Introduction to Algorithms: Lecture 2

Xue Chen
xuechen1989@ustc.edu.cn
2025 spring in

Homework 1

1. HW1 will be out this weekend and due on next Thursday.
2. VERY challenging — start it early!

Homework 1

1. HW1 will be out this weekend and due on next Thursday.
2. VERY challenging — start it early!
3. Grading policy: 75% correctness, 25% clarity:

   证明/计算：思路清晰，组织明确，说明关键步骤/性质，语言精炼

4. While you are encouraged to discuss it before writing, you must write it by yourself.

- 但关键想法的执行、以及作业文本的写作必须独立完成，并在作业中致谢（ACKNOWLEDGE）所有参与讨论的人
- 不允许其他任何形式的合作——尤其是与已经完成作业的同学“讨论”。
- 如果发现互相抄袭行为，抄袭和被抄袭双方的成绩都将被取消。因此请主动防止自己的作业被他人抄袭。
- 禁止在网上求助，搜索答案，询问已修过的同学等行为。如有问题，可以来我和助教的答疑时间

Outline

Introduction

Sort

Given an array A of n numbers $A[1], \ldots, A[n]$, sort them in order.

$$\text{Input: } n = 5, A = [5, 1, 2, 3, 0].$$
$$\text{Output: } A = [0, 1, 2, 3, 5].$$

Introduction

Sort

Given an array *A* of n numbers $A[1], \ldots, A[n]$, sort them in order.

$$\text{Input: } n = 5, A = [5, 1, 2, 3, 0].$$
$$\text{Output: } A = [0, 1, 2, 3, 5].$$

1. Most fundamental problem in CS.
2. *A* could be an array of strings, real numbers, graphs or anything as long as we could compare them.

Introduction

Sort

Given an array A of n numbers $A[1], \ldots, A[n]$, sort them in order.

$$\text{Input: } n = 5, A = [5, 1, 2, 3, 0].$$
$$\text{Output: } A = [0, 1, 2, 3, 5].$$

1. Most fundamental problem in CS.
2. A could be an array of strings, real numbers, graphs or anything as long as we could compare them.
3. Discuss several sorting algorithms: Insertion-sort, ShellSort, Merge-sort and Quicksort.
4. Many interesting ideas: number theory, divide & conquer, randomized algorithms, ...

Insertion sort

The most basic sorting algorithm:

1. Easy to implement.
2. Relative slow.

Basic idea: After sorting the first j elements,

Insertion sort

$$A[1] \le A[2] \le A[3] \le \cdots \le A[i] \le A[i+1] \le \cdots \le A[j], \qquad \text{insert } A[j+1]?$$

insert $A[j+1]$ into them.

Insertion sort

The most basic sorting algorithm:

1. Easy to implement.
2. Relative slow.

Basic idea: After sorting the first j elements,

Insertion sort

$$A[1] \leq A[2] \leq A[3] \leq \cdots \leq A[i] \leq A[i+1] \leq \cdots \leq A[j], \qquad \text{insert } A[j+1]?$$

insert $A[j+1]$ into them.

Insertion sort

$$A[1] \leq A[2] \leq A[3] \leq \cdots \leq A[i] \leq A[i+1] \leq \cdots \leq A[j], \qquad \text{insert } A[j+1]?$$

Insertion sort

Plan: In the 1st round, sort $A[1]$ and $A[2]$.

1st round:	$A = [5, 1, 2, 3, 0]$
compare $A[1]$ and $A[2]$ $\Rightarrow$	$A = [1, 5, 2, 3, 0]$

Insertion sort

Plan: In the 1st round, sort $A[1]$ and $A[2]$.
In the 2nd round, insert $A[3]$ into $A[1]$ and $A[2]$.

1st round:	$A = [5, 1, 2, 3, 0]$
compare $A[1]$ and $A[2]$ $\Rightarrow$	$A = [1, 5, 2, 3, 0]$
2nd round:	Insert $A[3] = 2$
compare 2 and $A[2]$	

Insertion sort

Plan: In the 1st round, sort $A[1]$ and $A[2]$.
In the 2nd round, insert $A[3]$ into $A[1]$ and $A[2]$.

1st round:	$A = [5, 1, 2, 3, 0]$
compare $A[1]$ and $A[2]$ $\Rightarrow$	$A = [1, 5, 2, 3, 0]$
2nd round:	Insert $A[3] = 2$
compare 2 and $A[2]$ $\Rightarrow$	$A = [1, 2, 5, 3, 0]$
compare 2 and $A[1]$ $\Rightarrow$	$A = [1, 2, 5, 3, 0]$

Insertion sort

Plan: In the 1st round, sort $A[1]$ and $A[2]$.
In the 2nd round, insert $A[3]$ into $A[1]$ and $A[2]$.
In the 3rd round, insert $A[4]$ into $A[1], A[2], A[3]$;

1st round:	$A = [5, 1, 2, 3, 0]$
compare $A[1]$ and $A[2]$ $\Rightarrow$	$A = [1, 5, 2, 3, 0]$
2nd round:	Insert $A[3] = 2$
compare 2 and $A[2]$ $\Rightarrow$	$A = [1, 2, 5, 3, 0]$
compare 2 and $A[1]$ $\Rightarrow$	$A = [1, 2, 5, 3, 0]$
3rd round:	Insert $A[4] = 3$
compare 3 and $A[3]$ $\Rightarrow$	$A = [1, 2, 3, 5, 0]$
compare 3 and $A[2]$	

Insertion sort

Plan: In the 1st round, sort $A[1]$ and $A[2]$.
In the 2nd round, insert $A[3]$ into $A[1]$ and $A[2]$.
In the 3rd round, insert $A[4]$ into $A[1], A[2], A[3]$;

$$
\begin{aligned}
\text{1st round:} & \qquad A = [5, 1, 2, 3, 0] \\
\text{compare } A[1] \text{ and } A[2] \ \Rightarrow & \qquad A = [1, 5, 2, 3, 0] \\
\text{2nd round:} & \qquad \text{Insert } A[3] = 2 \\
\text{compare 2 and } A[2] \ \Rightarrow & \qquad A = [1, 2, 5, 3, 0] \\
\text{compare 2 and } A[1] \ \Rightarrow & \qquad A = [1, 2, 5, 3, 0] \\
\text{3rd round:} & \qquad \text{Insert } A[4] = 3 \\
\text{compare 3 and } A[3] \ \Rightarrow & \qquad A = [1, 2, 3, 5, 0] \\
\text{compare 3 and } A[2] \ \Rightarrow & \qquad A = [1, 2, 3, 5, 0] \\
\text{Stop this round:} & \qquad \text{No need to compare 3 and } A[1]! \\
\text{4th round:} & \qquad \text{Insert } A[5] = 0 \\
\cdots & \qquad A = [0, 1, 2, 3, 5]
\end{aligned}
$$

Insertion sort

Plan: In the 1st round, sort $A[1]$ and $A[2]$.
In the 2nd round, insert $A[3]$ into $A[1]$ and $A[2]$.
In the 3rd round, insert $A[4]$ into $A[1], A[2], A[3]$;

$$\begin{aligned}
\text{1st round:} & \quad & A = [5, 1, 2, 3, 0] \\
\text{compare } A[1] \text{ and } A[2] & \Rightarrow & A = [1, 5, 2, 3, 0] \\
\text{2nd round:} & & \text{Insert } A[3] = 2 \\
\text{compare 2 and } A[2] & \Rightarrow & A = [1, 2, 5, 3, 0] \\
\text{compare 2 and } A[1] & \Rightarrow & A = [1, 2, 5, 3, 0] \\
\text{3rd round:} & & \text{Insert } A[4] = 3 \\
\text{compare 3 and } A[3] & \Rightarrow & A = [1, 2, 3, 5, 0] \\
\text{compare 3 and } A[2] & \Rightarrow & A = [1, 2, 3, 5, 0] \\
\text{Stop this round:} & & \text{No need to compare 3 and } A[1]! \\
\text{4th round:} & & \text{Insert } A[5] = 0 \\
\cdots & & A = [0, 1, 2, 3, 5]
\end{aligned}$$

Insertion sort

Plan: In the 1st round, sort $A[1]$ and $A[2]$.
In the 2nd round, insert $A[3]$ into $A[1]$ and $A[2]$.
In the 3rd round, insert $A[4]$ into $A[1], A[2], A[3]$; and so on.

$$
\begin{aligned}
\text{1st round:} & \qquad A = [5, 1, 2, 3, 0] \\
\text{compare } A[1] \text{ and } A[2] \ \Rightarrow & \qquad A = [1, 5, 2, 3, 0] \\
\text{2nd round:} & \qquad \text{Insert } A[3] = 2 \\
\text{compare 2 and } A[2] \ \Rightarrow & \qquad A = [1, 2, 5, 3, 0] \\
\text{compare 2 and } A[1] \ \Rightarrow & \qquad A = [1, 2, 5, 3, 0] \\
\text{3rd round:} & \qquad \text{Insert } A[4] = 3 \\
\text{compare 3 and } A[3] \ \Rightarrow & \qquad A = [1, 2, 3, 5, 0] \\
\text{compare 3 and } A[2] \ \Rightarrow & \qquad A = [1, 2, 3, 5, 0] \\
\text{Stop this round:} & \qquad \text{No need to compare 3 and } A[1]! \\
\text{4th round:} & \qquad \text{Insert } A[5] = 0 \\
\cdots & \qquad A = [0, 1, 2, 3, 5]
\end{aligned}
$$

Pseudo-code

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

On Page 18 of [CLRS]

Pseudo-code

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

On Page 18 of [CLRS]

Two questions

(1) Correctness?

(2) Running time of sorting n numbers?

Correctness

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Prove that A is sorted after the loop?

Correctness

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Prove that A is sorted after the loop?

Induction on loop j

Hypothesis: After every iteration of j, $A[1], \ldots, A[j]$ are in sorted order.

Correctness

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j - 1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Prove that A is sorted after the loop?

Induction on loop j

Hypothesis: After every iteration of j, $A[1], \ldots, A[j]$ are in sorted order.

1. Base case: $j = 1$, $A[1]$ is sorted.

Correctness

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Prove that A is sorted after the loop?

Induction on loop j

Hypothesis: After every iteration of j, $A[1], \ldots, A[j]$ are in sorted order.

1. Base case: $j = 1$, $A[1]$ is sorted.
2. Induction step: Suppose $A[1], \ldots, A[k]$ are sorted. Consider $A[1], \ldots, A[k+1]$ after the iteration of $j = k + 1$.

Observation

After the while loop, $A[i] \leqslant key < A[i+1]$.

Running time

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

To sort n numbers, the worst running time is

Running time

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

To sort *n* numbers, the worst running time is

$$\sum_{j=2}^{n} \left(O(1) + \sum_{i} O(1) \right)$$

Running time

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

To sort n numbers, the worst running time is

$$
\begin{aligned}
\sum_{j=2}^{n}\left(O(1)+\sum_{i} O(1)\right) &\leqslant O(n)+\sum_{j=2}^{n}\sum_{i=j-1}^{0} O(1) \\
&= O(n)+\sum_{j=2}^{n} O(j) \\
&= O(n^2)
\end{aligned}
$$

Running time

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

To sort n numbers, the worst running time is

$$\begin{aligned}
\sum_{j=2}^{n}\left(O(1)+\sum_{i} O(1)\right) &\leqslant O(n)+\sum_{j=2}^{n}\sum_{i=j-1}^{0} O(1)\\
&= O(n)+\sum_{j=2}^{n} O(j)\\
&= O(n^2)
\end{aligned}$$

Question: How about average-case running time?

Discussion

While Insertion-sort is easy to implement, the running time is slow.

Bottleneck

For each new element $key = A[j]$, it enumerates i to find the position

$$A[i] \leqslant key < A[i+1].$$

Two ideas to improve it:

Discussion

While Insertion-sort is easy to implement, the running time is slow.

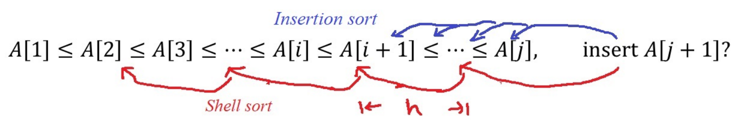
Bottleneck

For each new element $key = A[j]$, it enumerates i to find the position

$$A[i] \leqslant key < A[i+1].$$

Two ideas to improve it:

1. Any sorting algorithm that only swaps adjacent elements requires $\Omega(n^2)$ time — Move $A[j]$ further with each swap (called Shellsort).



Discussion

While Insertion-sort is easy to implement, the running time is slow.

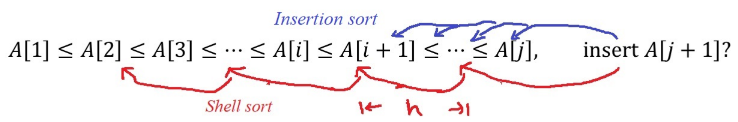
Bottleneck

For each new element $key = A[j]$, it enumerates i to find the position

$$A[i] \leqslant key < A[i+1].$$

Two ideas to improve it:

1. Any sorting algorithm that only swaps adjacent elements requires $\Omega(n^2)$ time — Move $A[j]$ further with each swap (called Shellsort).

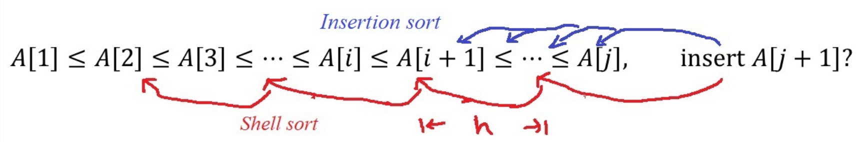


2. Find the position of $A[j]$ more efficiently (either in amortized time or by data structures).

Outline

Idea

Basic idea: Apply Insertion-sort multiple times with different increments h, like inserting $A[j]$ into $A[j-h], A[j-2h], A[j-3h], \ldots$



Idea

Basic idea: Apply Insertion-sort multiple times with different increments h, like inserting $A[j]$ into $A[j-h], A[j-2h], A[j-3h], \ldots$

Insertion sort

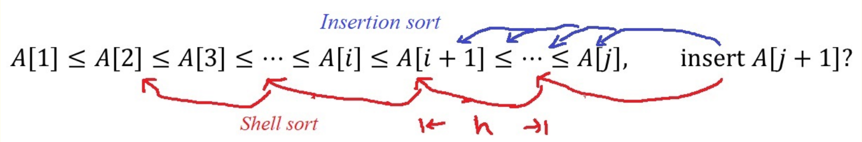
$$A[1] \leq A[2] \leq A[3] \leq \cdots \leq A[i] \leq A[i+1] \leq \cdots \leq A[j], \qquad \text{insert } A[j+1]?$$

Shell sort $\quad |\leftarrow \; h \; \rightarrow|$

Question: How to guarantee the correctness?

Idea

Basic idea: Apply Insertion-sort multiple times with different increments h, like inserting $A[j]$ into $A[j-h], A[j-2h], A[j-3h], \ldots$



Question: How to guarantee the correctness?
Solution: Set the last increment $h = 1$.

Description

```
procedure InsertionSort(h)
    for j = h + 1, ..., n do
        key = A[j]
        i = j − h
        while i > 0 and A[i] > key do
            A[i + h] = A[i]
            i = i − h
        A[i + h] = key
procedure Main(H)
    for each h ∈ H (from the largest to 1) do
        InsertionSort(h)
```

Description

procedure INSERTIONSORT(h)
  for $j = h + 1, \ldots, n$ **do**
    $key = A[j]$
    $i = j - h$
    while $i > 0$ and $A[i] > key$ **do**
      $A[i + h] = A[i]$
      $i = i - h$
    $A[i + h] = key$
procedure MAIN(H)
  for each $h \in H$ (from the largest to 1) **do**
    INSERTIONSORT(h)

Next

(1) Correctness?
(2) Running time?

Analysis

Since $1 \in H$, its correctness follows from the previous proof.

Question

How to choose H to minimize the running time?

Analysis

Since $1 \in H$, its correctness follows from the previous proof.

Question

How to choose H to minimize the running time?

Toy example: $H = \{\sqrt{n}, 1\}$. So time = time$\left(\textsc{InsertionSort}(h_1)\right)$ + time$\left(\textsc{InsertionSort}(h_2)\right)$

Analysis

Since $1 \in H$, its correctness follows from the previous proof.

Question

How to choose H to minimize the running time?

Toy example: $H = \{\sqrt{n}, 1\}$. So time = time$\Big($InsertionSort$(h_1)\Big)$ + time$\Big($InsertionSort$(h_2)\Big)$

1. For $h_2 = \sqrt{n}$, InsertionSort(h_2) takes at most n^2/h_2 times.

Analysis

Since $1 \in H$, its correctness follows from the previous proof.

Question

How to choose H to minimize the running time?

Toy example: $H = \{\sqrt{n}, 1\}$. So time = time$\left(\textsc{InsertionSort}(h_1)\right)$ + time$\left(\textsc{InsertionSort}(h_2)\right)$

1. For $h_2 = \sqrt{n}$, $\textsc{InsertionSort}(h_2)$ takes at most n^2/h_2 times.
2. For $h_1 = 1$, we expect each $A[j]$ will be moved within $O(h_2)$ steps.

Analysis

Since $1 \in H$, its correctness follows from the previous proof.

Question

How to choose H to minimize the running time?

Toy example: $H = \{\sqrt{n}, 1\}$. So time = time$\Big(\textsc{InsertionSort}(h_1)\Big)$ + time$\Big(\textsc{InsertionSort}(h_2)\Big)$

1. For $h_2 = \sqrt{n}$, $\textsc{InsertionSort}(h_2)$ takes at most n^2/h_2 times.
2. For $h_1 = 1$, we expect each $A[j]$ will be moved within $O(h_2)$ steps.

So the ideal running time is $O(n^2/h_2 + n \cdot h_2) = O(n^{1.5})$ — Question: Will this always happen?

Analysis

Since $1 \in H$, its correctness follows from the previous proof.

Question

How to choose H to minimize the running time?

Toy example: $H = \{\sqrt{n}, 1\}$. So time = time$\Big($InsertionSort$(h_1)\Big)$ + time$\Big($InsertionSort$(h_2)\Big)$

1. For $h_2 = \sqrt{n}$, InsertionSort(h_2) takes at most n^2/h_2 times.
2. For $h_1 = 1$, we expect each $A[j]$ will be moved within $O(h_2)$ steps.

So the ideal running time is $O(n^2/h_2 + n \cdot h_2) = O(n^{1.5})$
— Question: Will this always happen?

Next slide ☺ : Proving the running time is $o(n^2)$ rigorously is quite involved!

Formal Result

Theorem (1)

For $H = \{2^k - 1 | k = \log_2 n, \ldots, 1\}$, *the running time is* $O(n^{3/2})$.

Proof: InsertionSort(h) takes $\leqslant n^2/h$ time — Only need to consider InsertionSort($2^{\lceil \log_2 \sqrt{n} \rceil} - 1$), ..., InsertionSort(3), InsertionSort(1).

Formal Result

Theorem (1)

For $H = \{2^k - 1 | k = \log_2 n, \ldots, 1\}$*, the running time is* $O(n^{3/2})$*.*

Proof: InsertionSort(h) takes $\leqslant n^2/h$ time — Only need to consider InsertionSort($2^{\lceil \log_2 \sqrt{n} \rceil} - 1$), ..., InsertionSort(3), InsertionSort(1).

Lemma (1)

If we have applied InsertionSort*(*h_{t+1}*) and* InsertionSort*(*h_t*) to* A*, then the time of* InsertionSort*(*h_{t-1}*) is* $O(\frac{n \cdot h_{t+1} \cdot h_t}{h_{t-1}})$*.*

Proof of Lemma (1)

Lemma (1)

If we have applied InsertionSort*(h_{t+1}) and* InsertionSort*(h_t) to A, then the time of* InsertionSort*(h_{t-1}) is $O(\frac{n \cdot h_{t+1} \cdot h_t}{h_{t-1}})$.*

Proof of Lemma (1)

Lemma (1)

If we have applied INSERTIONSORT(h_{t+1}) *and* INSERTIONSORT(h_t) *to* A*, then the time of* INSERTIONSORT(h_{t-1}) *is* $O(\frac{n \cdot h_{t+1} \cdot h_t}{h_{t-1}})$.

1. OBS 1: If we have applied INSERTIONSORT(h), then A is always sorted with increments h in the future.

Proof of Lemma (1)

Lemma (1)

If we have applied InsertionSort*(h_{t+1}) and* InsertionSort*(h_t) to A, then the time of* InsertionSort*(h_{t-1}) is $O(\frac{n \cdot h_{t+1} \cdot h_t}{h_{t-1}})$.*

1. OBS 1: If we have applied InsertionSort(h), then A is always sorted with increments h in the future.
2. 2nd step: In InsertionSort(h_{t-1}), each element will be moved by at most $h_{t+1} \cdot h_t / h_{t-1}$ steps when h_{t+1} and h_t are co-prime.

Summary

1. Simple modifications improve the performance significantly — if one is not enough, try more!.
2. Deep math in short codes.

Summary

1. Simple modifications improve the performance significantly — if one is not enough, try more!.
2. Deep math in short codes.
3. The running time of ShellSort can be improved to $O(n \log^2 n)$ by choosing more complicated H — try it in Experiment 1
4. Advantage: (1) Extremely easy to implement;
   (2) Saving space — $O(1)$ extra bytes.
5. Disadvantage: Not the fastest in theory.

Discussion

Another bottleneck of INSERTIONSORT is that they did fully utilize the fact that $A[1], \ldots, A[j-1]$ have been sorted before inserting $A[j]$.

$$A[1] \le A[2] \le A[3] \le \cdots \le A[i] \le A[i+1] \le \cdots \le A[j], \qquad \text{insert } A[j+1]?$$

$$A[1] \le A[2] \le A[3] \le \cdots \le A[j+1] \le \cdots \le A[i] \le A[i+1] \le \cdots \le A[j]$$

Two ideas

Discussion

Another bottleneck of InsertionSort is that they did fully utilize the fact that $A[1], \ldots, A[j-1]$ have been sorted before inserting $A[j]$.

$$A[1] \le A[2] \le A[3] \le \cdots \le A[i] \le A[i+1] \le \cdots \le A[j], \qquad \text{insert } A[j+1]?$$

$$A[1] \le A[2] \le A[3] \le \cdots \le A[j+1] \le \cdots \le A[i] \le A[i+1] \le \cdots \le A[j]$$

Two ideas

1. Maintain a data structure to find the position and shift A quickly $\Rightarrow$ called heap

Discussion

Another bottleneck of INSERTIONSORT is that they did fully utilize the fact that $A[1], \ldots, A[j-1]$ have been sorted before inserting $A[j]$.

$$A[1] \le A[2] \le A[3] \le \cdots \le A[i] \le A[i+1] \le \cdots \le A[j], \qquad \text{insert } A[j+1]?$$

$$A[1] \le A[2] \le A[3] \le \cdots \le A[j+1] \le \cdots \le A[i] \le A[i+1] \le \cdots \le A[j]$$

Two ideas

1. Maintain a data structure to find the position and shift A quickly $\Rightarrow$ called heap
2. Reduce the insertion time by considering amortized time of multiple insertions.

Outline

Overview

Try 2nd idea

Reduce the insertion time by considering amortized time of multiple insertions.

Consider the insertion of $A[j], A[j+1], \ldots, A[k]$.
Example: $(A_1, \ldots, A[j-1]) = (1, 4, 7)$ and $(A[j], \ldots, A[k]) = (2, 3, 8)$.

Overview

Try 2nd idea

Reduce the insertion time by considering amortized time of multiple insertions.

Consider the insertion of $A[j], A[j+1], \ldots, A[k]$.
Example: $(A_1, \ldots, A[j-1]) = (1, 4, 7)$ and $(A[j], \ldots, A[k]) = (2, 3, 8)$.

Observation

After inserting $A[j]$ between A_1 and A_2, $A[j+1] = 3$ must be inserted behind $A[j] = 2$.

Overview

Try 2nd idea

Reduce the insertion time by considering amortized time of multiple insertions.

Consider the insertion of $A[j], A[j+1], \ldots, A[k]$.
Example: $(A_1, \ldots, A[j-1]) = (1, 4, 7)$ and $(A[j], \ldots, A[k]) = (2, 3, 8)$.

Observation

After inserting $A[j]$ between A_1 and A_2, $A[j+1] = 3$ must be inserted behind $A[j] = 2$.

We can implement it either as multiple insertions or merge.

Mergesort

If $A[p], \ldots, A[j-1]$ and $A[j], \ldots, A[k]$ are sorted, we can merge them in linear time $O(k-p)$ — the amortized time of inserting $A[j], \ldots, A[k]$ is $O(1)$ if $k - j \approx j - p$.

---

procedure MERGE(p, j, k)

  $n_1 = j - p, n_2 = k - j + 1$ — lengths of two sequences

  Let $L[1, \ldots, n_1 + 1]$ and $R[1, \ldots, n_2 + 1]$ be new arrays to avoid shift

  $L[i] = A[p + i - 1]$ for $i \leqslant n_1$ and $L[n_1 + 1] = +\infty$

  $R[i] = A[j + i - 1]$ for $i \leqslant n_2$ and $R[n_2 + 1] = +\infty$

  $i_\ell = 1, i_r = 1$ — two pointers on L and R separately

  for $i = p, \ldots, k$ **do**

    if $L[i_\ell] < R[i_r]$ **then**

      $A[i] = L[i_\ell]$

      $i_\ell = i_\ell + 1$

    else

      $A[i] = R[i_r]$

      $i_r = i_r + 1$

Mergesort

If $A[p], \ldots, A[j-1]$ and $A[j], \ldots, A[k]$ are sorted, we can merge them in linear time $O(k-p)$ — the amortized time of inserting $A[j], \ldots, A[k]$ is $O(1)$ if $k-j \approx j-p$.

---

procedure MERGE(p, j, k)

  $n_1 = j - p, n_2 = k - j + 1$ — lengths of two sequences

  Let $L[1, \ldots, n_1 + 1]$ and $R[1, \ldots, n_2 + 1]$ be new arrays to avoid shift

  $L[i] = A[p + i - 1]$ for $i \leqslant n_1$ and $L[n_1 + 1] = +\infty$

  $R[i] = A[j + i - 1]$ for $i \leqslant n_2$ and $R[n_2 + 1] = +\infty$

  $i_\ell = 1, i_r = 1$ — two pointers on L and R separately

  for $i = p, \ldots, k$ **do**

    if $L[i_\ell] < R[i_r]$ **then**

      $A[i] = L[i_\ell]$

      $i_\ell = i_\ell + 1$

    else

      $A[i] = R[i_r]$

      $i_r = i_r + 1$

Question: How to satisfy the above assumptions?

Mergesort

Main idea: Recursively sort $A[p], \ldots, A[j-1]$ and $A[j], \ldots, A[k]$ instead of applying INSERTIONSORT.

---

procedure SORT(p, r)
  if $p < r$ **then**
    $q = [(p + r)/2]$
    SORT(p, q)
    SORT($q + 1, r$)
    MERGE($p, q + 1, r$)

Mergesort

Main idea: Recursively sort $A[p], \ldots, A[j-1]$ and $A[j], \ldots, A[k]$ instead of applying INSERTIONSORT.

```
procedure SORT(p, r)
    if p < r then
        q = [(p + r)/2]
        SORT(p, q)
        SORT(q + 1, r)
        MERGE(p, q + 1, r)
```

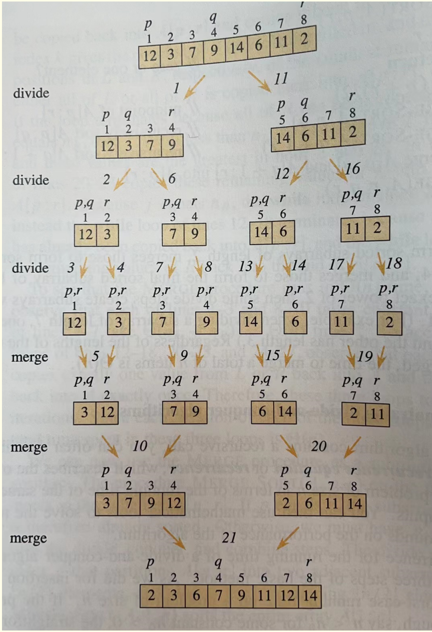
Analysis

Correctness: The correctness of SORT follows from the correctness of MERGE. One can apply induction (on loop i) to prove the later one.

Running time: How much does it improve upon INSERTIONSORT of $O(n^2)$?

Running Time

1. 1st Fact: The running time of MERGE(p, j, k) is $O(k - p)$.
2. To sort n elements, it recursively splits the sequence into $\log_2 n$ levels.
3. The running time of all MERGE in one level is $O(n)$
   $\Rightarrow$ the total running time is $O(n \log_2 n)$.
4. This technique is called Divide & Conquer.



Outline

1. Introduction
2. Shellsort
3. Mergesort
4. Quicksort

Introduction: Quicksort

While the running time of MergeSort is $O(n \log n)$, the constant is quite big — bottleneck: creating two arrays L and R.

Introduction: Quicksort

While the running time of MERGESORT is $O(n \log n)$, the constant is quite big — bottleneck: creating two arrays L and R.

New idea

Similar to MERGESORT, split the sequence into two. However, instead of dividing them equally, we pick a pivot to separate them.

Introduction: Quicksort

While the running time of MERGESORT is $O(n \log n)$, the constant is quite big — bottleneck: creating two arrays L and R.

New idea

Similar to MERGESORT, split the sequence into two. However, instead of dividing them equally, we pick a pivot to separate them.

Example: $A = (3, 0, 7, 6, 5, 4, 1, 8, 2)$.

Choose $x = 5$ as the pivot to separate them

$$(3, 0, 4, 1, 2), 5, (6, 7, 8).$$

Introduction: Quicksort

While the running time of MERGESORT is $O(n \log n)$, the constant is quite big — bottleneck: creating two arrays L and R.

New idea

Similar to MERGESORT, split the sequence into two. However, instead of dividing them equally, we pick a pivot to separate them.

Example: $A = (3, 0, 7, 6, 5, 4, 1, 8, 2)$.

Choose $x = 5$ as the pivot to separate them

$$(3, 0, 4, 1, 2), 5, (6, 7, 8).$$

Question: While we could apply the same subroutine to sort the sub-sequences, how to choose x?

Randomized Algorithms

If it chooses x deterministically, $\exists A$ s.t. its running time is n^2.

Randomized Algorithms

If it chooses x deterministically, $\exists A$ s.t. its running time is n^2.

Example

If it sets $x = A[3]$ as 1st pivot, adversary chooses $A[3]$ to be the smallest (largest) element in A such that one sub-sequence is empty.

How to avoid adversarial inputs ?

Randomized Algorithms

If it chooses x deterministically, $\exists A$ s.t. its running time is n^2.

Example

If it sets $x = A[3]$ as 1st pivot, adversary chooses $A[3]$ to be the smallest (largest) element in A such that one sub-sequence is empty.

How to avoid adversarial inputs ?

Solution

Pick a random element in A as the pivot.

Description

procedure PARTITION(p, r)
  Randomly pick $k \sim [p, \ldots, r]$
  $x = A[k]$ and $i = p - 1$
  Exchange $A[k]$ and $A[r]$
  for $j = p, \ldots, r - 1$ **do**
    if $A[j] \leqslant x$ **then**
      $i = i + 1$
      Exchange $A[i]$ and $A[j]$
  Exchange $A[i + 1]$ and $A[r]$
  Return $i + 1$
procedure QUICKSORT(p, r)
  if $p < r$ **then**
    q=PARTITION(p, r)
    QUICKSORT($p, q - 1$)
    QUICKSORT($q + 1, r$)

Analysis

Questions

(1) How procedure PARTITION separates elements into two subsequences $\leqslant x$ and $\geqslant x$? (2) Formal proof?

Analysis

Questions

(1) How procedure PARTITION separates elements into two subsequences $\leqslant x$ and $\geqslant x$? (2) Formal proof?

- OBS 1: In the j-loop, $A[p], \ldots, A[i]$ are always $\leqslant x$.

Analysis

Questions

(1) How procedure PARTITION separates elements into two subsequences $\leqslant x$ and $\geqslant x$? (2) Formal proof?

- OBS 1: In the j-loop, $A[p], \ldots, A[i]$ are always $\leqslant x$.
- OBS 2: In the j-loop, $A[i+1], \ldots, A[j]$ are always $\geqslant x$.

Analysis

Questions

(1) How procedure PARTITION separates elements into two subsequences $\leqslant x$ and $\geqslant x$? (2) Formal proof?

- OBS 1: In the j-loop, $A[p], \ldots, A[i]$ are always $\leqslant x$.
- OBS 2: In the j-loop, $A[i+1], \ldots, A[j]$ are always $\geqslant x$.
- Formal proof: Apply induction on j with the above 2 hypotheses.

Running Time

We will bound the expected running time (over random pivots).

```
procedure PARTITION(p, r)
    Randomly pick k ~ [p, ..., r]
    x = A[k] and i = p − 1
    Exchange A[k] and A[r]
    for j = p, ..., r − 1 do
        if A[j] ⩽ x then
            i = i + 1
            Exchange A[i] and A[j]
    Exchange A[i + 1] and A[r]
    Return i + 1
procedure QUICKSORT(p, r)
    if p < r then
        q=PARTITION(p, r)
        QUICKSORT(p, q − 1)
        QUICKSORT(q + 1, r)
```

Expected Running Time

1. Running time $T(n)$ is O(# comparisons)+$O(n)$: In PARTITION, j-loop compares each element in $[p, r-1]$ with pivot x.
2. Each pair get compared at most once: future calls won't compare the pivot $x = A[k]$ any more.

Expected Running Time

1. Running time $T(n)$ is O(# comparisons)+$O(n)$: In PARTITION, j-loop compares each element in $[p, r-1]$ with pivot x.
2. Each pair get compared at most once: future calls won't compare the pivot $x = A[k]$ any more.

Fact

Let $z_1 < z_2 < \ldots < z_n$ be sorted order of A and R.V. $X_{i,j} \in \{0, 1\}$ indicates whether QUICKSORT compares (z_i, z_j) or not.

From (1) and (2), $\mathbb{E}[T(n)] = O\left(\mathbb{E}[\text{\# comparisons}]\right) = O\left(\mathbb{E}\left[\sum_{i<j} X_{i,j}\right]\right)$.

Expected Running Time

1. Running time $T(n)$ is O(# comparisons)+$O(n)$: In PARTITION, j-loop compares each element in $[p, r-1]$ with pivot x.
2. Each pair get compared at most once: future calls won't compare the pivot $x = A[k]$ any more.

Fact

Let $z_1 < z_2 < \ldots < z_n$ be sorted order of A and R.V. $X_{i,j} \in \{0, 1\}$ indicates whether QUICKSORT compares (z_i, z_j) or not.

From (1) and (2), $\mathbb{E}[T(n)] = O\left(\mathbb{E}[\#\text{ comparisons}]\right) = O\left(\mathbb{E}\left[\sum_{i<j} X_{i,j}\right]\right)$.

$$\mathbb{E}\left[\sum_{i<j} X_{i,j}\right] = \sum_{i<j} \mathbb{E}\left[X_{i,j}\right] \qquad \text{(linearity of expectation)}$$

$$= \sum_{i<j} \Pr\left[z_i \text{ and } z_j \text{ get compared}\right].$$

Analysis (II)

To calculate $\Pr\left[z_i \text{ and } z_j \text{ get compared}\right]$, consider all random pivots chosen in QUICKSORT.

Question

What happens on those random pivots s.t. z_i and z_j get compared?

Analysis (II)

To calculate $\Pr\left[z_i \text{ and } z_j \text{ get compared}\right]$, consider all random pivots chosen in QUICKSORT.

Question

What happens on those random pivots s.t. z_i and z_j get compared?

Answer: Iff the 1st pivot in $z_i, \ldots, z_j$ is z_i or z_j.

Analysis (II)

To calculate $\Pr\left[z_i \text{ and } z_j \text{ get compared}\right]$, consider all random pivots chosen in QUICKSORT.

Question

What happens on those random pivots s.t. z_i and z_j get compared?

Answer: Iff the 1st pivot in $z_i, \ldots, z_j$ is z_i or z_j.

1. $\exists$ one pivot in $z_i, z_{i+1}, \ldots, z_j$. Otherwise we can not separate z_i and z_j.
2. Consider 1st pivot in $z_i, z_{i+1}, \ldots, z_j$: If it is not z_i or z_j, QUICKSORT separates z_i and z_j into two branches.

Analysis (II)

To calculate $\Pr\left[z_i \text{ and } z_j \text{ get compared}\right]$, consider all random pivots chosen in QUICKSORT.

Question

What happens on those random pivots s.t. z_i and z_j get compared?

Answer: Iff the 1st pivot in $z_i, \ldots, z_j$ is z_i or z_j.

1. $\exists$ one pivot in $z_i, z_{i+1}, \ldots, z_j$. Otherwise we can not separate z_i and z_j.
2. Consider 1st pivot in $z_i, z_{i+1}, \ldots, z_j$: If it is not z_i or z_j, QUICKSORT separates z_i and z_j into two branches.
3. Otherwise the 1st pivot is z_i or z_j — they got compared.

Analysis (II)

To calculate $\Pr\left[z_i \text{ and } z_j \text{ get compared}\right]$, consider all random pivots chosen in QUICKSORT.

Question

What happens on those random pivots s.t. z_i and z_j get compared?

Answer: Iff the 1st pivot in $z_i, \ldots, z_j$ is z_i or z_j.

1. $\exists$ one pivot in $z_i, z_{i+1}, \ldots, z_j$. Otherwise we can not separate z_i and z_j.
2. Consider 1st pivot in $z_i, z_{i+1}, \ldots, z_j$: If it is not z_i or z_j, QUICKSORT separates z_i and z_j into two branches.
3. Otherwise the 1st pivot is z_i or z_j — they got compared.

So the probability is $\frac{2}{j-i+1}$!

Wrap up

Running time $T(n)$ has an expectation in the order of

$$\begin{aligned}\mathbb{E}[\text{\# of comparisons}] &= \sum_{i<j} \Pr[\text{it compares } z_i \text{ and } z_j] \\ &= \sum_{i<j} \frac{2}{j-i+1} = O(n \ln n).\end{aligned}$$

Wrap up

Running time $T(n)$ has an expectation in the order of

$$\begin{aligned}\mathbb{E}[\text{\# of comparisons}] &= \sum_{i<j} \Pr[\text{it compares } z_i \text{ and } z_j] \\ &= \sum_{i<j} \frac{2}{j-i+1} = O(n \ln n).\end{aligned}$$

While this implies $\mathbb{E}[T(n)] \leqslant Cn \log n$ for $C = O(1)$,

1. Markov's inequality implies $\Pr\left[T(n) \geqslant 100Cn \log n\right] \leqslant 0.01$

Wrap up

Running time $T(n)$ has an expectation in the order of

$$\mathbb{E}[\text{\# of comparisons}] = \sum_{i<j} \Pr[\text{it compares } z_i \text{ and } z_j]$$
$$= \sum_{i<j} \frac{2}{j-i+1} = O(n \ln n).$$

While this implies $\mathbb{E}[T(n)] \leqslant Cn \log n$ for $C = O(1)$,

1. Markov's inequality implies $\Pr\left[T(n) \geqslant 100Cn \log n\right] \leqslant 0.01$
2. $\mathrm{Var}(T(n)) = \Theta(n^2) \Rightarrow \Pr\left[T(n) \geqslant 2Cn \log n\right] = O\left(\frac{1}{\log^2 n}\right)$
3. Sharp concentration:

$$\Pr\left[T(n) \geqslant (C+\delta) n \log n\right] = n^{-\Theta(\delta \log \log n)}$$

See lecture note from Berkeley: `https://people.eecs.berkeley.edu/~sinclair/cs271/n20.pdf`

Summary

4 sorting algorithms:

1. InsertionSort is simple and easy to implement, but $O(n^2)$ time.
2. ShellSort improves InsertionSort to $o(n^2)$ — tricky to analyze.

Summary

4 sorting algorithms:

1. InsertionSort is simple and easy to implement, but $O(n^2)$ time.
2. ShellSort improves InsertionSort to $o(n^2)$ — tricky to analyze.
3. MergeSort improves the time to $O(n \log n)$.
4. QuickSort has a better constant factor and more benefits (a) save space by sorting in order (b) extensions like FindMedian.

Questions?