.e., x is the r^{th} smallest element in A;

3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:

Continue the idea:

- let $r = rank(x)$ in A, i.e., x is the r^{th} smallest element in A;
- if $r = k$ (remember k?), then the algorithm **returns** x, done!

3.1 Deterministic selection

Idea of linear time deterministic `selection(A,n,k)` algorithm:

Continue the idea:

- let $r = rank(x)$ in A, i.e., x is the r^{th} smallest element in A;
- if $r = k$ (remember k?), then the algorithm **returns** x, done!
- otherwise,

  if $k < r$, **return** `selection(A1, r-1, k)`;

  else, **return** `selection(A2, n-r, k-r)`;

3.1 Deterministic selection

Algorithm $\textsc{Select}(A, n, k)$; $\longleftarrow$ finding the k^{th} smallest element in A

1. **if** $n < 140$, simply sort A and **return** $A[k]$;
2. Find a pivot
3. group elements in A into groups of 5 elements;
4. place 3^{rd} largest elements from all groups in list M;
5. $x = \textsc{Select}(M, \frac{n}{5}, \frac{n}{10})$;
6. let $r = rank(x)$ in A;
7. **if** $k = r$, return $A[r]$;
8. partition A into $A_1 = \{y : y \leq x\}$, $A_2 = \{z : x < z\}$;
9. **if** $k < r$, **return** $\textsc{Select}(A_1, r-1, k)$;
10. **else return** $\textsc{Select}(A_2, n-r, k-r)$;

3.1 Deterministic selection

Assume $T(n)$ to be the time function for $\textsc{Select}(A, n, k)$.

Algorithm $\textsc{Select}(A, n, k)$; $\longleftarrow$ time function $T(n)$ on n elements of A

1. **if** $n < 140$, simply sort A and **return** $A[k]$; $\longleftarrow c_1$
2. Find a pivot
3. group elements in A into groups of 5 elements; $\longleftarrow \leq c_3 n$
4. place 3^{rd} largest elements from all groups in list M; $\longleftarrow \leq c_4 n$
5. $x = \textsc{Select}(M, \frac{n}{5}, \frac{n}{10})$; $\longleftarrow T(\frac{n}{5})$
6. let $r = rank(x)$ in A; $\longleftarrow \leq c_6 n$
7. **if** $k = r$, return $A[r]$; $\longleftarrow c_7$
8. partition A into $A_1 = \{y : y \leq x\}$, $A_2 = \{z : x < z\}$; $\longleftarrow \leq c_8 n$
9. **if** $k < r$, **return** $\textsc{Select}(A_1, r-1, k)$; $\longleftarrow T(|A_1|)$
10. **else return** $\textsc{Select}(A_2, n-r, k-r)$; $\longleftarrow T(|A_2|)$

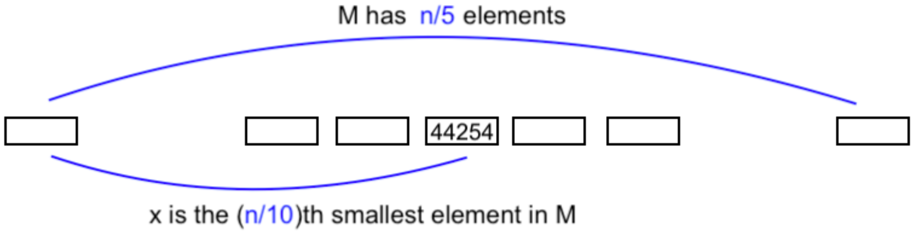
how small can they be?

$$T(n) \leq \begin{cases} c_1 & n < 140 \\ T(\frac{n}{5}) + \max\{T(|A_1|), T(|A_2|)\} + an + b & n \geq 140 \end{cases}$$

3.1 Deterministic selection

44254

3.1 Deterministic selection



3.1 Deterministic selection

3.1 Deterministic selection

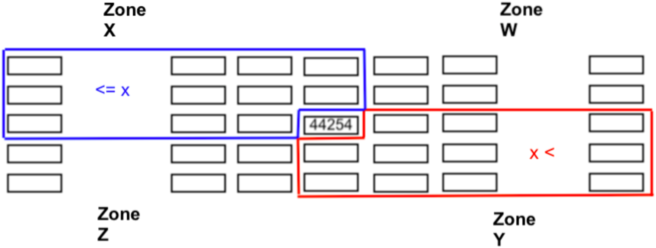
3.1 Deterministic selection

3.1 Deterministic selection

3.1 Deterministic selection

- $|X| = \frac{n}{10} \times 3 - 1; \implies |A_1| \geq \frac{3n}{10} - 1; \implies |A_2| = n - |A_1| - 1 \leq \frac{7n}{10};$

3.1 Deterministic selection



- $|X| = \frac{n}{10} \times 3 - 1; \implies |A_1| \geq \frac{3n}{10} - 1; \implies |A_2| = n - |A_1| - 1 \leq \frac{7n}{10};$
- $|Y| = \frac{n}{10} \times 3 - 1; \implies |A_2| \geq \frac{3n}{10} - 1; \implies |A_1| = n - |A_2| - 1 \leq \frac{7n}{10};$

3.1 Deterministic selection

- $|X| = \frac{n}{10} \times 3 - 1; \Longrightarrow |A_1| \geq \frac{3n}{10} - 1; \Longrightarrow |A_2| = n - |A_1| - 1 \leq \frac{7n}{10};$
- $|Y| = \frac{n}{10} \times 3 - 1; \Longrightarrow |A_2| \geq \frac{3n}{10} - 1; \Longrightarrow |A_1| = n - |A_2| - 1 \leq \frac{7n}{10};$
- $\max\{|A_1|, |A_2|\} \leq \frac{7n}{10};$

$$T(n) \leq \begin{cases} c_1 & n < 140 \\ \cancel{T(\frac{n}{5})}\ T(\frac{2n}{10}) + T(\frac{7n}{10}) + an + b & n \geq 140 \end{cases}$$

3.1 Deterministic selection

Deterministic Selection has time complexity:

$$T(n) \leq \begin{cases} c_1 & n < 140 \\ T(\frac{2n}{10}) + T(\frac{7n}{10}) + an + b & n \geq 140 \end{cases}$$

3.1 Deterministic selection

Deterministic Selection has time complexity:

$$T(n) \leq \begin{cases} c_1 & n < 140 \\ T(\frac{2n}{10}) + T(\frac{7n}{10}) + an + b & n \geq 140 \end{cases}$$

Use structural induction to prove that $T(n) = O(n)$, or equivalently, there is $c > 0$, $T(n) < cn$ for all $n \geq 1$;

3.1 Deterministic selection

Deterministic Selection has time complexity:

$$T(n) \leq \begin{cases} c_1 & n < 140 \\ T(\frac{2n}{10}) + T(\frac{7n}{10}) + an + b & n \geq 140 \end{cases}$$

Use structural induction to prove that $T(n) = O(n)$, or equivalently, there is $c > 0$, $T(n) < cn$ for all $n \geq 1$;

- base case: for all $1 \leq n < 140$, sorting is used.
  It takes $O(140 \log_2 140) \leq d \times 140 \times 7.2 = 1008d$ for some $d > 0$;
  [note that d is the constant associated with time complexity for a merge sort algorithm]
  So it suffices to choose $c \geq 1008d$ to satisfy $1008d \leq c \times n$;

3.1 Deterministic selection

Deterministic Selection has time complexity:

$$T(n) \leq \begin{cases} c_1 & n < 140 \\ T(\frac{2n}{10}) + T(\frac{7n}{10}) + an + b & n \geq 140 \end{cases}$$

Use structural induction to prove that $T(n) = O(n)$, or equivalently, there is $c > 0$, $T(n) < cn$ for all $n \geq 1$;

- base case: for all $1 \leq n < 140$, sorting is used.
  It takes $O(140 \log_2 140) \leq d \times 140 \times 7.2 = 1008d$ for some $d > 0$;
  [note that d is the constant associated with time complexity for a merge sort algorithm]
  So it suffices to choose $c \geq 1008d$ to satisfy $1008d \leq c \times n$;
- assumption: $T(\frac{2n}{10}) \leq c\frac{2n}{10}$, $T(\frac{7n}{10}) \leq c\frac{7n}{10}$;

3.1 Deterministic selection

Deterministic Selection has time complexity:

$$T(n) \leq \begin{cases} c_1 & n < 140 \\ T(\frac{2n}{10}) + T(\frac{7n}{10}) + an + b & n \geq 140 \end{cases}$$

Use structural induction to prove that $T(n) = O(n)$, or equivalently, there is $c > 0$, $T(n) < cn$ for all $n \geq 1$;

- base case: for all $1 \leq n < 140$, sorting is used.
  It takes $O(140 \log_2 140) \leq d \times 140 \times 7.2 = 1008d$ for some $d > 0$;
  [note that d is the constant associated with time complexity for a merge sort algorithm]
  So it suffices to choose $c \geq 1008d$ to satisfy $1008d \leq c \times n$;
- assumption: $T(\frac{2n}{10}) \leq c\frac{2n}{10}$, $T(\frac{7n}{10}) \leq c\frac{7n}{10}$;
- induction: $n \geq 140$,

$$T(n) \leq T(\frac{2n}{10}) + T(\frac{7n}{10}) + an + b$$

3.1 Deterministic selection

$$
\begin{aligned}
T(n) &\le T(\frac{2n}{10}) + T(\frac{7n}{10}) + an + b \\
&\le c\frac{2n}{10} + c\frac{7n}{10} + an + b \\
&= c\frac{9n}{10} + an + b \\
&= cn - c\frac{n}{10} + an + b \\
&\le cn - c\frac{n}{10} + (a+b)n \\
&\le cn \text{ when we choose } c \ge 10(a+b)
\end{aligned}
$$

We have proved that for $c = \max\{10(a+b), 1008d\}$, $T(n) \le cn$ for all $n \ge 1$.

3.2 Randomized algorithm analysis

```
function randomized selection (A, n, k)
{ assume 1 <= k <= n}

randomly pick a position i;
pivot x=A[i];
partition A into A_low and A_high around x
let r = rank of x;
if r == k
    return x;
if k < r
    randomized selection (A_low, r-1, k);
else
    randomized selection (A_high, n-r, k-r);
```

3.2 Randomized algorithm analysis

We analyze the expected (averaged) time for `randomized selection`.

Intuitively,

3.2 Randomized algorithm analysis

We analyze the expected (averaged) time for `randomized selection`.

Intuitively,

- with probability $\frac{1}{n}$, an element is picked;

3.2 Randomized algorithm analysis

We analyze the expected (averaged) time for `randomized selection`.

Intuitively,

- with probability $\frac{1}{n}$, an element is picked;
- with a large chance (80%) that an element with rank between top 11 and 90 is picked;

3.2 Randomized algorithm analysis

We analyze the expected (averaged) time for `randomized selection`.

Intuitively,

- with probability $\frac{1}{n}$, an element is picked;
- with a large chance (80%) that an element with rank between top 11 and 90 is picked;
- this element is used as pivot for partition;

3.2 Randomized algorithm analysis

We analyze the expected (averaged) time for `randomized selection`.

Intuitively,

- with probability $\frac{1}{n}$, an element is picked;
- with a large chance (80%) that an element with rank between top 11 and 90 is picked;
- this element is used as pivot for partition;
- after the partition, with large chance, the sublist has length a fraction of n;

3.2 Randomized algorithm analysis

We analyze the expected (averaged) time for `randomized selection`.

Intuitively,

- with probability $\frac{1}{n}$, an element is picked;
- with a large chance (80%) that an element with rank between top 11 and 90 is picked;
- this element is used as pivot for partition;
- after the partition, with large chance, the sublist has length a fraction of n;
- lead to averaged time $\widetilde{T}(n) = O(n)$ randomized selection algorithm

3.2 Randomized algorithm analysis

Formal analysis: (you read!)

3.2 Randomized algorithm analysis

Formal analysis: (you read!)

- once pivot is picked, algorithm may apply again on one sublist; upper bound time should be time spent on the longer sublists;

$$\begin{aligned}
\widetilde{T}(n) &= \frac{1}{n}\Big(\max\{\widetilde{T}(0), \widetilde{T}(n-1)\} + O(n) + \max\{\widetilde{T}(1), \widetilde{T}(n-2)\} + O(n) \\
&\quad + \cdots + \max\{\widetilde{T}(n-1), \widetilde{T}(0)\} + O(n)\Big) \\
&\leq \frac{2}{n}\Big(\widetilde{T}(n-1) + \widetilde{T}(n-2) + \cdots + \widetilde{T}(\frac{n}{2}) + n \times O(n)\Big) \\
&= \frac{2}{n}\Big(\sum_{j=\frac{n}{2}}^{n-1} \widetilde{T}(j) + O(n^2)\Big)
\end{aligned}$$

3.2 Randomized algorithm analysis

Alternatively, we write

$$\widetilde{T}(n) \leq \frac{2}{n}\left(\sum_{j=\frac{n}{2}}^{n-1} \widetilde{T}(j) + an^2\right)$$

3.2 Randomized algorithm analysis

Alternatively, we write

$$\widetilde{T}(n) \leq \frac{2}{n}\left(\sum_{j=\frac{n}{2}}^{n-1} \widetilde{T}(j) + an^2\right)$$

Claim: $\widetilde{T}(n) = O(n)$, i.e., $\widetilde{T}(n) \leq cn$ for some $c > 0$, when $n \geq 1$.

3.2 Randomized algorithm analysis

Alternatively, we write

$$\widetilde{T}(n) \leq \frac{2}{n}\left(\sum_{j=\frac{n}{2}}^{n-1} \widetilde{T}(j) + an^2\right)$$

Claim: $\widetilde{T}(n) = O(n)$, i.e., $\widetilde{T}(n) \leq cn$ for some $c > 0$, when $n \geq 1$.

Proof (by induction) [Assume that the smallest list length is 1]

3.2 Randomized algorithm analysis

Alternatively, we write

$$\widetilde{T}(n) \le \frac{2}{n}\left(\sum_{j=\frac{n}{2}}^{n-1} \widetilde{T}(j) + an^2\right)$$

Claim: $\widetilde{T}(n) = O(n)$, i.e., $\widetilde{T}(n) \le cn$ for some $c > 0$, when $n \ge 1$.

Proof (by induction) [Assume that the smallest list length is 1]

- base case: $\widetilde{T}(1) = b$ for some constant time b;
  we choose $c \ge b$ such that $b \le c \times 1$;

3.2 Randomized algorithm analysis

Alternatively, we write

$$\widetilde{T}(n) \le \frac{2}{n}\left(\sum_{j=\frac{n}{2}}^{n-1} \widetilde{T}(j) + an^2\right)$$

Claim: $\widetilde{T}(n) = O(n)$, i.e., $\widetilde{T}(n) \le cn$ for some $c > 0$, when $n \ge 1$.

Proof (by induction) [Assume that the smallest list length is 1]

- base case: $\widetilde{T}(1) = b$ for some constant time b;
  we choose $c \ge b$ such that $b \le c \times 1$;
- base case: $\widetilde{T}(2) \le \frac{2}{2}\big(\widetilde{T}(1) + a \times 2^2\big) = b + 4a$;
  we choose $c \ge b + 4a$ such that $b + 4a \le c \times 2$;

3.2 Randomized algorithm analysis

Alternatively, we write

$$\widetilde{T}(n) \le \frac{2}{n}\left(\sum_{j=\frac{n}{2}}^{n-1} \widetilde{T}(j) + an^2\right)$$

Claim: $\widetilde{T}(n) = O(n)$, i.e., $\widetilde{T}(n) \le cn$ for some $c > 0$, when $n \ge 1$.

Proof (by induction) [Assume that the smallest list length is 1]

- base case: $\widetilde{T}(1) = b$ for some constant time b;
  we choose $c \ge b$ such that $b \le c \times 1$;
- base case: $\widetilde{T}(2) \le \frac{2}{2}\left(\widetilde{T}(1) + a \times 2^2\right) = b + 4a$;
  we choose $c \ge b + 4a$ such that $b + 4a \le c \times 2$; [not needed]

3.2 Randomized algorithm analysis

Alternatively, we write

$$\widetilde{T}(n) \le \frac{2}{n}\left(\sum_{j=\frac{n}{2}}^{n-1} \widetilde{T}(j) + an^2\right)$$

Claim: $\widetilde{T}(n) = O(n)$, i.e., $\widetilde{T}(n) \le cn$ for some $c > 0$, when $n \ge 1$.

Proof (by induction) [Assume that the smallest list length is 1]

- base case: $\widetilde{T}(1) = b$ for some constant time b;
  we choose $c \ge b$ such that $b \le c \times 1$;
- base case: $\widetilde{T}(2) \le \frac{2}{2}\big(\widetilde{T}(1) + a \times 2^2\big) = b + 4a$;
  we choose $c \ge b + 4a$ such that $b + 4a \le c \times 2$; [not needed]
- assumption:

$$\widetilde{T}(k-1) \le c(k-1), \widetilde{T}(k-2) \le c(k-2), \dots, \widetilde{T}(\frac{k}{2}) \le c\frac{k}{2}$$

3.2 Randomized algorithm analysis

3.2 Randomized algorithm analysis

- induction:

$$\widetilde{T}(k) \leq \frac{2}{k}\left(\sum_{j=\frac{k}{2}}^{k-1} \widetilde{T}(j) + ak^2\right) \leq \frac{2}{k}\left(\sum_{j=\frac{k}{2}}^{k-1} c \times j + ak^2\right)$$

3.2 Randomized algorithm analysis

- induction:

$$\widetilde{T}(k) \leq \frac{2}{k}\left(\sum_{j=\frac{k}{2}}^{k-1} \widetilde{T}(j) + ak^2\right) \leq \frac{2}{k}\left(\sum_{j=\frac{k}{2}}^{k-1} c \times j + ak^2\right)$$

where

$$\begin{aligned}
\sum_{j=\frac{k}{2}}^{k-1} c \times j &= c\left(\frac{k}{2} + (\frac{k}{2}+1) + \cdots + (k-1)\right) \\
&= c\left(\frac{k}{2} + (\frac{k}{2}+1) + (\frac{k}{2}+2) \cdots + (\frac{k}{2}+\frac{k}{2}-1)\right) \\
&= c\left(\frac{k}{2} \times \frac{k}{2} + 1 + 2 + \cdots + (\frac{k}{2}-1)\right) \\
&\leq c\left(\frac{k^2}{4} + \frac{\frac{k}{2}-1}{2}(\frac{k}{2}-1+1)\right) \\
&= c\left(\frac{k^2}{4} + \frac{k(k-2)}{8}\right) \\
&= c\frac{3k^2-2k}{8} \leq c\frac{3k^2}{8}
\end{aligned}$$

3.2 Randomized algorithm analysis

$$
\begin{aligned}
\widetilde{T}(k) &\leq \frac{2}{k}\left(\sum_{j=\frac{k}{2}}^{k-1} c \times j + ak^2\right) \\
&\leq \frac{2}{k}\left(c\frac{3k^2}{8} + ak^2\right) \\
&= \frac{3ck + 8ak}{4} \leftarrow \text{we want this } \leq ck
\end{aligned}
$$

3.2 Randomized algorithm analysis

$$
\begin{aligned}
\widetilde{T}(k) &\leq \frac{2}{k}\left(\sum_{j=\frac{k}{2}}^{k-1} c \times j + ak^2\right) \\
&\leq \frac{2}{k}\left(c\frac{3k^2}{8} + ak^2\right) \\
&= \frac{3ck + 8ak}{4} \leftarrow \text{we want this } \leq ck
\end{aligned}
$$

That is

$$
\frac{3ck + 8ak}{4} \leq ck
$$

3.2 Randomized algorithm analysis

$$\widetilde{T}(k) \leq \frac{2}{k}\left(\sum_{j=\frac{k}{2}}^{k-1} c \times j + ak^2\right)$$
$$\leq \frac{2}{k}\left(c\frac{3k^2}{8} + ak^2\right)$$
$$= \frac{3ck + 8ak}{4} \leftarrow \text{we want this } \leq ck$$

That is

$$\frac{3ck + 8ak}{4} \leq ck$$

Solve the inequality, we get $ck \geq 8ak$, thus choose $c \geq 8a$.

3.2 Randomized algorithm analysis

$$\widetilde{T}(k) \leq \frac{2}{k}\left(\sum_{j=\frac{k}{2}}^{k-1} c \times j + ak^2\right)$$

$$\leq \frac{2}{k}\left(c\frac{3k^2}{8} + ak^2\right)$$

$$= \frac{3ck + 8ak}{4} \leftarrow \text{we want this } \leq ck$$

That is

$$\frac{3ck + 8ak}{4} \leq ck$$

Solve the inequality, we get $ck \geq 8ak$, thus choose $c \geq 8a$.

- Conclusion: we proved that for $c = \max\{b + 4a, 8a\}$, $n \geq 1$,

$$\widetilde{T}(n) \leq cn$$

4. Complexity lower bounds

4. Complexity lower bounds

- Complexity upper bound

$$T(n) \leq U(n)$$

4. Complexity lower bounds

- Complexity upper bound

$$T(n) \leq U(n) \implies T(n) = O(U(n))$$

4. Complexity lower bounds

- Complexity upper bound

$$T(n) \leq U(n) \implies T(n) = O(U(n))$$

"on all inputs x of size n"

4. Complexity lower bounds

- Complexity upper bound

$$T(n) \le U(n) \implies T(n) = O(U(n))$$

"on all inputs x of size n"

– same as "on worst case inputs of size n"

$$T(n) = T_{\mathrm{w.c}}(n) \le U(n)$$

4. Complexity lower bounds

- Complexity upper bound

$$T(n) \leq U(n) \implies T(n) = O(U(n))$$

"on all inputs x of size n"

– same as "on worst case inputs of size n"

$$T(n) = T_{\mathrm{w.c}}(n) \leq U(n)$$

- Complexity lower bound

$$L(n) \leq T_{w.c}(n)$$

4. Complexity lower bounds

- Complexity upper bound

$$T(n) \leq U(n) \implies T(n) = O(U(n))$$

"on all inputs x of size n"

– same as "on worst case inputs of size n"

$$T(n) = T_{\mathrm{w.c}}(n) \leq U(n)$$

- Complexity lower bound

$$L(n) \leq T_{w.c}(n) \implies T(n) = \Omega(L(n))$$

4. Complexity lower bounds

- Complexity upper bound

$$T(n) \leq U(n) \implies T(n) = O(U(n))$$

"on all inputs x of size n"

– same as "on worst case inputs of size n"

$$T(n) = T_{\mathrm{w.c}}(n) \leq U(n)$$

- Complexity lower bound

$$L(n) \leq T_{w.c}(n) \implies T(n) = \Omega(L(n))$$

"on worst case of inputs of size n"

(**NOT necessarily all inputs** of size n)

4. Complexity lower bounds

Example: Insertion Sort Algorithm

4. Complexity lower bounds

Example: Insertion Sort Algorithm

- upper bound $T(n) = O(n^2)$;

4. Complexity lower bounds

Example: Insertion Sort Algorithm

- upper bound $T(n) = O(n^2)$; can we say $T(n) = O(n^3)$?

4. Complexity lower bounds

Example: Insertion Sort Algorithm

- upper bound $T(n) = O(n^2)$; can we say $T(n) = O(n^3)$?
- lower bound $T(n) = \Omega$

4. Complexity lower bounds

Example: Insertion Sort Algorithm

- upper bound $T(n) = O(n^2)$; can we say $T(n) = O(n^3)$?
- lower bound $T(n) = \Omega(n^2)$;

4. Complexity lower bounds

Example: Insertion Sort Algorithm

- upper bound $T(n) = O(n^2)$; can we say $T(n) = O(n^3)$?
- lower bound $T(n) = \Omega(n^2)$; can we say $T(n) = \Omega(n)$?

4. Complexity lower bounds

Example: Insertion Sort Algorithm

- upper bound $T(n) = O(n^2)$; can we say $T(n) = O(n^3)$?
- lower bound $T(n) = \Omega(n^2)$; can we say $T(n) = \Omega(n)$?

Example: estimate complexity upper and lower bounds for
`Recursive Fibonacci` Algorithm

4. Complexity lower bounds

(1) Lower bound of an algorithm

4. Complexity lower bounds

(1) Lower bound of an algorithm

In contrast to upper bounds, a lower bound $L(n)$ for time function $T(n)$ of a known algorithm is such that

$$T(n) = \Omega(L(n))$$

4. Complexity lower bounds

(1) Lower bound of an algorithm

In contrast to upper bounds, a lower bound $L(n)$ for time function $T(n)$ of a known algorithm is such that

$$T(n) = \Omega(L(n))$$

- for example, we can prove for `Binary Search` that $T(n) = \Omega(\log_2 n)$

4. Complexity lower bounds

(1) Lower bound of an algorithm

In contrast to upper bounds, a lower bound $L(n)$ for time function $T(n)$ of a known algorithm is such that

$$T(n) = \Omega(L(n))$$

- for example, we can prove for `Binary Search` that $T(n) = \Omega(\log_2 n)$

  That is, there are constants $c > 0$, $n_0 > 0$ such that
  $T(n) \geq c \log_2 n$ holds for $n \geq n_0$.

4. Complexity lower bounds

(1) Lower bound of an algorithm

In contrast to upper bounds, a lower bound $L(n)$ for time function $T(n)$ of a known algorithm is such that

$$T(n) = \Omega(L(n))$$

- for example, we can prove for `Binary Search` that $T(n) = \Omega(\log_2 n)$

  That is, there are constants $c > 0$, $n_0 > 0$ such that

  $T(n) \geq c \log_2 n$ holds for $n \geq n_0$.

  You can use induction to do the proof.

4. Complexity lower bounds

(1) Lower bound of an algorithm

In contrast to upper bounds, a lower bound $L(n)$ for time function $T(n)$ of a known algorithm is such that

$$T(n) = \Omega(L(n))$$

- for example, we can prove for `Binary Search` that $T(n) = \Omega(\log_2 n)$

  That is, there are constants $c > 0$, $n_0 > 0$ such that
  $T(n) \geq c \log_2 n$ holds for $n \geq n_0$.
  You can use induction to do the proof.

- another example, for `Quick sort`, $T(n) = \Omega(n^2)$,

4. Complexity lower bounds

(1) Lower bound of an algorithm

In contrast to upper bounds, a lower bound $L(n)$ for time function $T(n)$ of a known algorithm is such that

$$T(n) = \Omega(L(n))$$

- for example, we can prove for `Binary Search` that $T(n) = \Omega(\log_2 n)$

  That is, there are constants $c > 0$, $n_0 > 0$ such that

  $T(n) \geq c \log_2 n$ holds for $n \geq n_0$.

  You can use induction to do the proof.

- another example, for `Quick sort`, $T(n) = \Omega(n^2)$, because the given list may already be sorted or reversely sorted,

4. Complexity lower bounds

(1) Lower bound of an algorithm

In contrast to upper bounds, a lower bound $L(n)$ for time function $T(n)$ of a known algorithm is such that

$$T(n) = \Omega(L(n))$$

- for example, we can prove for `Binary Search` that $T(n) = \Omega(\log_2 n)$

  That is, there are constants $c > 0$, $n_0 > 0$ such that
  $T(n) \geq c \log_2 n$ holds for $n \geq n_0$.
  You can use induction to do the proof.

- another example, for `Quick sort`, $T(n) = \Omega(n^2)$, because the given list may already be sorted or reversely sorted, resulting in

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

4. Complexity lower bounds

(2) Lower bound of a problem

4. Complexity lower bounds

(2) Lower bound of a problem

More interestingly, lower bounds can be established for certain problems, regardless of algorithms.

4. Complexity lower bounds

(2) Lower bound of a problem

More interestingly, lower bounds can be established for certain problems, regardless of algorithms.

- Sorting can be solved by `insertion sort` in time $O(n^2)$;

4. Complexity lower bounds

(2) Lower bound of a problem

More interestingly, lower bounds can be established for certain problems, regardless of algorithms.

- Sorting can be solved by `insertion sort` in time $O(n^2)$;
- we can do better: using `merge sort` in time $O(n \log_2 n)$;

4. Complexity lower bounds

(2) Lower bound of a problem

More interestingly, lower bounds can be established for certain problems, regardless of algorithms.

- Sorting can be solved by `insertion sort` in time $O(n^2)$;
- we can do better: using `merge sort` in time $O(n \log_2 n)$;
- can we do better?

4. Complexity lower bounds

(2) Lower bound of a problem

More interestingly, lower bounds can be established for certain problems, regardless of algorithms.

- Sorting can be solved by `insertion sort` in time $O(n^2)$;
- we can do better: using `merge sort` in time $O(n \log_2 n)$;
- can we do better?
- or can we prove that Sorting has lower bound $\Omega(n \log_2 n)$ regardless what algorithms we may have?

4. Complexity lower bounds

(2) Lower bound of a problem

More interestingly, lower bounds can be established for certain problems, regardless of algorithms.

- Sorting can be solved by `insertion sort` in time $O(n^2)$;
- we can do better: using `merge sort` in time $O(n \log_2 n)$;
- can we do better?
- or can we prove that Sorting has lower bound $\Omega(n \log_2 n)$ regardless what algorithms we may have?
- apparently, proving lower bounds for Sorting cannot rely on creating new algorithms.

4. Complexity lower bounds

What is the least number of comparisons needed to find the maximum from n integer elements?

4. Complexity lower bounds

What is the least number of comparisons needed to find the maximum from n integer elements?

- since every element needs to participate in a comparison, $\geq \frac{n}{2}$;

4. Complexity lower bounds

What is the least number of comparisons needed to find the maximum from n integer elements?

- since every element needs to participate in a comparison, $\geq \frac{n}{2}$;
- can we prove a higher lower bound?

4. Complexity lower bounds

What is the least number of comparisons needed to find the maximum from n integer elements?

- since every element needs to participate in a comparison, $\geq \frac{n}{2}$;
- can we prove a higher lower bound? yes!

  use a graph representation to analyze [explained in the class].

4. Complexity lower bounds

What is the least number of comparisons needed to find the maximum from n integer elements?

- since every element needs to participate in a comparison, $\geq \frac{n}{2}$;
- can we prove a higher lower bound? yes!

  use a graph representation to analyze [explained in the class].
  conclusion: $n - 1$ comparisons are necessary.

4. Complexity lower bounds

What is the least number of comparisons needed to find the maximum from n integer elements?

- since every element needs to participate in a comparison, $\geq \frac{n}{2}$;
- can we prove a higher lower bound? yes!

  use a graph representation to analyze [explained in the class].
  conclusion: $n-1$ comparisons are necessary.
- but $n-1$ comparisons are also sufficient (enough), so the lower bound $n-1$ is said **optimal**
  (cannot be further improved)

4. Complexity lower bounds

Claim: Sorting problem has lower bound $\Omega(n \log_2 n)$ on comparison-based models (comparisons are done between elements).

4. Complexity lower bounds

Claim: Sorting problem has lower bound $\Omega(n \log_2 n)$ on comparison-based models (comparisons are done between elements).

- we examine "any given algorithm";

4. Complexity lower bounds

Claim: Sorting problem has lower bound $\Omega(n \log_2 n)$ on comparison-based models (comparisons are done between elements).

- we examine "any given algorithm";
- it is treated as a blackbox; with input L and output sorted L;

4. Complexity lower bounds

Claim: Sorting problem has lower bound $\Omega(n \log_2 n)$ on comparison-based models (comparisons are done between elements).

- we examine "any given algorithm";
- it is treated as a blackbox; with input L and output sorted L;
- no info about the inside of the algorithm is known/assumed;

4. Complexity lower bounds

Claim: Sorting problem has lower bound $\Omega(n \log_2 n)$ on comparison-based models (comparisons are done between elements).

- we examine "any given algorithm";
- it is treated as a blackbox; with input L and output sorted L;
- no info about the inside of the algorithm is known/assumed;
- but we know

$$(a_1, \ldots, a_n) \Longrightarrow \boxed{\text{black box}} \Longrightarrow (a'_1, \ldots, a'_n)$$

where $(a'_1, \ldots, a'_n)$ is a rearrangement (*permutation*) of $(a_1, \ldots, a_n)$ such that

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n$$

4. Complexity lower bounds

$$(a_1, a_2, a_3, a_4)$$

$$\Downarrow$$

$$\boxed{\text{black box}}$$

$$\Downarrow$$

$$(a'_1, a'_2, a'_3, a'_4)$$

4. Complexity lower bounds

$$(a_1, a_2, a_3, a_4)$$

$$\Downarrow$$

$$\boxed{\text{black box}}$$

$$\Downarrow$$

$$(a'_1, a'_2, a'_3, a'_4)$$

to make the algorithm work, it has to accommodate

$(10, 20, 30, 40) \Longrightarrow (10, 20, 30, 40)$;

4. Complexity lower bounds

$$(a_1, a_2, a_3, a_4)$$

$$\Downarrow$$

$$\boxed{\text{black box}}$$

$$\Downarrow$$

$$(a'_1, a'_2, a'_3, a'_4)$$

to make the algorithm work, it has to accommodate

$(10, 20, 30, 40) \Longrightarrow (10, 20, 30, 40);$

$(10, 20, 40, 30) \Longrightarrow (10, 20, 30, 40);$

4. Complexity lower bounds

$$(a_1, a_2, a_3, a_4)$$

$$\Downarrow$$

$$\boxed{\text{black box}}$$

$$\Downarrow$$

$$(a'_1, a'_2, a'_3, a'_4)$$

to make the algorithm work, it has to accommodate

$(10, 20, 30, 40) \Longrightarrow (10, 20, 30, 40)$;

$(10, 20, 40, 30) \Longrightarrow (10, 20, 30, 40)$;

$(40, 20, 10, 30) \Longrightarrow (10, 20, 30, 40)$;

4. Complexity lower bounds

$$(a_1, a_2, a_3, a_4)$$
$$\Downarrow$$
$$\boxed{\text{black box}}$$
$$\Downarrow$$
$$(a'_1, a'_2, a'_3, a'_4)$$

to make the algorithm work, it has to accommodate

$(10, 20, 30, 40) \Longrightarrow (10, 20, 30, 40);$

$(10, 20, 40, 30) \Longrightarrow (10, 20, 30, 40);$

$(40, 20, 10, 30) \Longrightarrow (10, 20, 30, 40);$

$\ldots$

4. Complexity lower bounds

$$(a_1, a_2, a_3, a_4)$$

$$\Downarrow$$

$$\boxed{\text{black box}}$$

$$\Downarrow$$

$$(a_1', a_2', a_3', a_4')$$

to make the algorithm work, it has to accommodate

$(10, 20, 30, 40) \Longrightarrow (10, 20, 30, 40)$;
$(10, 20, 40, 30) \Longrightarrow (10, 20, 30, 40)$;
$(40, 20, 10, 30) \Longrightarrow (10, 20, 30, 40)$;
$\ldots$

- there are $4!$ scenarios of inputs; implication?

4. Complexity lower bounds

$$(a_1, a_2, a_3, a_4)$$

$$\Downarrow$$

$$\boxed{\text{black box}}$$

$$\Downarrow$$

$$(a'_1, a'_2, a'_3, a'_4)$$

to make the algorithm work, it has to accommodate

$(10, 20, 30, 40) \Longrightarrow (10, 20, 30, 40)$;
$(10, 20, 40, 30) \Longrightarrow (10, 20, 30, 40)$;
$(40, 20, 10, 30) \Longrightarrow (10, 20, 30, 40)$;
$\ldots$

- there are $4!$ scenarios of inputs; implication?
- the algorithm needs to accommodate these $4!$ scenarios; meaning?

4. Complexity lower bounds

$$(a_1, a_2, a_3, a_4)$$
$$\Downarrow$$
$$\boxed{\text{black box}}$$
$$\Downarrow$$
$$(a'_1, a'_2, a'_3, a'_4)$$

to make the algorithm work, it has to accommodate

$$(10, 20, 30, 40) \Longrightarrow (10, 20, 30, 40);$$
$$(10, 20, 40, 30) \Longrightarrow (10, 20, 30, 40);$$
$$(40, 20, 10, 30) \Longrightarrow (10, 20, 30, 40);$$
$$\ldots$$

- there are $4!$ scenarios of inputs; implication?
- the algorithm needs to accommodate these $4!$ scenarios; meaning? it has at least $4!$ different "output ports".

4. Complexity lower bounds

Now we examine comparison-based algorithms:

4. Complexity lower bounds

Now we examine comparison-based algorithms:

An algorithm can be abstracted as a *decision tree*:

4. Complexity lower bounds

Now we examine comparison-based algorithms:

An algorithm can be abstracted as a *decision tree*:

- each step is a comparison "$\leq$" between 2 elements in the input;

4. Complexity lower bounds

Now we examine comparison-based algorithms:

An algorithm can be abstracted as a *decision tree*:

- each step is a comparison "$\leq$" between 2 elements in the input;
- there are two possible outcomes YES/NO from a comparisons;

4. Complexity lower bounds

Now we examine comparison-based algorithms:

An algorithm can be abstracted as a *decision tree*:

- each step is a comparison "$\leq$" between 2 elements in the input;
- there are two possible outcomes YES/NO from a comparisons;
- other operations are neglected in analysis;

4. Complexity lower bounds

Now we examine comparison-based algorithms:

An algorithm can be abstracted as a *decision tree*:

- each step is a comparison "$\leq$" between 2 elements in the input;
- there are two possible outcomes YES/NO from a comparisons;
- other operations are neglected in analysis;
- each leaf is an "output port";

4. Complexity lower bounds

Now we examine comparison-based algorithms:

An algorithm can be abstracted as a *decision tree*:

- each step is a comparison "$\leq$" between 2 elements in the input;
- there are two possible outcomes YES/NO from a comparisons;
- other operations are neglected in analysis;
- each leaf is an "output port";
- worst time is the longest path from the root to a leaf (depth of tree);

4. Complexity lower bounds

Now we examine comparison-based algorithms:

An algorithm can be abstracted as a *decision tree*:

4. Complexity lower bounds

Now we examine comparison-based algorithms:

An algorithm can be abstracted as a *decision tree*:

- number of leaves: $\geq n!$ on input L of n elements;

4. Complexity lower bounds

Now we examine comparison-based algorithms:

An algorithm can be abstracted as a *decision tree*:

- number of leaves: $\geq n!$ on input L of n elements;
- depth d vs leaves l : $d \geq \log_2 l$ on binary tree;

4. Complexity lower bounds

Now we examine comparison-based algorithms:

An algorithm can be abstracted as a *decision tree*:

- number of leaves: $\geq n!$ on input L of n elements;
- depth d vs leaves l : $d \geq \log_2 l$ on binary tree;
  so depth of the decision tree is $\geq \log_2(n!)$;

4. Complexity lower bounds

Now we examine comparison-based algorithms:

An algorithm can be abstracted as a *decision tree*:

- number of leaves: $\geq n!$ on input L of n elements;
- depth d vs leaves l : $d \geq \log_2 l$ on binary tree;
  so depth of the decision tree is $\geq \log_2(n!)$;
- because $n! \geq \frac{n}{2}(\frac{n}{2}+1)\dots n \geq (\frac{n}{2})^{\frac{n}{2}}$;

4. Complexity lower bounds

Now we examine comparison-based algorithms:

An algorithm can be abstracted as a *decision tree*:

- number of leaves: $\geq n!$ on input L of n elements;
- depth d vs leaves l : $d \geq \log_2 l$ on binary tree;
  so depth of the decision tree is $\geq \log_2(n!)$;
- because $n! \geq \frac{n}{2}(\frac{n}{2}+1)\dots n \geq (\frac{n}{2})^{\frac{n}{2}}$;

$$\log_2(n!) \geq \frac{n}{2}\log_2(\frac{n}{2}) = \Omega(n\log_2 n)$$

4. Complexity lower bounds

Now we examine comparison-based algorithms:

An algorithm can be abstracted as a *decision tree*:

- number of leaves: $\geq n!$ on input L of n elements;
- depth d vs leaves l : $d \geq \log_2 l$ on binary tree;
  so depth of the decision tree is $\geq \log_2(n!)$;
- because $n! \geq \frac{n}{2}(\frac{n}{2}+1)\dots n \geq (\frac{n}{2})^{\frac{n}{2}}$;

$$\log_2(n!) \geq \frac{n}{2}\log_2(\frac{n}{2}) = \Omega(n\log_2 n)$$

Claim: all comparison-based algorithms need time $\Omega(n\log_2 n)$ to solve the sorting problem.

4. Complexity lower bounds

Consequence: `Merge Sort` is optimal as a comparison-based algorithm for Sorting.

4. Complexity lower bounds

Consequence: `Merge Sort` is optimal as a comparison-based algorithm for Sorting.

Claim: All comparison-based algorithms need time $\Omega(\log_2 n)$ to solve the problem of "searching a list for a key" (regardless if the list is sorted or not).

4. Complexity lower bounds

Consequence: `Merge Sort` is optimal as a comparison-based algorithm for Sorting.

Claim: All comparison-based algorithms need time $\Omega(\log_2 n)$ to solve the problem of "searching a list for a key" (regardless if the list is sorted or not).

- (If proved) it means Binary Search algorithms are optimal.

4. Complexity lower bounds

Consequence: `Merge Sort` is optimal as a comparison-based algorithm for Sorting.

Claim: All comparison-based algorithms need time $\Omega(\log_2 n)$ to solve the problem of "searching a list for a key" (regardless if the list is sorted or not).

- (If proved) it means Binary Search algorithms are optimal.
- Proof (homework question)