Lec01 算法分析

the analysis of algorithm is the theoretical study of computer-programme performance and resource usage

"how to make things fast" predominantly!

what's more important than performance?

maintainability、robustness、features (functionality)、modularity、security、scalability、user-friendliness、…

why still study algorithms and performance?

① performance measures the line between the feasible and the infeasible (real-time requirements)

② algorithms give you a language for talking about program behavior (pervasive)

③ ton of fun

"有一个很好的比喻来形容性能、以及为何性能处于最底层，它扮演的角色就如同经济中的货币一般" (currency)

$\boxed{\$100}$ 途径 VS food, water, shelter "more important" "目的"

弹幕：实现其他功能需要牺牲性能

sometimes people are willing to pay a factor of three in performance, in order to trade for something that is worth it

in a word, you can use performance to pay for other things that you want, that's why (in some sense) performance is in the bottom of the heap

Problem: sorting

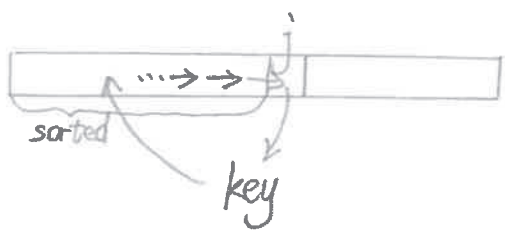
input: sequence $\langle a_1, a_2, \cdots, a_n \rangle$ of numbers

output: permutation $\langle a_1', a_2', \cdots, a_n' \rangle$ such that $a_1' \le a_2' \le \cdots \le a_n'$ (monotonically increasing)

Insertion Sort $(A; n)$ // sorts $A[1 \ldots n]$

```
for j ← 2 to n
    do key ← A[j]
       i ← j-1
       while i > 0 and A[i] > key
           do A[i+1] ← A[i]
              i ← i-1
       A[i+1] ← key
```

"pseudocode"



"一步步地把前面的值抄到下一位上，直到找到此键的合适位置"

Example: 8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

2 3 4 6 8 9 done!

running time:

- depends on input (eg. sorted already, reverse sorted is the worst case)
- depends on input size (6 elements vs 6×10^9 elements)
  - parameterize in input size
- want upper bounds (a guarantee to the user)

kinds of analysis

- worst case (usually)

  define $T(n)$ to be the maximum time on any input size n

- average case (sometimes)

  define $T(n)$ to be the expected time over all inputs of size n

  "mathematical expectation"

  need assumption of the statistical distribution of inputs

- best case (bogus, no good)

  cheat: just check for some particular input, ignoring the vast majority

what is the worst case time for insertion sort?

- depends on computer
  - relative speed (on same machine)
  - absolute speed (on different machines)

  Big Idea: asymptotic analysis

  1. to ignore machine-dependent constants
  2. to look at the growth of the running time, $T(n)$ as $n \to \infty$

- the input is reverse sorted

we assume every elemental operation is going to take some constant amount of time

$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$ (arithmetic series)

insertion sort is moderately fast for small n,
but it is not at all for large n.

Merge Sort

$T(n)$	Merge Sort $A[1 \ldots n]$
abuse $\to \Theta(1)$	1. if $n=1$, done
sloppy $\to 2T(n/2)$	2. recursively sort $A[1 \ldots \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \ldots n]$
$\Theta(n)$	3. "merge" 2 sorted lists

key subroutine is "merge"

20 13
13 11
7 9
→ 2 1 ← "smaller"

⇓

20 13
13 11
7 9 ←
"smaller" → 2 ~~1~~

⇓
⋮
⇓

20 13
→ 13 11
~~7~~ 9 ← "smaller"
~~2~~ ~~1~~

⇓
⋮
⇓
done

Time $= \Theta(n)$ on n total elements

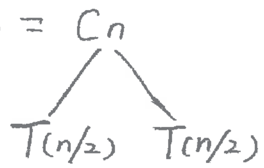
$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T(n/2) + \Theta(n), & \text{if } n > 1 \end{cases}$$

usually omit

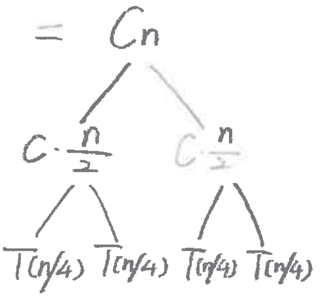
recursion tree technique

$T(n) = 2T(n/2) + Cn$, $C > 0$

$T(n) =$

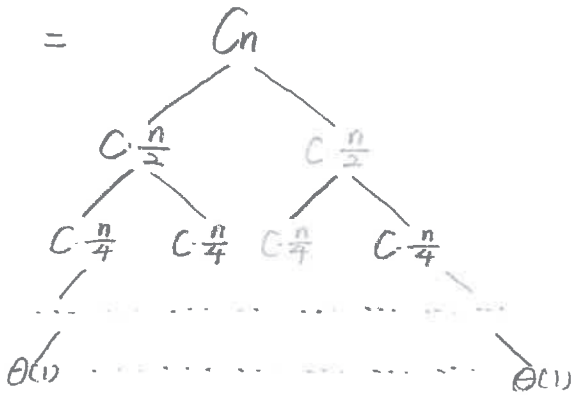


$=$



$= \cdots$

$=$



height $= \log_2 n$

#leaves $= n$

total $= Cn \cdot \log_2 n + \Theta(n)$
$= \Theta(n \log_2 n)$
$< \Theta(n^2)$

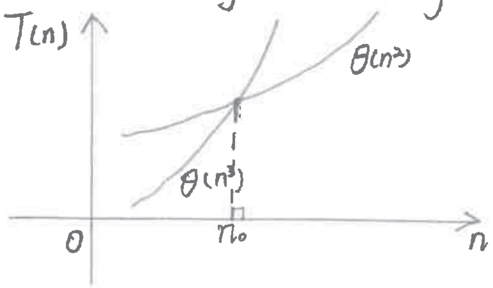
Asymptotic Notation

Θ-notation : drop low-order items and ignore leading constants

$$3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$$

"throughout the course, you are going to be responsible both for mathematical rigor as if it were a math course, and engineering commonsense because it's an engineering course"

as $n \to \infty$, $\Theta(n^2)$ algorithms always beat $\Theta(n^3)$ algorithms ($\exists n$, even on slow machine)
(affect leading constants)



sometimes it could be that n_0 is so large that computers aren't able to run the problem, that's why we are interested in some of the slower algorithms (they may still be faster on reasonable sizes of inputs, even though they may be asymptotically slower)

"if you want to be a good programmer,
you just program every day for 2 years, you will be an excellent programmer.
if you want to be a world-class programmer,
you can program every day for 10 years.
or you can program every day for 2 years and then take an algorithm class"