Lec 01 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)

$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

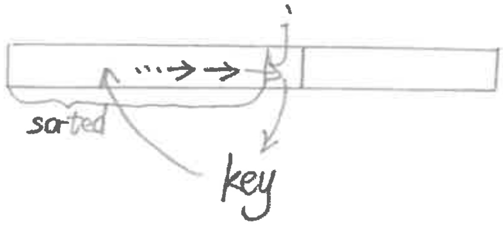
input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers

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

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

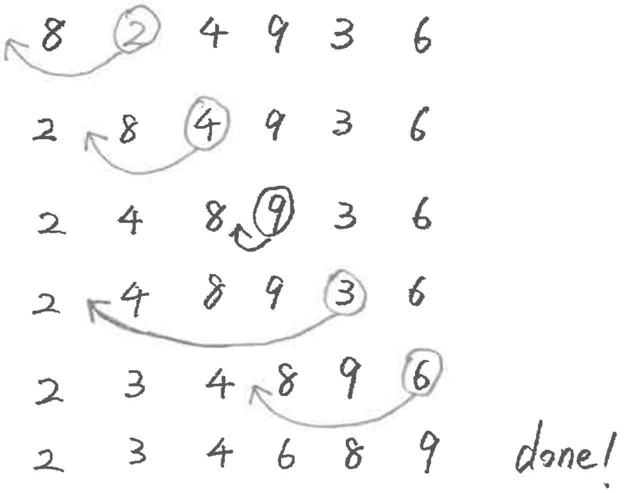
```
for j ← 2 to n
    do key ← A[j]
       i ← j-1
       while i > 0 and A[i] > key
           do A[i+1] ← A[i]
              i ← i-1
       A[i+1] ← key
```

"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

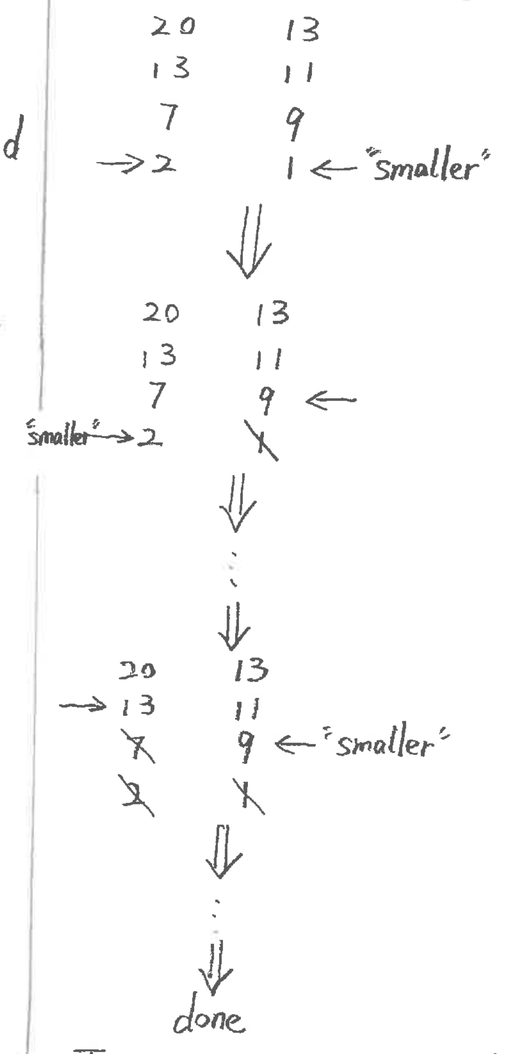
$$T(n) = \sum_{j=2}^{n} \theta(j) = \theta(n^2) \quad \text{(arithmetic series)}$$

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"



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

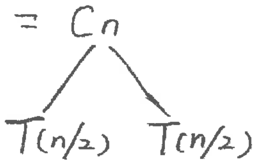
usually omit

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

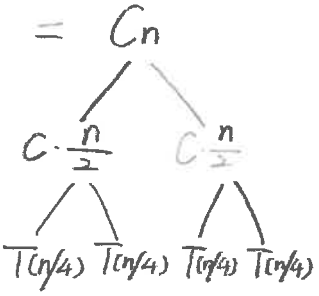
recursion tree technique

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

$T(n) =$

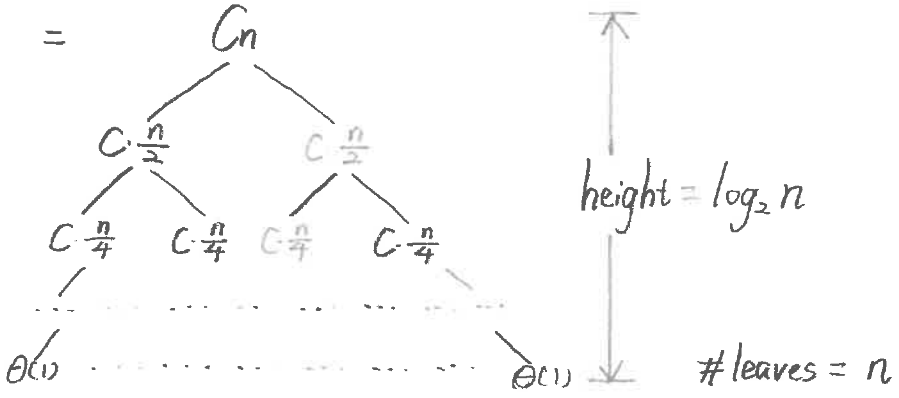


$=$



$= \cdots$

$=$



$$\begin{aligned} \text{total} &= Cn \cdot \log_2 n + \theta(n) \\ &= \theta(n \log_2 n) \\ &< \theta(n^2) \end{aligned}$$

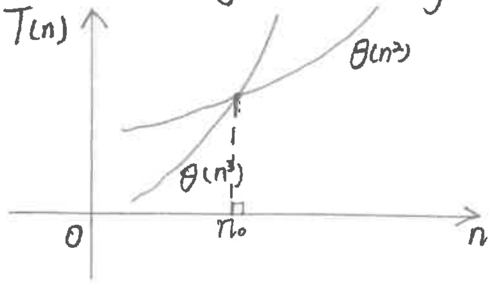
Asymptotic Notation

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 $\underbrace{\text{slow machine}}_{\text{(affect leading constants)}}$)



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"

Lec 02 渐近符号、递归及解法

Asymptotic Notation

- O notation

$f(n) = O(g(n))$ means there are some suitable constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$

Example: $2n^2 = O(n^3)$ (or $2n^2 \in O(n^3)$) ☆

the equation is not symmetric here!

another way to think about what it really means is that $f(n)$ is in the set of functions that are like $g(n)$, i.e.

$$O(g(n)) = \{ f(n) : \exists c, n_0 > 0,\ 0 \le f(n) \le c \cdot g(n), \text{for all } n \ge n_0 \}$$

- Macro convention
(宏)

a set in a formula represents an anonymous function in that set

Example: $f(n) = \underbrace{n^3}_{\text{`basically'}} + \underbrace{O(n^2)}_{\text{`error bound'}}$

means there is a function $h(n)$ which is in $O(n^2)$, such that $f(n) = n^3 + h(n)$

Example: $n^2 + O(n) = O(n^2)$ (or $n^2 + O(n) \subset O(n^2)$)

the equation is not symmetric!

means for any $f(n) \in O(n)$, $\underbrace{\text{there is}}_{`\exists'}$ an $h(n) \in O(n^2)$, such that $n^2 + f(n) = h(n)$

- Ω notation (lower bounds)

$$\Omega(g(n)) = \{ f(n) : \exists c, n_0 > 0,\ 0 \le c\, g(n) \le f(n), \text{for all } n \ge n_0 \}$$

Example: $\sqrt{n} = \Omega(\lg n)$

Analogies: O is '$\le$', Ω is '$\ge$', Θ is '$=$'

- $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

Example: $n^2 + O(n) = \Theta(n^2)$

- o & w notations

o is '$<$' (strictly), w is '$>$'.

def: for every constant c, there exists a constant n_0, ...

$\forall c > 0, \exists n_0 > 0, \ldots$

Example: $2n^2 = o(n^3)$

$\frac{1}{2}n^2 = \Theta(n^2) \neq o(n^2)$

Solving Recurrences (3 main methods)

- substitution method

1. guess the form of the solution
2. verify whether the recurrence satisfies this bound by induction (归纳法)
3. solve for constants

$$\underbrace{O(1) + O(1) + \cdots + O(1)}_{\text{无穷个}} \underset{\triangle}{\neq} O(1)$$

Example:

$T(n) = 4T(\frac{n}{2}) + n$, $T(1) = \Theta(1)$

Guess $T(n) = O(n^3)$

induction procedure

Assume $T(k) \leq c \cdot k^3$. for $k \leq n$

$T(n) = 4T(\frac{n}{2}) + n \leq 4 \cdot c \cdot (\frac{n}{2})^3 + n = \frac{1}{2} \cdot c \cdot n^3 + n$

$= \underbrace{c \cdot n^3}_{\text{desired}} - \underbrace{(\frac{1}{2} c \cdot n^3 - n)}_{\text{residual}}$

$\leq cn^3$. if residual part is non-negative e.g. $c \geq 1, n \geq 1$

$= O(n^3)$

and Base case:

$T(1) = \Theta(1) \leq c$, if c is chosen sufficiently large

tight bound

Try $T(n) = O(n^2)$

Assume $T(k) \leq c \cdot k^2$ for $k \leq n$

$T(n) = 4T(\frac{n}{2}) + n$

$\leq 4 \cdot c \cdot (\frac{n}{2})^2 + n$

$= c \cdot n^2 + n$ $\underbrace{= O(n^2)}_{\text{true, but useless}}$

$= c \cdot n^2 - (-n)$

$\not\leq c \cdot n^2$

"strengthen the induction hypothesis"

Assume $T(k) \leq c_1 \cdot k^2 - c_2 k$

$T(n) = 4T(\frac{n}{2}) + n$

$= 4[c_1 \cdot (\frac{n}{2})^2 - c_2 \cdot \frac{n}{2}] + n$

$= c_1 \cdot n^2 + (1 - 2c_2) \cdot n$

$= \underbrace{c_1 n^2 - c_2 \cdot n}_{\text{desired}} - \underbrace{(-1 + c_2)\, n}_{\text{residual}}$, want non-negative

$\leq c_1 n^2 - c_2 n$, if $c_2 \geq 1$

and Base case,

$$\begin{cases} T(1) \leqslant C_1 \cdot 1^2 - C_2 \cdot 1 \\ T(1) = \Theta(1) \end{cases}$$ ⟹ so we need to choose C_1 to be sufficiently larger than C_2

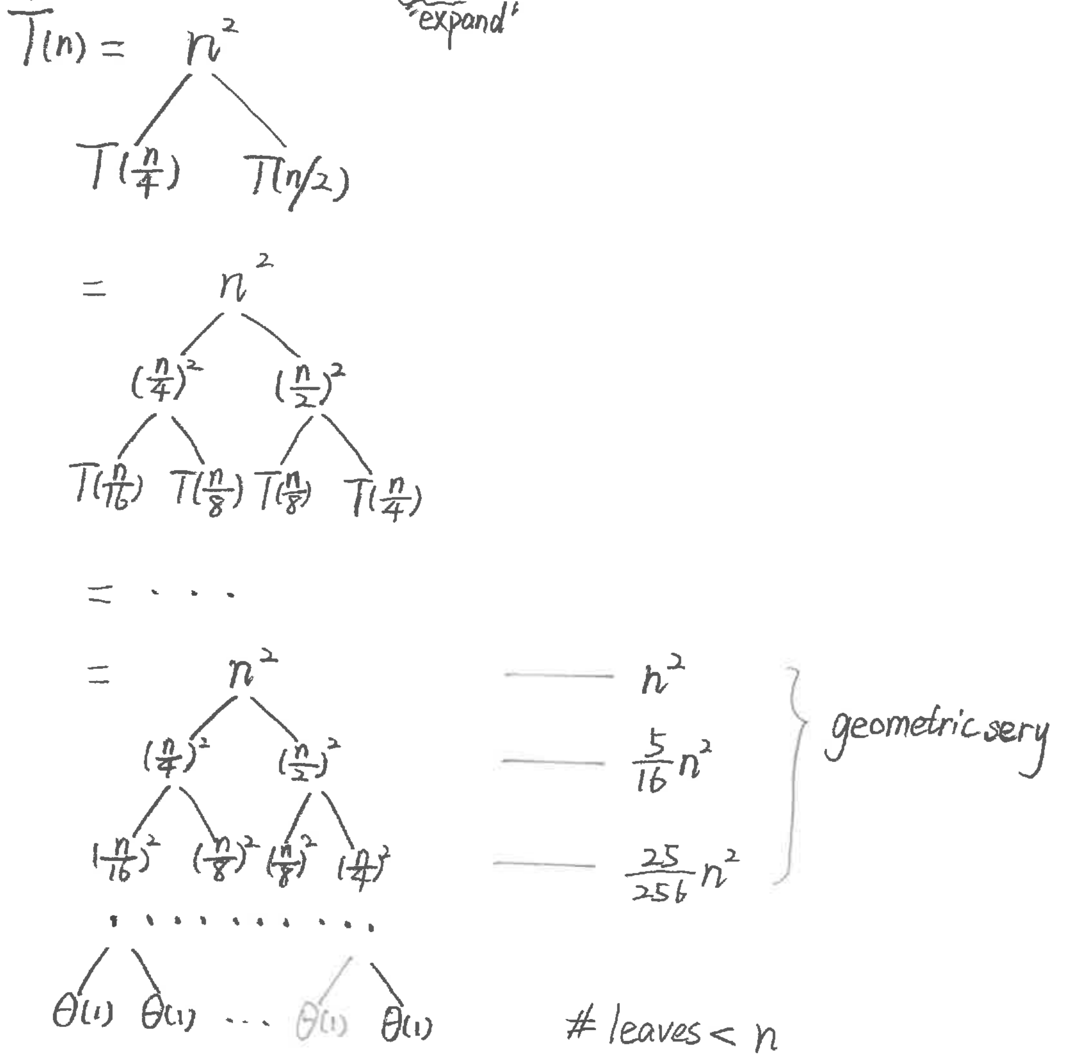
- recursion-tree method (not that rigorous)

"technically, what you should do is to find out what the answer is with recursion-tree method, then prove that it is actually right with substitution method"

Example:

$$T(n) = T(\tfrac{n}{4}) + T(\tfrac{n}{2}) + n^2$$

draw a picture (用树的形式展开递归)
'expand'



$T(n) = n^2$
$T(\frac{n}{4})$ $T(n/2)$
$= n^2$
$(\frac{n}{4})^2$ $(\frac{n}{2})^2$
$T(\frac{n}{16})$ $T(\frac{n}{8})$ $T(\frac{n}{8})$ $T(\frac{n}{4})$
$= \cdots$
$= n^2$
$(\frac{n}{4})^2$ $(\frac{n}{2})^2$
$(\frac{n}{16})^2$ $(\frac{n}{8})^2$ $(\frac{n}{8})^2$ $(\frac{n}{4})^2$
$\Theta(1)$ $\Theta(1)$ … $\Theta(1)$ $\Theta(1)$
n^2
$\frac{5}{16}n^2$
$\frac{25}{256}n^2$
geometric sery
# leaves < n

total (level by level)

$$\leq \left(1 + \frac{5}{16} + (\frac{5}{16})^2 + \cdots + (\frac{5}{16})^k + \cdots\right) \cdot n^2$$

it doesn't go on infinitely, but let's assume it goes on infinitely, that would be an upper bound that goes on forever

$$= \lim_{n\to\infty} \frac{1 \times [1 - (\frac{5}{16})^n]}{1 - \frac{5}{16}} \cdot n^2$$

$$= \frac{16}{11} n^2$$

$$< 2n^2 = O(n^2)$$

- master method

it is pretty restrictive, it only applies to a particular family of recurrences of the form

$$T(n) = \underbrace{a}_{\#\text{subproblems} = a}T(\underbrace{n/b}_{\text{every sub-problems you recurse on should be of the same size } \frac{n}{b}}) + \overbrace{f(n)}^{f(n)\text{是非递归的代价}}$$

where $a \geq 1$, $\underbrace{b > 1}_{\text{to make sure the subproblems are getting smaller}}$, $f(n)$ should be $\underbrace{\text{asymptotically positive}}_{\text{for large enough } n,\ (\exists n_0)\ f(n) \text{ is positive}}$.

to compare $f(n)$ with $n^{\log_b a}$ asymptotically

△ case 1 $f(n) = O(n^{\log_b a - \varepsilon})$, for some $\varepsilon > 0$ $(\exists \varepsilon > 0)$

$\Rightarrow T(n) = \Theta(n^{\log_b a})$

△ case 2 $f(n) = \Theta(n^{\log_b a} \cdot \log_2^k n)$

$\Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log_2^{k+1} n)$

△ case 3 $f(n) = \Omega(n^{\log_b a + \varepsilon})$, for some $\varepsilon > 0$ $(\exists \varepsilon > 0)$

& $af(n/b) \leq (1-\varepsilon') \cdot f(n)$, for some $\varepsilon' > 0$ $(\exists \varepsilon' > 0)$

$\Rightarrow T(n) = \Theta(f(n))$

Example:

$T(n) = 4T(\frac{n}{2}) + n$ $(a=4, b=2, f(n)=n)$

$n^{\log_b a} = n^2 > n = f(n)$, in case 1

so $T(n) = \Theta(n^2)$

Example:

$T(n) = 4T(\frac{n}{2}) + n^2$

in case 2, so $T(n) = \Theta(n^2 \log_2 n)$

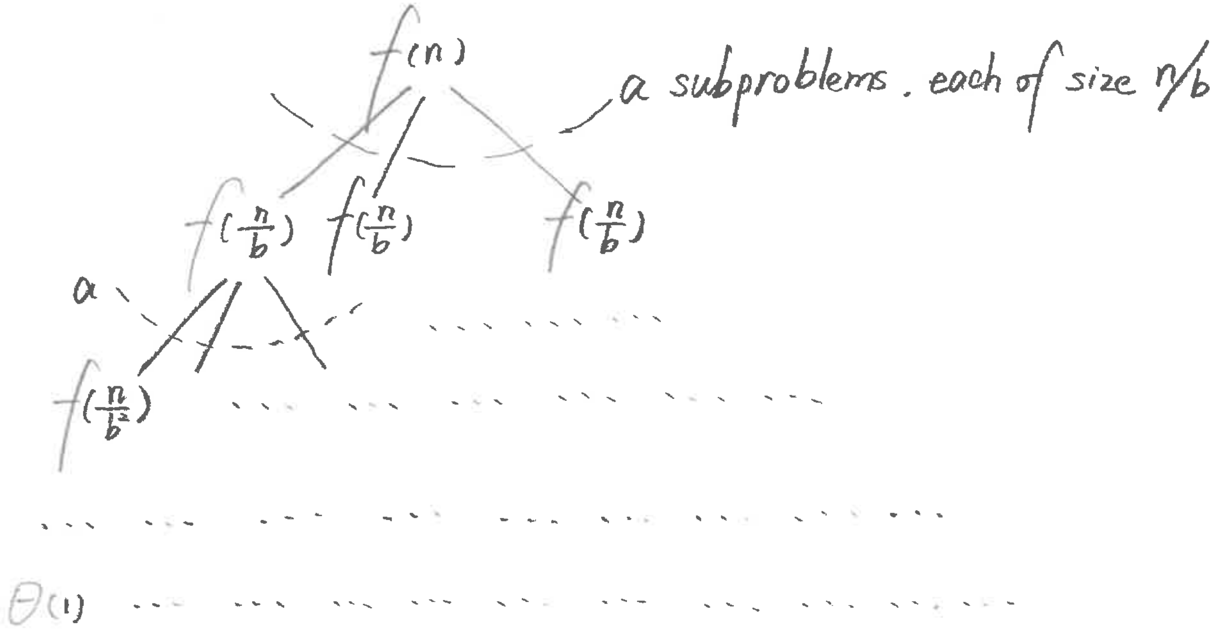
Example:

$T(n) = 4T(\frac{n}{2}) + n^3$

in case 3, so $T(n) = \Theta(n^3)$

proof sketch / intuition

key words: recursion tree, level by level



height of this tree is $h = \log_b n$

number of leaves is $a^h = a^{\log_b n} = n^{\log_b a}$

“dominated by f(n)”

“case 3” is that costs decrease geometrically as we go down the tree

“case 1” is that costs increase geometrically as we go down the tree

“dominated by $\Theta(n^{\log_b a})$”

1st-level cost : $f(n)$

2nd-level cost : $af(\frac{n}{b})$

3rd-level cost : $a^2f(\frac{n}{b^2})$

. . . .

“case 2” is that “the top is $\overset{\text{roughly/asymptotically}}{\text{equal}}$ to the bottom”, the total cost is “one-level cost × height of the tree”, i.e.

$$f(n)\cdot\log_b n \approx f(n)\cdot\log_2 n$$

$\underline{f(n) = \Theta(n^{\log_b a}\cdot\log_2^k n)}$ $\Theta(n^{\log_b a}\cdot\log_2^{k+1} n)$

Lec03 分治法

Divide and Conquer

1. divide the problem (more precisely, the instance of that problem) into subproblems
   should be smaller insome sens
2. conquer each subproblem recursively
3. combine those solutions into a solution for the whole problem

Example: Merge Sort

1. divide the array into two halves
2. conquer recursively sort each subarray
3. combine those solutions (merge two sorted arrays) in linear time

running time: $T(n) = \underset{\text{"number of subproblems"}}{2}T(\overset{\text{"size of subproblems"}}{\tfrac{n}{2}}) + \underset{\text{"extra work"}}{\Theta(n)}$

"in case 2" $= \Theta(n \log_2 n)$

Example: Binary Search // to find x in a sorted array

1. divide: compare x with the middle element in your array
2. conquer: recurse in $\underset{\triangle}{\text{one}}$ subarray
3. combine: trivial (do nothing)

$$T(n) = T(\tfrac{n}{2}) + \Theta(1)$$

$$= \Theta(\log_2 n)$$

Example: Powering a Number // given number x, integer $n \geq 0$, to compute x^n

naive algorithm: $\underbrace{x \cdot x \cdot \cdots \cdot x}_{n \text{ copies of } x \text{ totally}} = x^n$, $\Theta(n)$ time

divide-and-conquer algorithm:

$$x^n = \begin{cases} x^{\frac{n}{2}} \cdot x^{\frac{n}{2}} & \text{, if } x \text{ is even} \\ x^{\frac{n-1}{2}} \cdot x^{\frac{n-1}{2}} \cdot x & \text{, if } x \text{ is odd} \end{cases}$$

$$T(n) = T(\tfrac{n}{2}) + \Theta(1)$$

$$= T(\log_2 n)$$

Example: Fibonacci Numbers // $F_n = \begin{cases} 0 & \text{, if } n=0 \\ 1 & \text{, if } n=1 \\ F_{n-1} + F_{n-2} & \text{, if } n \geq 2 \end{cases}$

naive algorithm: recursive algorithm

$T(n) = \Omega(\varphi^n)$, $\varphi = \frac{1+\sqrt{5}}{2}$

"exponential time" – BAD

"polynomial time – GOOD"

$F_n = F_{n-1} + F_{n-2}$

"you are solving two subproblem of almost the same size" bottom-up algorithm: (better、带记忆表的、cache)

"if you build up the recursion tree for Fibonacci of n, you will see that there are lots of common subtrees"

to compute

$$\begin{tikzcd} F_0 , F_1 , F_2 , F_3 , \cdots , & F_{n-2} , & F_{n-1} , & F_n \arrow[ll, bend left] \arrow[l, bend left] \end{tikzcd}$$

$T(n) = \Theta(n)$

naive recursive squaring (mathematical trick)

$F_n = \varphi^n / \sqrt{5}$ rounded to the nearest integer

$T(n) = \Theta(\log_2 n)$

recursive squaring

Theorem: $\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n}_{\text{similar to " powering a number "}}$

$T(n) = \Theta(\log_2 n)$

proof (induction):

base $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}$ ✓

step $\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \underbrace{\begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}}_{\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ ✓

Example: Matrix Multiplication

input: $A = [a_{ij}]_{n \times n}$, $B = [b_{ij}]_{n \times n}$

output: $C = [c_{ij}]_{n \times n} = AB$

standard algorithm: $\Theta(n^3)$

divide-and-conquer algorithm:

an idea: $n \times n$ matrix $= 2 \times 2$ block matrix of $\frac{n}{2} \times \frac{n}{2}$ sub-metrices

$$\underset{C}{\left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right]} = \underset{A}{\left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right]} \cdot \underset{B}{\left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right]}$$

8 recursive multiplications of $\frac{n}{2} \times \frac{n}{2}$ sub-metrices, and 4 matrix-sum ($\Theta(n^2)$)

$T(n) = 8T(\frac{n}{2}) + \Theta(n^2)$

$= \Theta(n^3)$ "That kind of sucks!"

Strassen's algorithm:

the idea is that we have got to somehow reduce the number of multiplications (from 8 to 7)

$$P_1 = A_{11} \cdot (B_{12} - B_{22})$$
$$P_2 = (A_{11} + A_{12}) \cdot B_{22}$$
$$P_3 = (A_{21} + A_{22}) \cdot B_{11}$$
$$P_4 = A_{22} \cdot (B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$
$$P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$
$$P_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$$

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_5 + P_1 - P_3 - P_7$$

$$T(n) = 7T(\tfrac{n}{2}) + \Theta(n^2)$$
$$= \Theta(n^{\log_2 7}) \qquad \log_2 7 \approx 2.81$$

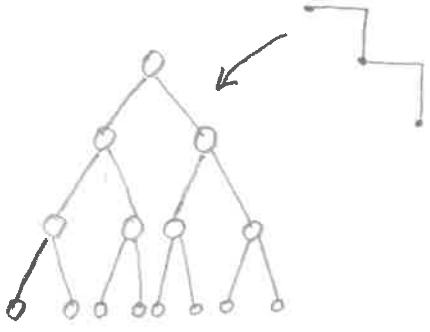
Strassen's algorithm is still not the best algorithm for matrix multiplication, the best $\underbrace{\text{so far}}_{2005}$ is like $n^{2.376}$, getting closer to n^2

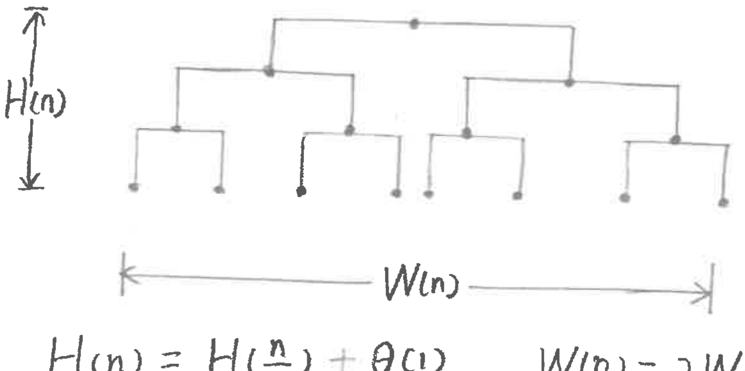
Example: VLSI layout (very large scale integration)

// embed a complete binary tree of n nodes,

// in a grid with minimum area

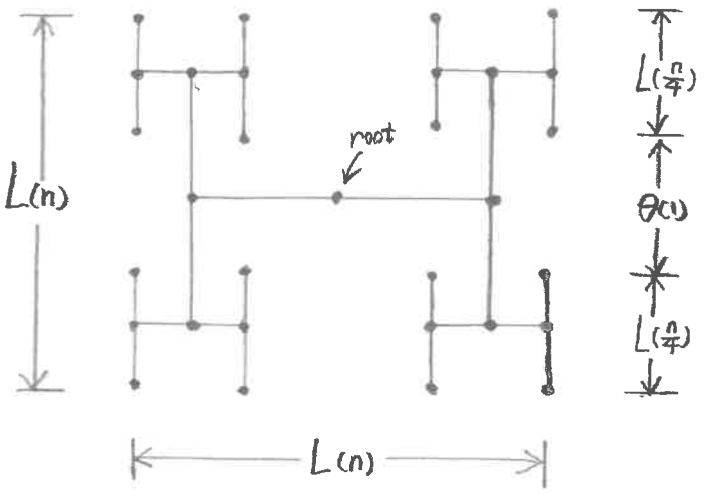
△ naive embedding:





$$H(n) = H(\tfrac{n}{2}) + \Theta(1) \qquad W(n) = 2W(\tfrac{n}{2}) + O(1)$$
$$= \Theta(\log_2 n) \qquad = \Theta(n)$$
$$\text{Area} = \Theta(n\log_2 n)$$

△ the H layout



$$L(n) = 2L(\tfrac{n}{4}) + \theta(1)$$

"case 1" $= \theta(\sqrt{n})$

Goal:

$W(n) = \theta(\sqrt{n})$

$H(n) = \theta(\sqrt{n})$

$\Rightarrow Area = \theta(n)$

Then:

$\log_4 2 = \frac{1}{2}$

$\Rightarrow T(n) = 2T(\frac{n}{4}) + O(n^{\frac{1}{2}-\varepsilon})$

Lec 04 快排及随机化算法

Quick Sort, by Tony Hoare in 1962

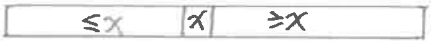
- divide and conquer
- sorts "in place" (rearranges elements where they are)

  节省内存
- very practical (with tuning)

△ divide and conquer

1. divide: (★)

   to partition array into 2 subarrays, around an element called pivot x, such that elements in the lower subarray are less than or equal to x, and elements in the upper subarray are greater than or equal to x.



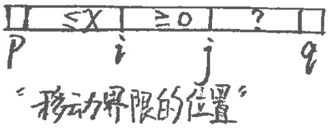
2. conquer:

   to recursively sort the 2 subarrays

3. combine:

   trivial

△ key: linear-time ($\theta(n)$) partitioning subroutine



```
partition (A, p, q)   // A[p...q]
    x ← A[p]     // pivot = A[p]
    i ← p
    for j ← p+1 to q
        do if A[j] ≤ x
            then i ← i+1
                exchange A[i] ↔ A[j]
    exchange A[p] ↔ A[i]
    return i
```

```
Example:  6   10  13  5   8   3   2   11      x ← 6 (pivot)
          i   j
          . . .  . . .  . . .

          6   ⑩   13  ⑤   8   3   2   11
          i   i+1     j

          6   5   13  10  8   3   2   11
              i           j
```

```
6   5   13   10   8   3   2   11
    i                 j

6   5   3   10   8   13   2   11
        i                 j

6   5   3   2   8   13   10   11        (loop terminates)
            i                  j

6   5   3   2   8   13   10   11        (to put the pivot element in the middle
p           i                            between the two subarrays)
```

$$\underbrace{2\quad 5\quad 3}_{\leqslant pivot}\quad \underset{pivot}{\boxed{6}}\quad \underbrace{8\quad 13\quad 10\quad 11}_{\geqslant pivot}$$

```
QuickSort (A, p, q)
    if p < q
        then r ← partition (A, p, q)
             QuickSort (A, p, r-1)
             QuickSort (A, r+1, q)
initial call: QuickSort (A, 1, n)
```

- analysis

worst case:

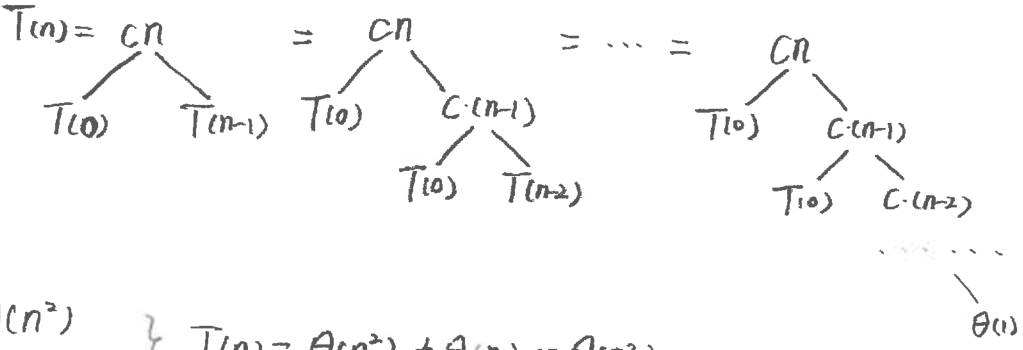
if you always pick the pivot, and everything is greater than or everything is less than this pivot, you are not going to partition the array very well.

⇔ if it is already sorted or reverse sorted

in those cases, one side of each partition has no elements

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \theta(n) \\ &= \theta(1) + T(n-1) + \theta(n) \\ &= T(n-1) + \theta(n) \\ &= \theta(n^2) \qquad \text{(arithmetic series)} \end{aligned}$$

recursion tree for $T(n) = T(0) + T(n-1) + cn$



height = n

$$\left.\begin{aligned} &\theta\Big(\sum_{k=1}^{n} c\cdot k\Big) = \theta(n^2) \\ &n\cdot T(0) = n\cdot\theta(1) = \theta(n) \end{aligned}\right\}\quad T(n) = \theta(n^2) + \theta(n) = \theta(n^2)$$

best case (intuition only):

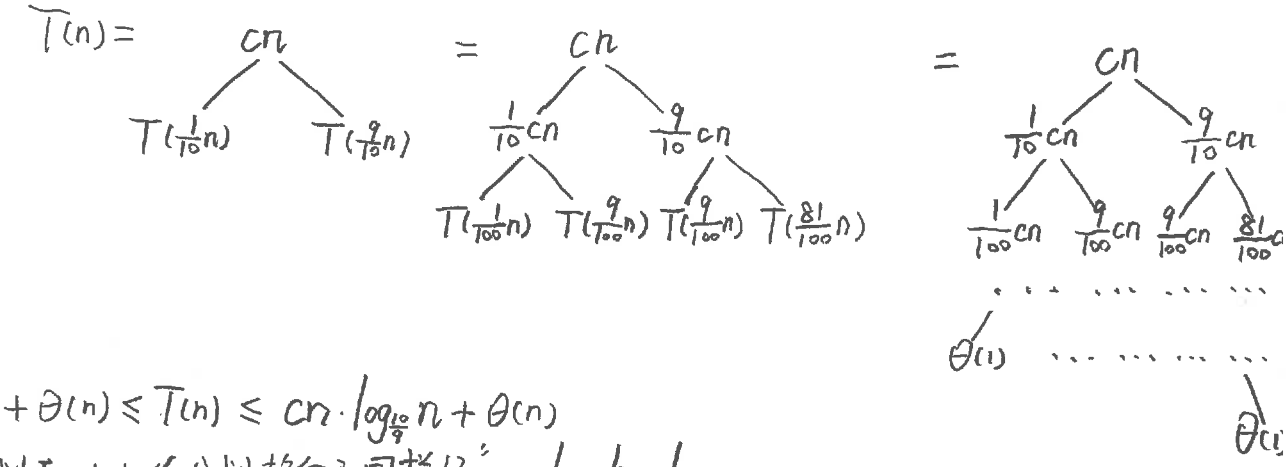
if we are really lucky, partition splits the array $n/2 : n/2$

$$T(n) = 2T(\tfrac{n}{2}) + \Theta(n)$$
$$= \Theta(n\log_2 n)$$

suppose split is always $1/10 : 9/10$.

$$T(n) = T(\tfrac{1}{10}n) + T(\tfrac{9}{10}n) + \underbrace{\Theta(n)}_{\leq cn}$$

recursion tree.



$$cn\log_{10} n + \Theta(n) \leq T(n) \leq cn\cdot\log_{\frac{10}{9}} n + \Theta(n)$$

"1:9 的分划和 1:1 的分划趋向于同样好" lucky!

suppose we alternate lucky, unlucky, lucky, ...

$L(n) = 2U(\frac{n}{2}) + \Theta(n)$, lucky step

$U(n) = L(n-1) + \Theta(n)$, unlucky step

then $L(n) = 2[L(\frac{n}{2}-1) + \Theta(\frac{n}{2})] + \Theta(n)$

$= 2L(\frac{n}{2}-1) + \Theta(n)$

$= \Theta(n\lg n)$ lucky!

how can we ensure that we are usually lucky?

to randomly choose the pivot, randomized-QuickSort,

then (1) the running time is independent of the input ordering

(2) it makes no assumptions about the input distribution

(3) there is no specific input that can elicit the worst-case behavior "引出,探出,诱出"

(4) the worst-case is determined only by a random-number generator

Analysis

T(n) = random variable for the running time assuming that the random numbers are independent for $k = 0, 1, \cdots, n-1$, let $x_k = \begin{cases} 1 & \text{, if partition generates a } k:n-k-1 \text{ split} \\ 0 & \text{, otherwise} \end{cases}$

$$E(x_k) = 0 \cdot P(x_k = 0) + 1 \cdot P(x_k = 1)$$
$$= P(x_k = 1)$$
$$= \frac{1}{n}$$

$$T(n) = \begin{cases} T(0) + T(n-1) + \theta(n) & \text{, if } 0:n-1 \text{ split} \\ T(1) + T(n-2) + \theta(n) & \text{, if } 1:n-2 \text{ split} \\ \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots & \\ T(n-1) + T(0) + \theta(n) & \text{, if } n-1:0 \text{ split} \end{cases}$$
$$= \sum_{k=0}^{n-1} x_k \cdot [T(k) + T(n-k-1) + \theta(n)]$$

$$E(T(n)) = E\left(\sum_{k=0}^{n-1} x_k \cdot [T(k) + T(n-k-1) + \theta(n)] \right)$$
$$= \sum_{k=0}^{n-1} E\left(x_k \cdot [T(k) + T(n-k-1) + \theta(n)] \right)$$

期望的线性叠加性：期望的和等于和的期望

vice versa

$$= \sum_{k=0}^{n-1} E(x_k) \cdot E(T(k) + T(n-k-1) + \theta(n))$$
$$= \frac{1}{n} \sum_{k=0}^{n-1} E(T(k) + T(n-k-1) + \theta(n))$$
$$= \underbrace{\frac{1}{n} \sum_{k=0}^{n-1} E(T(k))} + \underbrace{\frac{1}{n} \sum_{k=0}^{n-1} E(T(n-k-1))} + \underbrace{\frac{1}{n} \sum_{k=0}^{n-1} \theta(n)}_{\theta(n^2)}$$

"identical"

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

"to absorb $k = 0, 1$ terms into $\theta(n)$ for technical convenience"

$$= \frac{2}{n} \sum_{k=2}^{n-1} E(T(k)) + \theta(n)$$

prove: $E(T(n)) \leq a \cdot n \cdot \lg n$ for const $a > 0$

proof: choose a big enough so that $a \cdot n \cdot \lg n > E(T(n))$ for small n

use fact: $\sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$

substitution: $E(T(n)) \leq \frac{2}{n} \sum_{k=2}^{n-1} (a \cdot k \cdot \lg k) + \theta(n)$

$$\leq \frac{2a}{n} \cdot \left(\frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + \theta(n)$$
$$= a n \cdot \lg n - \frac{a}{4} n + \theta(n)$$
$$= \underbrace{a \cdot n \lg n}_{\text{desired}} - \underbrace{\left(\frac{a}{4} n - \theta(n) \right)}_{\text{residual}}$$
$$\leq a \cdot n \cdot \lg n$$, if a is big enough so that "$\frac{an}{4} > \theta(n)$"

Lec 05 线性时间排序

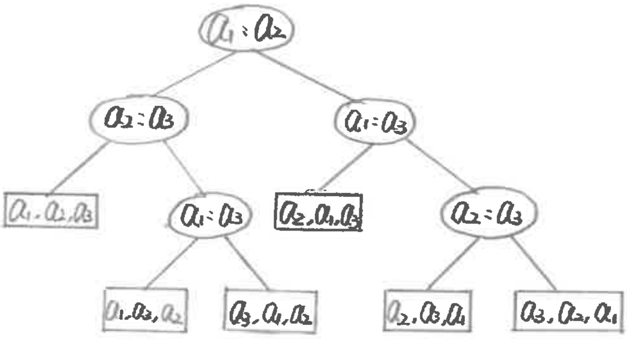
how fast can we sort ?

it depends on what we call the computational model
"what you are allowed to do with the elements"

can we do better than $\Theta(n\lg n)$?

Comparison Sorting Model:
only use comparisons to determine the relative order of elements

Decision-Tree Model: (决策树)
more general than comparison model
Example:
to sort $\langle a_1, a_2, a_3\rangle$



what this tree means is that each node you're making a comparison $(x:y)$. if $x<y$, go left, and go right otherwise. when you get down to a leaf, this is the answer.

Def. in general, $\langle a_1, a_2, \cdots, a_n\rangle$,
△ each internal node (non-leaf node) has a label of the form "$i:j$" where $1\le i,j\le n$, means we compare a_i and a_j,
and we have two subtrees from every such node,
- left subtree which tells you what the algorithm does when $a_i\le a_j$
- right subtree which gives subsequent comparisons if $a_i>a_j$

△ each leaf node gives a permutation, $\langle \pi(1), \pi(2), \cdots, \pi(n)\rangle$,
such that $a_{\pi(1)}\le a_{\pi(2)}\le\cdots\le a_{\pi(n)}$

用决策树的表达方式构建比较排序算法：(转换的过程)

"基于比较的排序可以转换成决策树，但是有一些排序算法无法以决策树的形式表现出来"

- one tree for each n
- view algorithm as splitting into two forks (subtrees) whenever it makes a comparison
- tree lists comparisons along all possible instruction traces
- running time (the number of comparisons) = the length of path
- the worst-case running time = the height of the tree

lower bound on decision-tree sorting:

“下界” any decision-tree sorting n elements has height $\Omega(n\lg n)$ “≥”

proof: the number of leaves must be at least $n!$
(as there are n factorial permutations of an input)
the height of the tree $:= h$, then it has at most 2^h leaves

$\Rightarrow \#\text{leaves} \le 2^h$

$\Rightarrow n! \le 2^h$

$\Rightarrow h \ge \log_2(n!)$

“斯特林公式” $\ge \log_2(\frac{n}{e})^n$

$= n \log_2 \frac{n}{e}$

$= n\log_2 n - n\log_2 e$

$= \Omega(n\lg n)$

corollary: merge sort and heapsort are asymptotically optimal ($n\lg n$), 优
but this is only in the comparison model,
Randomized QuickSort is too in expectation.

“基于比较模型的话, nlgn 就是极限了”

Sorting in Linear Time “不可能比线性时间更快完成排序,因为得遍历数据”

2 algorithms for instance

① counting sort

input: $A[1...n]$, each $A[i]$ is an integer from the range of 1 to k “当k较小时,性能比较好”

output: $B[1...n]$ = sorting of A

auxiliary storage: $C[1...k]$

Counting Sort:

```
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to n
    do C[A[j]] ← C[A[j]]+1          // C[i]表示数值i出现的次数
for i ← 2 to k
    do C[i] ← C[i] + C[i-1]         // C[i]表示 小于等于i的元素的数目 (对前缀做加法)
                                                                   “prefix sum”
for j ← n downto 1                  // distribution
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]]-1
```

Example:

$A = [4, 1, 3, 4, 3]$

$$C = \begin{array}{|c|c|c|c|}\hline 0&0&0&0\\\hline \multicolumn{1}{c}{1}&\multicolumn{1}{c}{2}&\multicolumn{1}{c}{3}&\multicolumn{1}{c}{4}\end{array} \xrightarrow{\text{counting}} C = \begin{array}{|c|c|c|c|}\hline 1&0&2&2\\\hline \multicolumn{1}{c}{1}&\multicolumn{1}{c}{2}&\multicolumn{1}{c}{3}&\multicolumn{1}{c}{4}\end{array}$$

$$\xrightarrow{\text{prefix sum}} C' = \begin{array}{|c|c|c|c|}\hline 1&1&3&5\\\hline \multicolumn{1}{c}{1}&\multicolumn{1}{c}{2}&\multicolumn{1}{c}{3}&\multicolumn{1}{c}{4}\end{array}$$

$j=5$, $A[j]=3$, $B = \begin{array}{|c|c|c|c|c|}\hline \ &\ &3&\ &\ \\\hline\end{array}$

$C'[A[j]]=3$ ← "值为3的元素应该放在位置3或者更靠前"

$C'=[1,1,\textcircled{2},5]$

$j=4$, $A[j]=4$, $B = \begin{array}{|c|c|c|c|c|}\hline \ &\ &3&\ &4\\\hline\end{array}$

$C'[A[j]]=5$

$C'=[1,1,2,\textcircled{4}]$

. . . .

$B = \begin{array}{|c|c|c|c|c|}\hline 1&3&3&4&4\\\hline\end{array}$

$T(n) = O(k+n)$,

it is a great algorithm if k is relatively small, like at most n.

so you could write a combination algorithm that if $k > n\lg n$ and if $k \le n\lg n$

a stable sorting algorithm preserves the relative order of equal elements, counting sort is stable

② radix sort 基数排序 by Herman Hollerith in 1890

radix sort is going to work for a much larger range of numbers in linear time

first. sort by the most significant digit first (need many boxes)

right: (by Hollerith) sort by the least significant digit first, using stable sorting

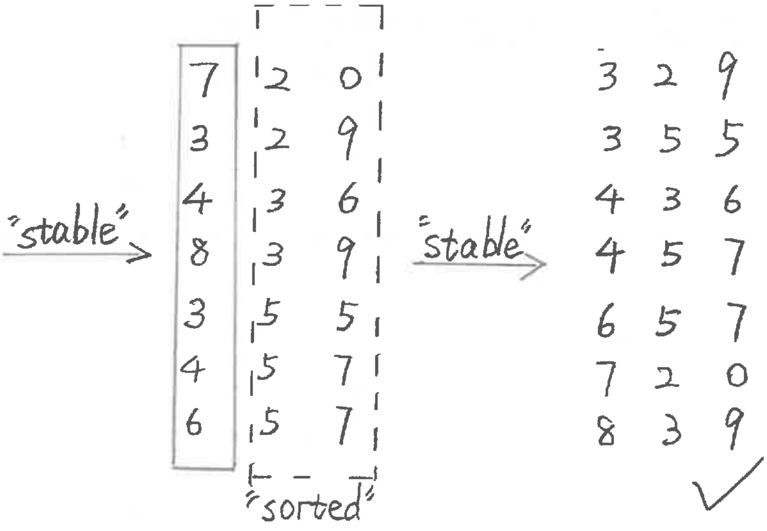
> Hollerith 在1911年创建制表机公司(tabulating machine company),
> 然后在1924年合并了其他几个公司,组成了 IBM

the whole idea is that we are doing a digit-by-digit sort,
from least significant digit to most significant digit

the nice thing about this algorithm is that there are no bins, it's one big bin at all times

Example:

$$\begin{array}{ccc}3&2&9\\4&5&7\\6&5&7\\8&3&9\\4&3&6\\7&2&0\\3&5&5\end{array} \xrightarrow{\text{"LSB"}} \begin{array}{cc|c|}3&2&9\\4&5&7\\6&5&7\\8&3&9\\4&3&6\\7&2&0\\3&5&5\end{array} \xrightarrow{\text{"stable"}} \begin{array}{c|c|c}7&2&0\\3&5&5\\4&3&6\\4&5&7\\6&5&7\\3&2&9\\8&3&9\end{array}$$



"when I have equal elements here, I have already sorted the suffix"

"好的部分是我们不用分成一个一个箱子了，而是始终把它们放在一个大箱子里面。"

Correctness:

to induct on the digit position $\overset{t}{\vee}$ that we are currently sorting,

assume that by induction that it is already sorted on lower $t-1$ digits,

and then the next thing we do is to sort on the t-th digit.

if two elements are the same (has the same t-th digit),

stability ⇒ keep the order ⇒ still sorted

else,

put them in the right order

next we are going to use counting sort for each round

(we could use any sorting algorithm we want for individual digits)

Analysis:

- use counting sort ($O(k+n)$)
- say n integers, each b bits long ($0 \sim 2^b-1$) "可以把几个比特放在一起当做一位数处理，相当于八进制、十六进制等"
- split each integer into $\underbrace{b/r}_{\text{"需要运算的轮数"}}$ "digits", each "digit" is r bits long "基于2^r进制来表示这个数" "counting sort 的 $k=2^r$"

$$T(n) = O\left(\frac{b}{r}\cdot(n+2^r)\right)$$

$$\min_r T(n)$$

$$r = \log_2 n,\quad T(n) = O\left(\frac{bn}{\log_2 n}\right)$$

if numbers (integers) are in the range $0 \sim 2^b-1$
$0 \sim n^d-1$

then $T(n) = O(d\cdot n)$

Lec06 顺序统计、中值

Order Statistics

given n elements in array (unsorted), to find the k-th smallest element

naive algorithm: to sort, and then return the k-th element

$k=1$ minimum

$k=n$ maximum

$k=\lfloor \frac{n+1}{2} \rfloor$ or $\lceil \frac{n+1}{2} \rceil$ medians (typically)

randomized divide and conquer algorithm:

```
(pseu-code) Rand-Select (A, p, q, i)   // to find the i-th smallest in A[p...q]
                                        "找第i小的元素"
    if p = q then
        return A[p]
    r ← Rand-Partition(A, p, q)
    k ← r - p + 1        // A[r] is the k-th smallest element in A[p...q]
    if i = k then return A[r]
    elif i < k then
        return Rand-Select (A, p, r-1, i)
    else (i > k) then
        return Rand-Select (A, r+1, q, i-k)
```

Example:

$A = 6, 10, 13, 5, 8, 3, 2, 11 \quad , i = 7$

$$A = \begin{array}{|c|c|c|c|c|c|c|c|}\hline \underset{\substack{\uparrow \\ \text{pivot}}}{6} & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\ \hline \end{array}$$

$$A' = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 2 & 5 & 3 & \underset{\uparrow}{6} & 8 & 13 & 10 & 11 \\ \hline \end{array}$$

$r=4$，"意味着如果我们一开始对A进行排序，6一定会在index=4的位置(从1开始)"，

即"6是第4小的元素"。

$k = r-p+1 = 4-1+1 = 4 < 7 = i$

$$A' = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 2 & 5 & 3 & \underset{\uparrow}{6} & 8 & 13 & 10 & 11 \\ \hline \end{array}$$

$\underbrace{8 \; 13 \; 10 \; 11}_{\text{"找这里第 } i-k=3 \text{ 大的元素"}}$

Intuition for Analysis:

(today assume distinct elements)

lucky case: (1/10 : 9/10 for example)

$$T(n) \leq T(\tfrac{9}{10}n) + \Theta(n)$$
$$= \Theta(n)$$

unlucky case: (0 : n-1)

$$T(n) = T(n-1) + \Theta(n)$$
$$= \Theta(n^2) \quad \text{"arithmetic"}$$

Analysis of Expected Time:

- let $T(n)$ be the random variable for running time of Random-Select on an input of size n, assuming random numbers are chosen independently
- define indicator random variable x_k $(k=0,1,2,\cdots,n-1)$.

$$x_k = \begin{cases} 0, & \text{otherwise} \\ 1, & \text{if the partition comes out that } k \text{ on the left-hand side } (k : n-k-1 \text{ split}) \end{cases}$$

$$T(n) \leqslant \begin{cases} T(\max\{0,n-1\}) + \theta(n) & , \text{if } 0:n-1 \text{ split} \\ T(\max\{1,n-2\}) + \theta(n) & , \text{if } 1:n-2 \text{ split} \\ \cdots \quad \cdots \quad \cdots \quad \cdots & \cdots \\ T(\max\{n-1,0\}) + \theta(n) & , \text{if } n-1:0 \text{ split} \end{cases}$$

$$= \sum_{k=0}^{n-1} x_k \cdot \left[T(\max\{k,n-1-k\}) + \theta(n)\right]$$

$$E(T(n)) = E\left(\sum_{k=0}^{n-1} x_k \cdot [T(\max\{k,n-1-k\}) + \theta(n)]\right)$$

$$= \sum_{k=0}^{n-1} E\left(x_k \cdot [T(\max\{k,n-1-k\}) + \theta(n)]\right)$$ ← "递归中每次产生的随机数，都独立于首次调用时产生的随机数"

$$= \sum_{k=0}^{n-1} E(x_k) \cdot E(T(\max\{k,n-1-k\}) + \theta(n))$$

$$= \sum_{k=0}^{n-1} \frac{1}{n} \cdot E(T(\max\{k,n-1-k\}) + \theta(n))$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E(T(\max\{k,n-1-k\})) + \frac{1}{n}\sum_{k=0}^{n-1}\theta(n)$$

$$\leqslant \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} E(T(k)) + \theta(n)$$

claim: $E(T(n)) \leqslant c \cdot n$ for sufficiently large constant $c > 0$

proof: (substitution method)

assume true for "$<n$"

$$E(T(n)) \leqslant \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} \underbrace{E(T(k))}_{\text{"}\leqslant ck \text{ by hypothesis"}} + \theta(n)$$

$$\leqslant \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} c \cdot k + \theta(n)$$

$$= \frac{2c}{n}\underbrace{\sum_{k=\lfloor n/2 \rfloor}^{n-1} k}_{\text{"}\leqslant \frac{3}{8}n^2\text{"}} + \theta(n)$$

$$= c \cdot n - \underbrace{\left(\frac{1}{4}cn - \theta(n)\right)}_{\text{"取任意大的}c/\exists c/c\text{足够大时，非负"}}$$

the fact: Random-Select has expected running time $\theta(n)$,
in the worst case $\theta(n^2)$.

"如果最坏情况的复杂度是$\theta(n)$就好了，那就是最好的结果。毕竟所有的数字(元素)都得看一遍，因此复杂度不可能小于$\theta(n)$"

"怎么避免随机性呢？"

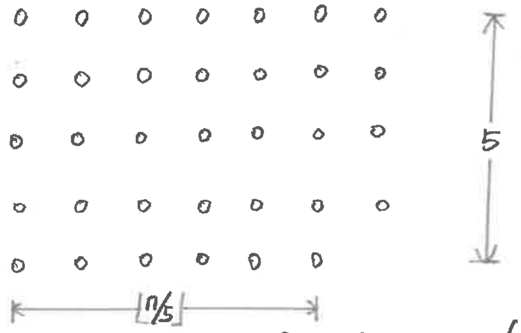
worst-case、linear-time order statistics {Blum、Floyd、Pratt、Rivest、Tarjan} in 1973

"RSA的R"

△idea: generate good pivot recursively

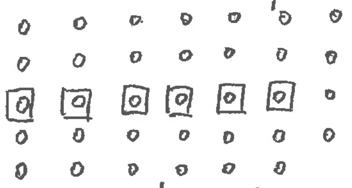
Select (i, n): "找第i大的元素" "数组的大小"

① divide the n elements into ⌊n/5⌋ groups of 5 elements each

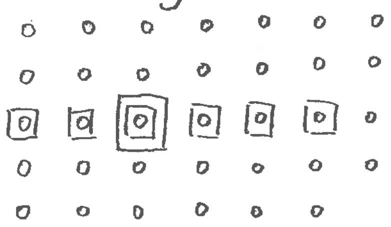


"每一列都是一组"

find the median of each group $(\Theta(n))$



② recursively select the median x of the ⌊n/5⌋ group-medians $(T(n/5))$

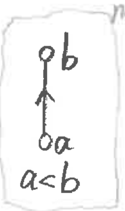


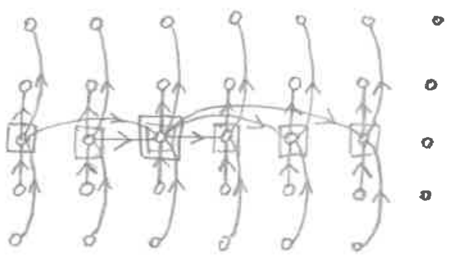
```
③ partition with $x$ as partition element,
   let $k = \text{rank}(x)$ := "x是数组的k-th smallest element"
④ if $i = k$ then
       return $x$
   elif $i < k$ then
       recursively select the i-th smallest element in the lower part of the array
   else $(i > k)$ then
       recursively select the $(i-k)$-th smallest element in the upper part of the array
```

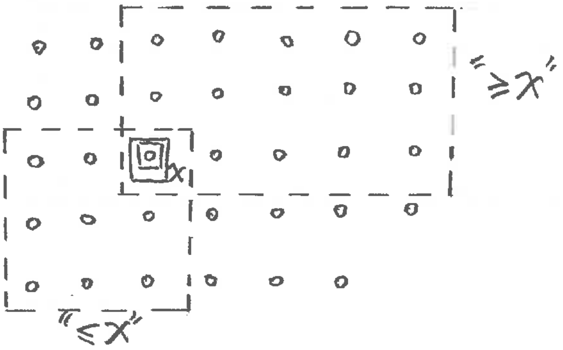
same as Rand-Select

Analysis:

notation







"已知大小关系（与x）的元素有多少个呢？"

at least (guaranteed),

$\geq 3\lfloor\lceil n/5\rceil/2\rfloor$ elements LTE x

$\geq 3\lfloor\lceil n/5\rceil/2\rfloor$ elements GTE x ← "避免了极端情况的出现"

(as $\lfloor\lceil n/5\rceil/2\rfloor$ $\underbrace{\text{group-medians}}_{}$ $\leq x$, as well as $\lfloor\lceil n/5\rceil/2\rfloor$ $\underbrace{\text{group-medians}}_{}$ $\geq x$)

"再根据传递关系，得到前面的系数3"

所以，至少有大概 $\frac{3}{10}$ 的元素在 x 的左边，也至少有大概 $\frac{3}{10}$ 的元素在 x 的右边。

简化一下，

for $n \geq 50$, $3\lfloor n/10\rfloor \geq \frac{n}{4}$ ← 最坏也只是 $\frac{1}{4}:\frac{3}{4}$ 的分法

"a powerful subroutine"!

$$T(n) \leq T(\tfrac{n}{5}) + \underbrace{T(\tfrac{7}{10}n)}_{} + \theta(n)$$

$\frac{3}{10}:\frac{7}{10}$ 的分法

$$\approx T(\tfrac{n}{5}) + \underbrace{T(\tfrac{3}{4}n)}_{} + \theta(n)$$

用"简化一下"部分的结论（$n \geq 50$ 时，$3\lfloor n/10\rfloor \geq \frac{n}{4}$）

claim: $T(n) \leq c \cdot n$

proof: (substitution)

assume true for smaller n

$$T(n) \leq c\cdot\tfrac{1}{5}n + c\cdot\tfrac{3}{4}n + \theta(n)$$

$$= \tfrac{19}{20}\cdot c\cdot n + \theta(n)$$

$$= cn - \underbrace{(\tfrac{1}{20}cn - \theta(n))}_{\text{if } c \text{ is large enough, non-negative}}$$

$$\leq cn, \text{ for } c \text{ sufficiently large}$$

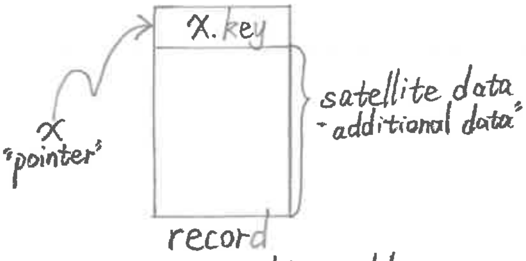
注：5是这个算法能成功的最小数字。

3不行！7行！

Lec 07 哈希表

symbol-table problem (in compilers):

table S holding n records



operations on this table

- insert (S, x): $S \leftarrow S \cup \{x\}$
  "insert a record into this table"
- delete (S, x): $S \leftarrow S - \{x\}$

} dynamic set

- search (S, k): return x such that x.key = k or nil if no such x
  "search for a given key"

direct access table

"it works when the keys are drawn from small distribution"

suppose keys are drawn from $U = \{0, 1, \cdots, m-1\}$.

assume the keys are distinct,

set up an array $T[0 \ldots m-1]$ to represent the dynamic set S.

such that $T[k] = \begin{cases} x, & \text{if } x \in S \text{ and } x.key = k \\ nil, & \text{otherwise} \end{cases}$

相当于存放指针的数组

all operations take constant time in the worst case

limitations: ① m should be small

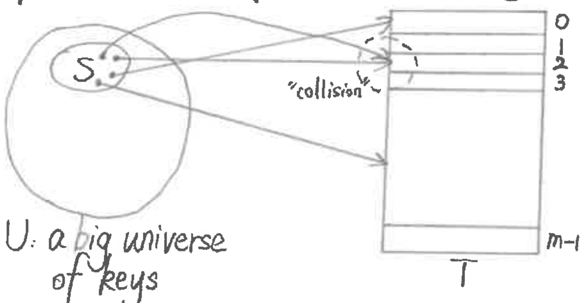
② even worse, most of the table would be empty in some case

"我们希望在保存记录的同时，让表的规模尽可能的小、保留某些特性"

Hashing

a hash function h maps keys "randomly" into slots of table T

"数组的索引"



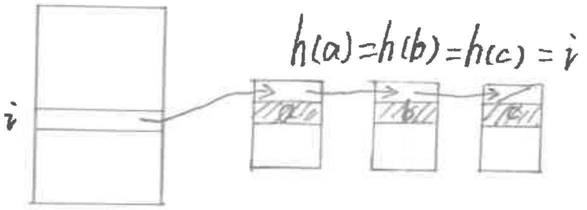
when a record (to be inserted) maps to an already occupied slot, a collision occurs

"对每个槽创建一个链表，把所有映射到这个槽的元素都存放到这个槽的链表里面去"

resolving collisions by chaining!

the idea is to link records in the same slot into a list

Example:



Analysis:

worst-case: every key hashes to the same slot （所有键都哈希映射到同一个槽）

access takes $\Theta(n)$ time if $|S| = n$

average-case: assumption of <u>simple uniform hashing</u>

"each key $k \in S$ is equally likely to be hashed to any slot in T, independent of where other keys are hashed"

Def. the load factor of a hash table with n keys at m slots is $\alpha = \frac{n}{m}$ = average number of keys per slot

expected unsuccessful search time $= \Theta(\underset{\text{hash \& access slot}}{1} + \overset{\text{search list}}{\alpha})$

expected search time $= \Theta(1)$ if $\alpha = O(1)$, i.e., if $n = O(m)$

expected successful search time $= \Theta(1+\alpha)$ too.

Choosing a Hash Function

- should distribute keys uniformly into slots
- regularity in key distribution should not affect uniformity

"键值分布的特点、"

Example: division method, $h(k) = k \bmod m$

don't pick m (with small divisor d)!

if $d=2$ (i.e. m is even) and all keys are even, then odd slots never used

"regularity in key distribution"

if $m = 2^r$, then hash doesn't depend on all bits of k

$$k = 1011000111\underbrace{011010}_{h(k)} \qquad r=6, \; m=2^6$$

pick m = prime (质数) not too close to a power of 2 or 10 (有很多关于质数的好定理)

Example: multiplication method

槽的数量 $m = 2^r$, and computer has w bit words

$$h(k) = (A \cdot k \bmod 2^w) \underbrace{\text{rsh}}_{\text{"right shifted"}} (w-r)$$

A: an odd integer in the range $2^{w-1} < A < 2^w$

fast method! (faster than division)

if $m = 8 = 2^{3}$ ($r=3$), $w=7$, $A = 1011001$, $k = 1101011$

then $A \cdot k = 10010100110011$

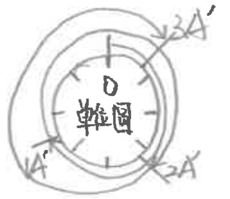
$A \cdot k \bmod 2^w$ = (忽略前几位，只取后W位) 0110011

$(A \cdot k \bmod 2^w)$ rsh $(w-r)$ = 011 = $h(k)$

$$\begin{array}{r} 1011001 = A \\ \overset{\text{"times"}}{\times}\; 1101011 = k \\ \hline \underbrace{1001010}_{\text{high-order ignored}} \,\vdots\, \underbrace{011}_{h(k)}\,\overrightarrow{\;0011\;}_{\text{rsh}} \end{array}$$

modular wheel for intuition:

$$\begin{array}{r} 1011001 = A \\ \times\ 1101011 = k \end{array} \longrightarrow \begin{array}{r} \underset{\uparrow\ \text{小数点}}{.}1011001 = A' \\ \times\ 1101011 = k \end{array}$$



resolving $\underbrace{\text{collisions}}_{\text{回到一开始那个问题}}$ by open addressing - no storage for links

- the idea is that to probe the table systematically, until an empty slot is found
- $h : \underset{\substack{\uparrow \\ \text{universe of keys}}}{U} \times \underset{\substack{\uparrow \\ \text{probe number}}}{\{0, 1, \cdots, m-1\}} \longrightarrow \underset{\substack{\uparrow \\ \text{slot}}}{\{0, 1, \cdots, m-1\}}$
- the probe sequence should be permutation of 0 to m-1
- the table may actually fill up in the end ($n \leq m$, n is #elements, m is #slots)
- deletion is difficult, yet not impossible

"有人按照探查序列来查找另一个键，他本应先发现这里不是他要的键，继而再向下查找，然而现在却发现这个槽是空的"

Example: insert $k=496$ into table as below

586
133
204
481

0-step: probe $h(496, 0)$
假设哈希映射到204这个槽，发现已经被占了.

1-step: 则再探查一次，$h(496, 1)$
假设哈希映射到586这个槽，发现已经被占了，

2-step: $h(496, 2)$
假设哈希映射到一个空槽．则把键放入这个槽。

search is the same probe sequence.

if successful, it finds the record.

if unsuccessful, it finds nil

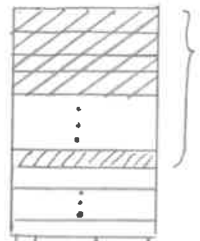
probing strategies for open addressing

- linear probing

  $h(k,i) = (h(k,0) + i) \bmod m$

  "一个个地查找"

  "primary clustering": long runs of filled slots



如果一连块区域都被占用了，那接下来都得先遍历到这个区域的底部

- double hashing probing

  $h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m$

  excellent!

  usually pick $m = 2^r$ and $h_2(k)$ to be odd

Analysis of open addressing

assumption of uniform hashing: each key is equally likely to have any one of the $m!$ permutations as its probe sequence is independent of other keys

Theorem: the expected rumber of probes is at most $\frac{1}{1-\alpha}$ if $\alpha < 1$

键数小于槽数 这是开放寻址法的前提条件

proof: (unsuccessful search)

1st probe always necessary,

with $\frac{n}{m}$ probability, we have a collision ⇒ 2nd probe necessary

(you are not going to hit the same slot) with probability $\frac{n-1}{m-1}$, collision ⇒ 3rd probe nec.

. . . . $\frac{n-2}{m-2}$ . . . .

note $\frac{n-i}{m-i} < \frac{n}{m} = \alpha$ for $i = 1, 2, \cdots, n-1$

$$E(\#\text{probes}) = 1 + \frac{n}{m}\left(1 + \frac{n-1}{m-1}\left(1 + \frac{n-2}{m-2}\left(\cdots\left(1 + \frac{1}{m-n}\right)\cdots\right)\right)\right)$$

$$\leq 1 + \alpha(1 + \alpha(1 + \alpha(\cdots(1 + \alpha)\cdots)))$$

"每步都有连锁的影响"

$$\leq 1 + \alpha + \alpha^2 + \alpha^3 + \cdots$$

$$= \sum_{i=0}^{\infty} \alpha^i$$

$$= \frac{1}{1-\alpha} \quad \text{geometric series}$$

const $\alpha < 1$ ⇒ $O(1)$ probes

if $\alpha = 0.5$ (i.e. 50% full), then 2 probes, if 90% full, then 10 probes (急剧上升)

Lec 08 全域哈希与完全哈希

addressing a fundamental weakness of hashing,
for any choice of hash function, there exists a bad set of keys that all hash to the same slot,
the idea is to choose a hash function at random, independently from the keys

the name of the scheme is universal hashing (全域哈希)

Def. let U be a universe of keys, and let H be a finite collection of hash functions, mapping U to the slots in our hash table $\{0, 1, \cdots, m-1\}$, say that H is universal if for all pairs of distinct keys ($\forall x, y \in U$ and $x \neq y$), the following is true:

$$\left|\{h \in H : h(x) = h(y)\}\right| = \frac{|H|}{m}$$

"在函数集 H中，对于任意键对，能将它们（指键对）哈希映射到同个位置的哈希函数的数目等于 $\frac{|H|}{m}$"

"也可以这样看，如果哈希函数 h 是随机地从函数集 H 里选出的，那么 x 与 y 发生碰撞的概率是 $\frac{1}{m}$"

Thm. choose h randomly from H, suppose we're hashing n keys into m slots in table T, then for given key x, the expected number of collisions with x is less than $\frac{n}{m}$, i.e. $E(\#\text{collisions with } x) < \underbrace{\frac{n}{m}}_{\text{"}\alpha\text{: the load factor of the table"}}$.

proof:

let C_x be the random variable denoting the total number of collisions of keys in T with x, and let $c_{xy} = \begin{cases} 1, & \text{if } h(x) = h(y) \\ 0, & \text{otherwise} \end{cases}$

note that $E(c_{xy}) = \frac{1}{m}$ and $C_x = \sum_{y \in T, y \neq x} c_{xy}$

$$\begin{aligned} E(C_x) &= E\Big(\sum_{y \in T, y \neq x} c_{xy}\Big) \\ &= \sum_{y \in (T - \{x\})} E(c_{xy}) \quad \longleftarrow \text{期望的线性性质} \\ &= \sum_{y \in (T - \{x\})} \frac{1}{m} \\ &= \frac{n-1}{m} \quad \text{Q.E.D.} \end{aligned}$$

constructing an universal hash function (一种构造全域哈希的方法)

① let m be prime (质数),
decompose any key k in our universe into $r+1$ digits: $k = \langle k_0, k_1, \cdots, k_r \rangle$, $0 \leq k_i \leq m-1$
"这种做法的思想是把 k 用 m 进制来表示"

② pick an a at random, $a = \langle a_0, a_1, \cdots, a_r \rangle$ "同样的，我们也把 a 看成是 m 进制数",
each a_i is chosen randomly from $\{0, 1, 2, \cdots, m-1\}$, "a_i 是随机的 m 进制数,

③ define $h_a(k) = \Big(\underbrace{\sum_{i=0}^{r} a_i k_i}_{\vec{a}\text{与}\vec{k}\text{的点乘}}\Big) \bmod m$

how big is $H = \{h_a\}$? ans: m^{r+1}

Thm: H is universal.

proof: let $x = <x_0, x_1, \cdots, x_r>$

$y = <y_0, y_1, \cdots, y_r>$ be distinct keys

x and y differ in at least one digit,

without loss of generality, position 0.

for how many hash functions $h_a \in H$, do x and y collide?

must have $h_a(x) = h_a(y)$ if they collide

$$\Rightarrow \sum_{i=0}^{r} a_i x_i \equiv \sum_{i=0}^{r} a_i y_i \pmod m$$

$$\Rightarrow \sum_{i=0}^{r} a_i (x_i - y_i) \equiv 0 \pmod m$$

$$\Rightarrow a_0(x_0 - y_0) + \sum_{i=1}^{r} a_i(x_i - y_i) \equiv 0 \pmod m$$

$$\Rightarrow a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \pmod m$$

number theory fact: let m be prime, for any $z \in \mathbb{Z}_m$ (integers mod m), $z \not\equiv 0$, $\exists$ unique $z^{-1} \in \mathbb{Z}_m$, $z \cdot z^{-1} \equiv 1 \pmod m$

Example.

$m = 7$

z	1	2	3	4	5	6
z^{-1}	1	4	5	2	3	6

since $x_0 \neq y_0$, $\exists (x_0 - y_0)^{-1}$,

$$\Rightarrow a_0 \equiv \left(-\sum_{i=1}^{r} a_i(x_i - y_i)\right) \cdot (x_0 - y_0)^{-1}$$

"$a_0, a_1, a_2, \cdots, a_r$ 线性相关"

☆ 若两个互异的键被哈希到同一个位置上，那么a_0实际上由其它所有的a_i所决定

thus, for any choice of $a_1, a_2, \cdots, a_r$, exactly 1 of the m choices for a_0 will cause x and y to collide, and no collision for other $m-1$ choices for a_0

so the number of hash functions that cause x and y to collide

$$= \underbrace{m}_{\text{choices for } a_1} \cdot \underbrace{m}_{\text{choices for } a_2} \cdot \cdots \cdot \underbrace{m}_{\text{choices for } a_r} \cdot \underbrace{1}_{\text{choices for } a_0\text{, this value } \left(-\sum_{i=1}^{r} a_i(x_i - y_i)\right) \cdot (x_0 - y_0)^{-1}}$$

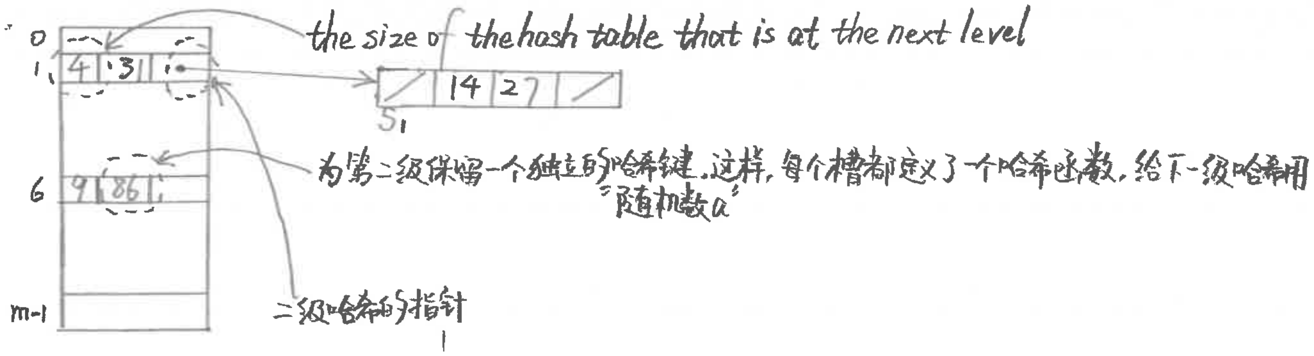
$$= m^r = \frac{|H|}{m} \quad \text{Q.E.D.}$$

perfect hashing (完全哈希)

"suppose I give you a set of keys, build a static table for me, so I can look up whether the key is in the table"

given n keys, construct a static hash table of size $m = O(n)$, such that search takes $O(1)$ time in the worst case

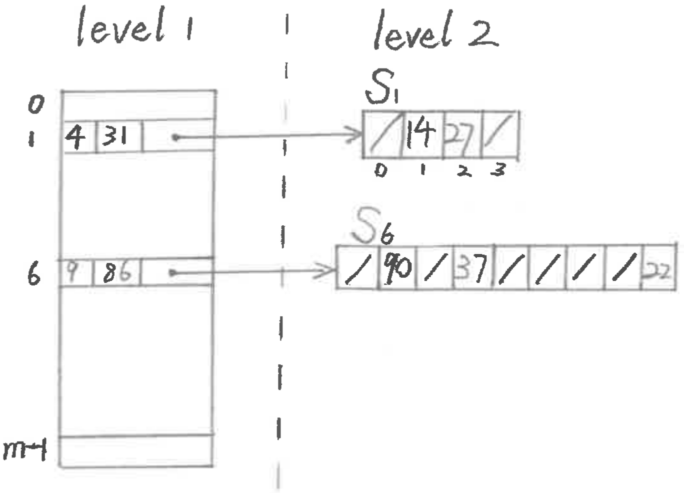
the idea is to use a two-level scheme (双级结构), with universal hashing at both levels, so that no collisions at level two



the size of the hash table that is at the next level

为第二级保留一个独立的哈希键，这样，每个槽都定义了一个哈希函数，给下一级哈希用“随机数a”

二级哈希的指针



$h(14)=h(27)=1$

$h_{31}(14)=1$
$h_{31}(27)=2$

如果能保证在第二级没有碰撞，那么只需要花费O(1)的时间就能在最坏情况下完成对数据的查找。

if n_i items that hash to level-one's slot i, then use $m_i = n_i^2$ slots in the level-two hash table.

“此时，第二级表将会非常稀疏”

and what I am going to show is that under those circumstances, it's easy for me to find hash functions such that there are no collisions.

Analysis for Level 2

Thm. hash n keys into $m=n^2$ slots, using a random hash function in an universal set H, then the expected number of collisions is less than $\frac{1}{2}$.

proof: the probability that 2 given keys collide under h is $\frac{1}{m}=\frac{1}{n^2}$,

C_n^2 pairs of keys.

therefore, $E(\#\text{collisions}) = C_n^2 \cdot \frac{1}{n^2} = \frac{1}{2}\cdot\frac{n-1}{n} < \frac{1}{2}$ Q.E.D.

Markov Inequality

for random variable x which is bounded below by 0,

$P\{x \geq t\} \leq \frac{E(x)}{t}$

proof:

$$E(x) = \sum_{x=0}^{\infty} x \cdot P(x)$$
$$\geq \sum_{x=t}^{\infty} x \cdot P(x)$$
$$\geq \sum_{x=t}^{\infty} t \cdot P(x)$$
$$= t \cdot \sum_{x=t}^{\infty} P(x)$$
$$= t \cdot P(x \geq t) \quad \text{Q.E.D.}$$

Corollary: $P\{\text{no collisions}\} \geq \frac{1}{2}$

proof: $P\{\text{at least one collision}\} \leq E(\#\text{collisions})/1$

$$< \frac{1}{2}$$

$P\{0 \text{ collision}\} = 1 - P\{\geq 1 \text{ collision(s)}\} \geq \frac{1}{2}$ Q.E.D.

So to find a good level-2 hash function,
just test a few at random, and we will find one quickly,
since at least half will work. (可行性分析)

Analysis for Storage (证明是 $O(n)$ 大小)

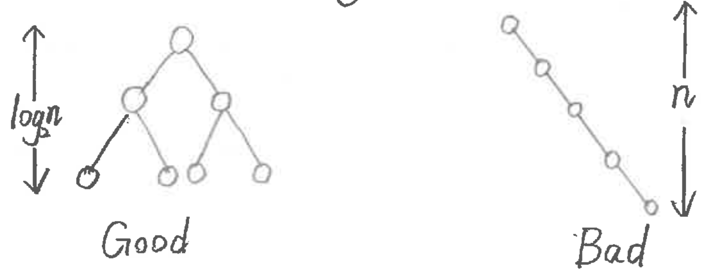
for level 1, choose $m = n$,
and let n_i be the random variable for the number of keys that hash to slot i in T
use $m_i = n^2$ slots in each level 2 table S_i

$E(\text{total storage}) = n + E\left(\sum_{i=0}^{m-1} \Theta(n_i^2)\right)$ ← 桶排序里的知识

$= \Theta(n)$ by bucket-sort analysis

Lec 09　二叉搜索树

(Randomly Built) Binary Search Trees, BSTs for short



△ BST sort (A):

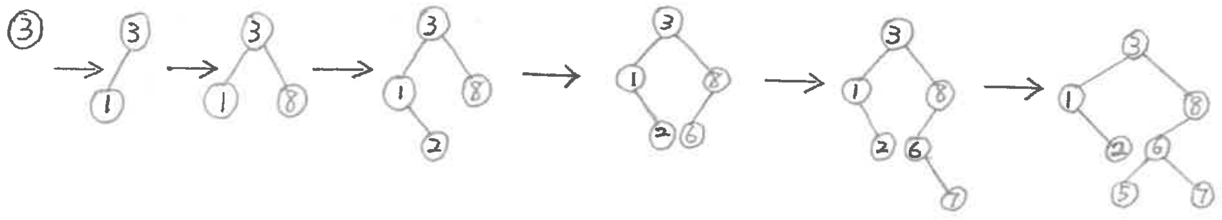
// build the BST and then traverse it in order

```
T ⟵ $\phi$
for i ⟵ 1 to n
    do Tree-Insert (T, A[i])
Inorder-Tree-Walk (T.root)
```

Example:

A =

3	1	8	2	6	7	5



Time:

$O(n)$ for walk,

$\Omega(n\lg n)$ for n Tree-Inserts, meanwhile, $O(n^2)$ for n Tree-Inserts

best case is a perfectly balanced tree　　　worst case is the array is already sorted/reverse-sorted

if already sorted/reverse-sorted, then it's a bad shape!

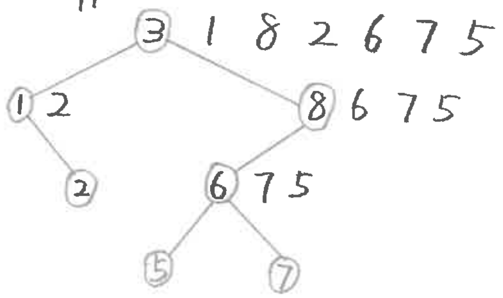
if lucky, it is a balanced tree with $O(\lg n)$ height $\Longrightarrow$ $O(n\lg n)$ time

Quicksort?

it turns out the running time of this algorithm is the same as the running time of quicksort.

Relation to Quicksort

BST sort and Quicksort make the same comparisons, but in a different order.



△ Randomized BST sort

① randomly permuted the array A

② BST sort(A)

Time = time(randomized Quicksort), i.e.

$$E[\text{time(randomized BST sort)}] = E[\text{time(randomized Quicksort)}] = \Theta(n\lg n)$$

randomly built BST = the tree resulted from randomized BST sort

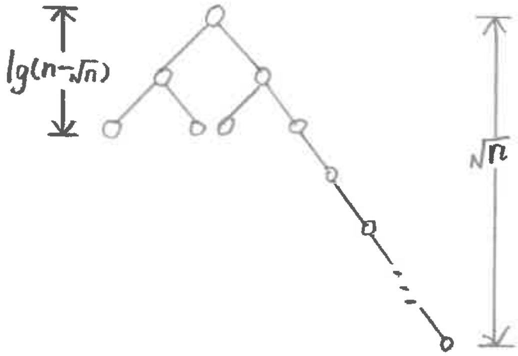
$$\text{time(BST sort)} = \sum_{\text{node}} \text{depth(node)}$$ ←"所有结点的深度的和"

$$\Longrightarrow E(\text{BST sort}) = \Theta(n\lg n)$$

$$E\left[\frac{1}{n}\sum_{\text{node}} \text{depth(node)}\right] = \Theta(n\lg n)/n = \Theta(\lg n)$$ ←"树结点的平均深度"

并不意味着树的高度就是lgn

Example:



$$\text{avg-depth} \le \frac{1}{n}(n\lg n + \sqrt{n}\cdot\sqrt{n}) = O(\lg n)$$

说明：只知道平均深度是 lgn 的话，并不代表树的高度就是 lgn

Theorem: $E(\text{height of randomized built BST}) = O(\lg n)$ ↖"n个结点"

proof outline:

① prove Jensen's inequality $f[E(X)] \le E[f(X)]$ for convex function f "凹函数"

② instead of analyzing X_n = r.v. of height of BST on n nodes, "random variable"

analyze $Y_n = 2^{X_n}$

③ prove that $E(Y_n) = O(n^3)$

④ conclude that

$$\begin{cases} E(2^{X_n}) = E(Y_n) = O(n^3) \\ 2^{E(X_n)} \le E(2^{X_n}) \end{cases}$$

$$\Longrightarrow E(X_n) \le \lg O(n^3)$$
$$= 3\lg n + O(1)$$

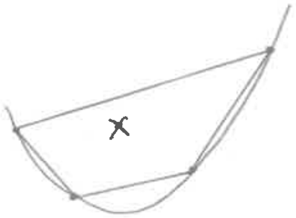
proof: ① $f: R \to R$ is convex if for all x, y and all $\alpha, \beta \geq 0$, $\alpha+\beta=1$,

$$f(\alpha x+\beta y) \leq \alpha f(x)+\beta f(y)$$

Lemma if $f: R \to R$ is convex,

$x_1, x_2, \cdots, x_n \in R$, $\alpha_1, \alpha_2, \cdots, \alpha_n \geq 0$ with $\alpha_1+\alpha_2+\cdots+\alpha_n=1$,

then $f(\sum_{k=1}^{n}\alpha_k x_k) \leq \sum_{k=1}^{n}\alpha_k f(x_k)$



proof: (induction)

base $n=1$, $f(x_1) \leq f(x_2)$

step

$$
\begin{aligned}
& f(\sum_{k=1}^{n}\alpha_k x_k) \\
&= f(\alpha_n x_n+\sum_{k=1}^{n-1}\alpha_k x_k) \\
&= f(\alpha_n x_n+(1-\alpha_n)\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}x_k) \\
&\leq \alpha_n f(x_n)+(1-\alpha_n)f(\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}x_k) && \leftarrow \text{"凹函数的定义"/n=2 case} \\
&\leq \alpha_n f(x_n)+(1-\alpha_n)\sum_{k=1}^{n-1}\frac{\alpha_k}{1-\alpha_n}f(x_k) && \leftarrow \text{induction hypothesis} \\
&= \alpha_n f(x_n)+\sum_{k=1}^{n-1}\alpha_k f(x_k) \\
&= \sum_{k=1}^{n}\alpha_k f(x_k) \quad \text{Q.E.D.}
\end{aligned}
$$

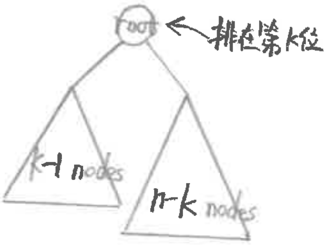
next, to prove Jensen's inequality, suppose x is an integer

$$
\begin{aligned}
f[E(X)] &= f(\sum_{x=-\infty}^{+\infty}x\cdot \underbrace{P(X=x)}_{\text{"sum to 1"}}) \\
&\leq \sum_{x=-\infty}^{+\infty}P(X=x)\cdot f(x) && \leftarrow \text{by Lemma} \\
&= E[f(x)]
\end{aligned}
$$

② expected BST height analysis

X_n = random variable of height of a randomly built BST on n nodes

$Y_n = 2^{X_n}$ ($y=2^x$ is a convex function)



if root r has rank k,

then $X_n = 1+\max\{X_{k-1}, X_{n-k}\}$

$Y_n = 2\max\{Y_{k-1}, Y_{n-k}\}$

define indicator random variables,

$$Z_{nk} = \begin{cases} 1, & \text{if the root has rank } k \\ 0, & \text{otherwise} \end{cases}$$

$$P(Z_{nk}=1)=E(Z_{nk})=\frac{1}{n}$$

$$Y_n = \sum_{k=1}^{n} Z_{nk} \cdot \left(2\max\{Y_{k-1}, Y_{n-k}\}\right)$$

$$E(Y_n) = E\left[\sum_{k=1}^{n} Z_{nk} \cdot \left(2\max\{Y_{k-1}, Y_{n-k}\}\right)\right]$$

$$= \sum_{k=1}^{n} E\left[Z_{nk} \cdot \left(2\max\{Y_{k-1}, Y_{n-k}\}\right)\right] \quad \longleftarrow \text{期望的线性性质}$$

$$= 2\sum_{k=1}^{n} \left[\underbrace{E(Z_{nk})}_{=\frac{1}{n}} \cdot E\left(\max\{Y_{k-1}, Y_{n-k}\}\right)\right] \quad \longleftarrow \text{事件的独立性}$$

$$= \frac{2}{n}\sum_{k=1}^{n} E\left(\max\{Y_{k-1}, Y_{n-k}\}\right)$$

$$\leq \frac{2}{n}\sum_{k=1}^{n} E\left(Y_{k-1} + Y_{n-k}\right)$$

$$= \frac{2}{n}\sum_{k=1}^{n} \left[E(Y_{k-1}) + E(Y_{n-k})\right] \quad \longleftarrow \text{期望的线性性质}$$

$$= 2 \times \frac{2}{n}\sum_{k=1}^{n} E(Y_{k-1})$$

$$= \frac{4}{n}\sum_{k=0}^{n-1} E(Y_k)$$

③ (substitution method to solve the recurrence)

claim: $E(Y_k) \leq cn^3$

proof: (substitution method = induction)

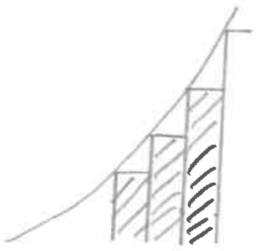
base $n = \Theta(1)$ true if c is sufficiently large

step $E(Y_n) \leq \frac{4}{n}\sum_{k=0}^{n-1} E(Y_k) \quad \leftarrow k<n$

$$\leq \frac{4}{n}\sum_{k=0}^{n-1} c \cdot k^3 \quad \text{(induction hypothesis)}$$

$$\leq \frac{4c}{n}\int_0^n x^3\, dx$$

$$= c \cdot n^3$$



so $E(Y_k) = O(n^3)$

④ 略

Lec10 平衡搜索树

balanced search tree: search tree data structure maintaining a dynamic set of n elements, using a tree of height O(lgn) (不一定是二叉)

Examples:

- AVL trees 1962
- 2-3 trees 1970
- 2-3-4 trees
- B-trees
- red-black trees
- skip lists
- treaps (树堆) 1996

red-black trees

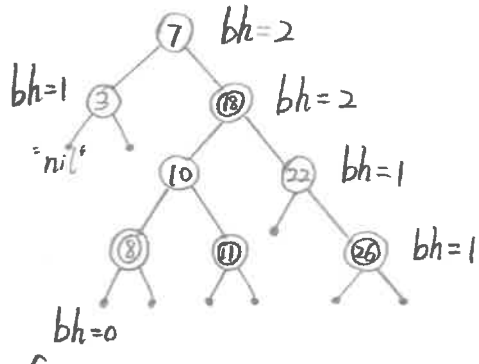
BST data structure with extra information in each node called the color field, and there are several properties that a tree with a color field has to satisfy in order to be called a red-black tree, 这些性质被称为红黑性(red-black properties)

red-black properties

1. every node is either $\underset{\circledcirc}{\underline{\text{red}}}$ or $\underset{\circ}{\underline{\text{black}}}$
2. the root and the leaves (nils) are all black
3. every red node has black parent
4. all simple path (不重复任何结点), from a node x to a descended leaf of x, have the same number of black nodes on them, #black-nodes = black-height(x)

does not count x itself

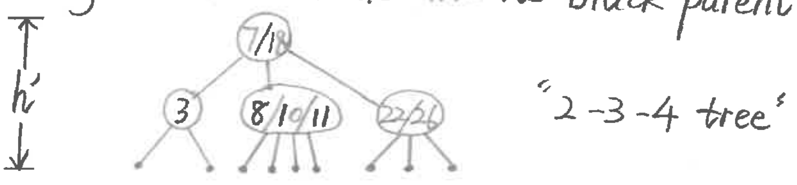
Example:



height of red-black tree

a red-black tree with n keys has height $h \leq 2\lg(n+1) = O(\lg n)$

proof sketch: merge each red node into its black parent



此时，所有的"nil"叶结点都有相同的深度（性质4），等于 black-height(root)。"balanced!"
每个内部结点、都有2到4个子结点。

proof:

$\# \text{leaves} = n+1$ (in either tree)

in 2-3-4 trees, $2^{h'} \le \# \text{leaves} \le 4^{h'}$

so $2^{h'} \le n+1$

$h' \le \log_2(n+1)$

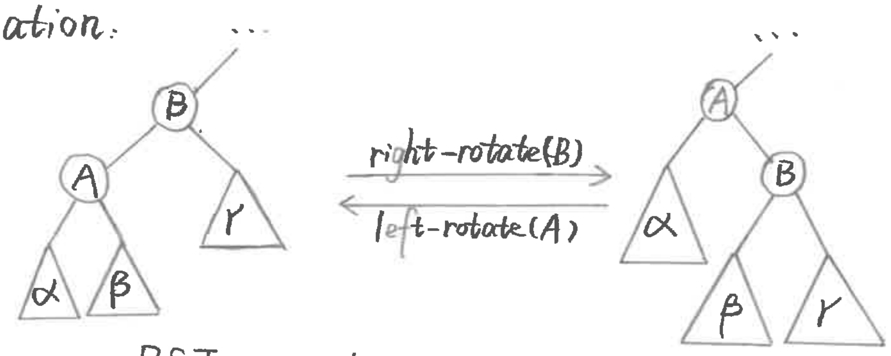
而性质3告诉我们，每个红结点只能连着黑结点，所以最多只能红黑相间，故一条路径上红结点数最多是总结点数的一半，而所有路径里最长的那条，就是树的高度。

$h \le 2h' \le 2\lg(n+1)$ Q.E.D.

Corollary:

- Queries (search, min, max, successor, predecessor) can in $O(\lg n)$ time in a red-black tree
- Updates (insert, delete) modify the tree
  - BST operation
  - color changes
  - restructuring of links via rotations, $O(1)$ time

Rotation:



preserves BST property

$\forall a \in \alpha, b \in \beta, c \in \gamma$, $a \le A \le b \le B \le c$ ✓

RB-Insert(T, x):

// idea: Tree-Insert(x), color the node as red ← 如果设为黑色，则会扰乱 black-height

// problem: parent might be red ⇒ violate property 3

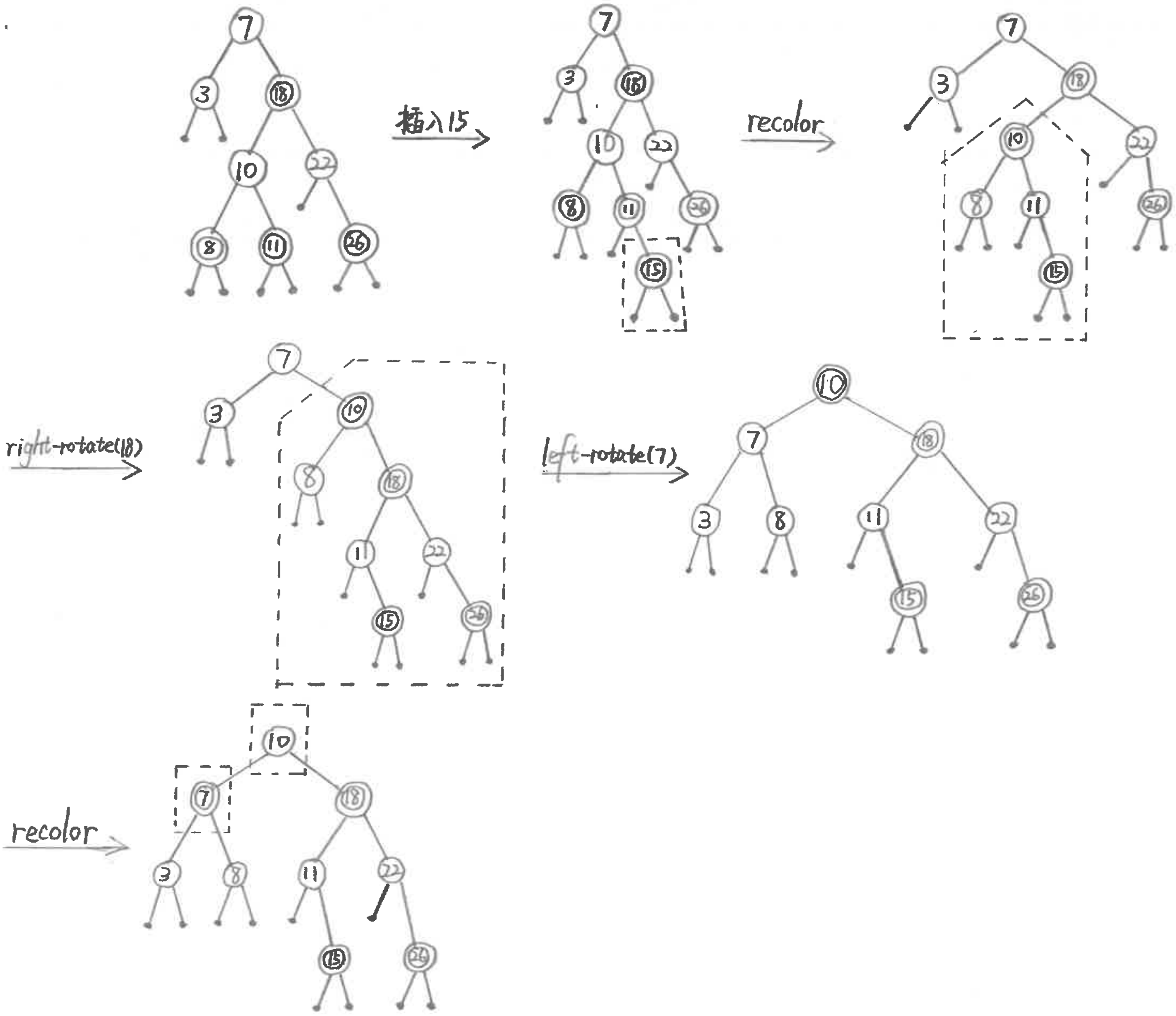
// so we need to move the violation of property 3 up the tree

// via recoloring or rotation until we can fix violation

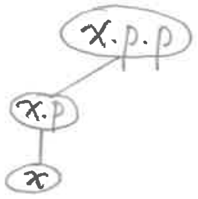
// 先看一个例子

// 伪代码附其后

// 目的：将x加入动态集中，同时维持并保留其红黑性

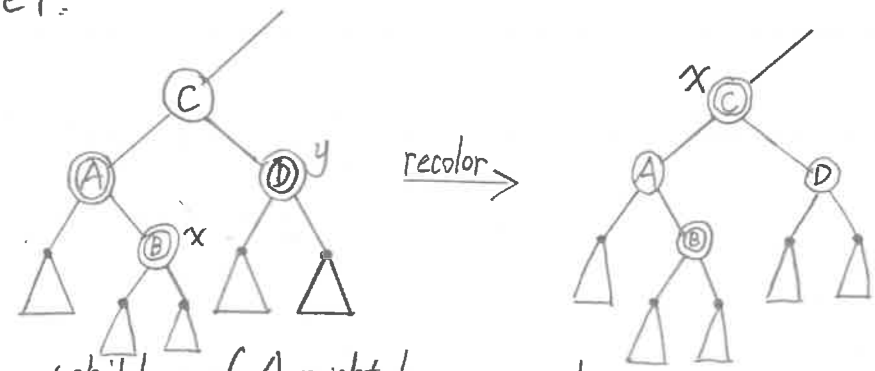


```
RB-Insert(T, x):
    Tree-Insert(T,x)
    x.color ← RED
    while x ≠ T.root and x.color = RED
        do if x.parent = x.parent.parent.left
            then y ← x.parent.parent.right
            if y.color = RED
                then <case1>
            else if x = x.parent.right
                then <case2>
                <case3>
        else
            <similar with if-condition, but reverse the notions of right and left>
    T.root.color ← Black
```



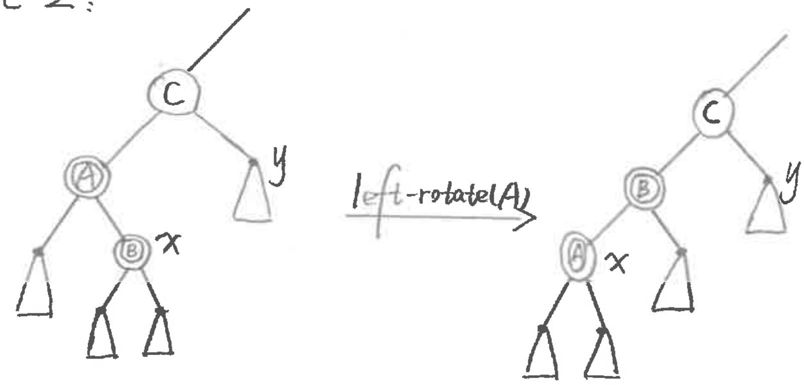
△ : tree with black root, all △ have same black height

case 1.

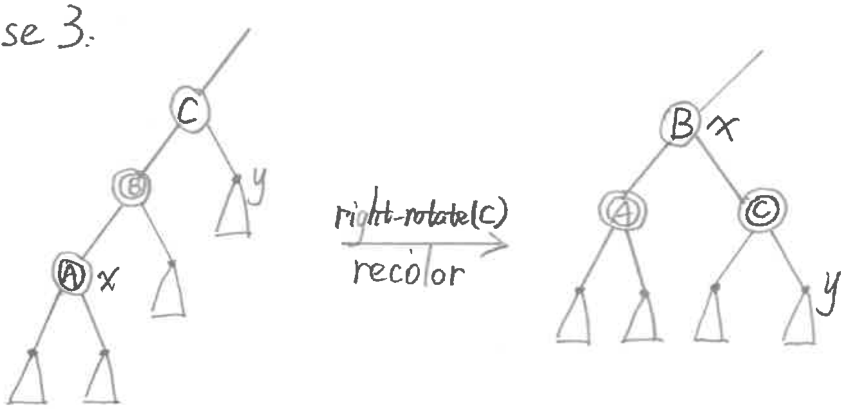


(children of A might be reversed)
recurrence on C

case 2.



case 3.



$T(n) = O(\lg n)$

Lec11扩充的数据结构、动态有序统计和区间树

augmenting data structures

normally, rather than designing data structures from scratch, you tend to take existing data structures and build your functionality into them

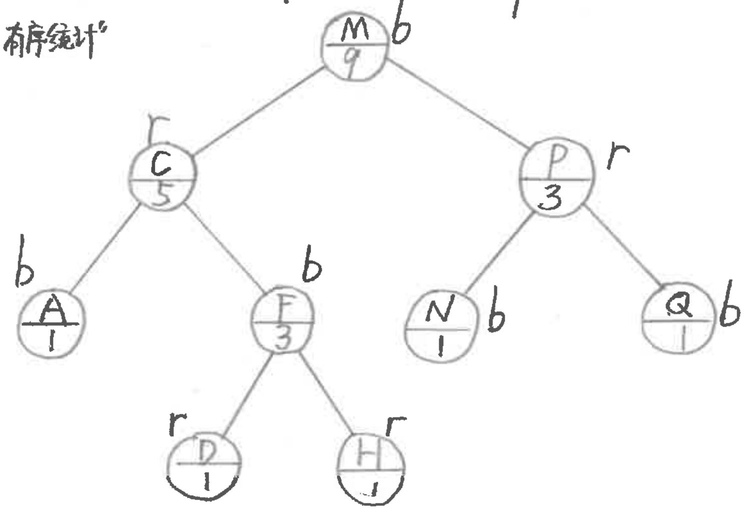
dynamic order statistics

OS-Select (i): return the i-th smallest item in dynamic set

OS-Rank (x): return the rank of x in the sorted order of dynamic set

the basic idea is to keep the sizes of subtrees in the nodes of a red-black tree

"order statistics 有序统计"



(record the subtree sizes in the red-black tree)

$x.size = x.left.size + x.right.size + 1$ (= the rank of x)

Trick: sentinel (标记法), 即 dummy record (伪记录) for nil (nil.size = 0)

根据这个,开始写 OS-Select (i) 的代码。

```
OS-Select (x, i) // the i-th smallest in the subtree rooted at x
    k ← x.left.size + 1    // k = rank(x)
    if i = k then return x
    if i < k then return OS-Select (x.left, i)
    else return OS-Select (x.right, i-k)
```

Question: why not just let nodes keep its ranks themselves?

Answer: 难以维护。比如插入一个最小的元素,所有的记录都要被修改。

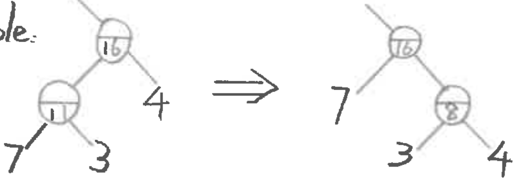
modifying ops: insert, delete

strategy: update subtree sizes when inserting and deleting, (O(lgn) time)

but must handle rebalancing

- r-b color changes: no effect on the size of subtrees
- rotations: look at children and fix up in O(1) time

Example:



data-structure augmentation

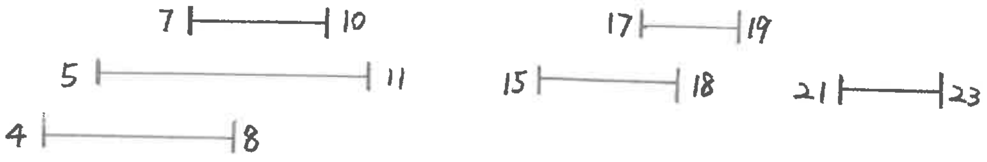
methodology: (Ex: OS-trees)

1. choose an underlying data structure (RB-tree)
2. determine what additional information we wish to maintain in the data-structure (subtree sizes)
3. verify that the information can be maintained for the modifying operations
4. develop new operations that use the information you stored (OS-Select, OS-Rank)

usually, must play with interactions between steps

Example: interval trees

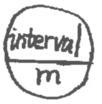
maintain a set of intervals, e.g. time intervals

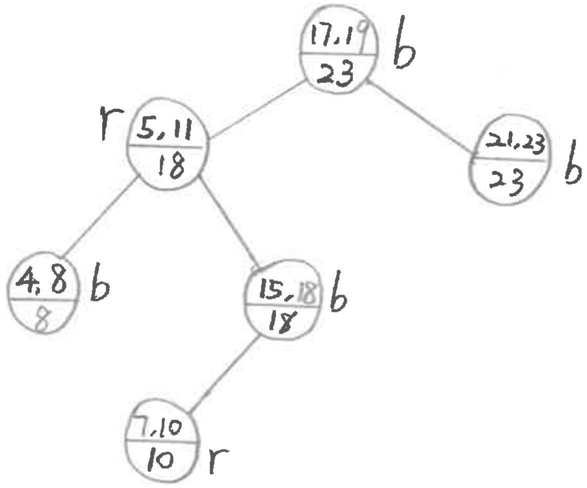


say $i=[7,10]$, $i.low=7$, $i.high=10$

目标 查询(Query)：给定一个区间，查询集合里所有与给定区间发生重合的区间有哪些。

1. 选择红黑树，选择 interval 的 lower endpoint 作为关键字
2. 在一个结点里存储这个结点的子树的最大值 m





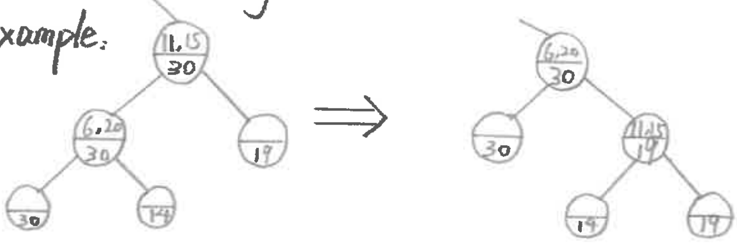
$m = \max\{x.high, x.left.m, x.right.m\}$

3. modifying operations

- insert (O(lgn) time)

fix m's on the way down, but need to handle rotations

Example:



- delete

  留作习题答案略

4. new operations

```
Interval-Search (i) // find an interval that overlaps i
    x ← root
    while x ≠ nil and (i.low > x.interval.high or x.interval.low > i.high)
        do //
            if x.left ≠ nil and i.low ≤ x.left.m
                then x ← x.left
            else
                then x ← x.right
    return x
```

Time = $O(\lg n)$

to list <u>all</u> overlaps,

"拿到一个就删一个，直到拿完，最后再放回去" $O(k\lg n)$

"输出敏感：时间与输出数量有关"

Correctness Analysis

Theorem: let $L=\{i' \in x.\text{left}\}$, $R=\{i' \in x.\text{right}\}$,

if search goes right, then $\{i' \in L, i' \text{ overlaps } i\} = \phi$

if search goes left, then $\{i' \in L, i' \text{ overlaps } i\} = \phi$

$\Rightarrow \{i' \in R, i' \text{ overlaps } i\} = \phi$

proof: suppose search goes right.

if $x.\text{left} = \text{nil}$, done since $L = \phi$

otherwise, $i.\text{low} > x.\text{left}.m$ $= j.\text{high}$ for some $j \in L$

no other intervals in L has a larger high endpoint than $j.\text{high}$

therefore, $\{i' \in L, i' \text{ overlaps } i\} = \phi$

suppose search goes left, and $\{i' \in L, i' \text{ overlaps } i\} = \phi$

then $i.\text{low} \leq x.\text{left}.m = j.\text{high}$ for some $j \in L$

since $j \in L$ and j doesn't overlap i

$\Rightarrow i.\text{high} < j.\text{low}$

$\because \forall i' \in R$, $j.\text{low} \leq i'.\text{low}$

$\therefore \{i' \in R, i' \text{ overlaps } i\} = \phi$

Q.E.D.

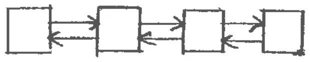
Lec12 跳跃表

skip lists: (Pugh in 1989)

a new balanced search structure, a data structure that maintains a dynamic set, supporting insertion, deletion and search

starting from scratch,

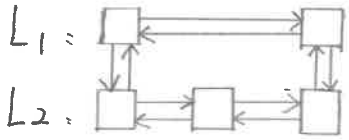
a sorted linked list



search takes $\Theta(n)$ time

how can I make it better?

two sorted linked lists, links between equal keys in L_1 and L_2



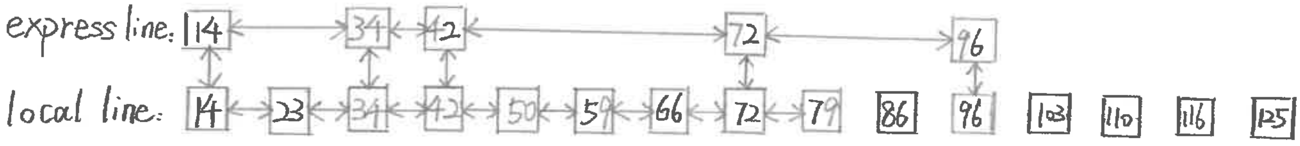
L_1 stores some subsets

L_2 stores all the elements

Example. (纽约的第七大道线、快线 express lines)

14 23 34 42 50 59 66 72 79 86 96 103 110 116 125

"express and local lines"
站站停车



Search (x)

- walk right in top list L_1 until going right would go to far
- walk down to L_2
- walk right in L_2 until find x or an element $> x$

what keys go in L_1?

best is to spread them out uniformly

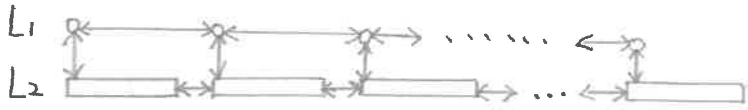
$\Rightarrow$ cost of search $\approx |L_1| + \frac{|L_2|}{|L_1|}$

$\because |L_2|$ 是一个常数 n

$\therefore \min |L_1| + \frac{n}{|L_1|}$

$\Rightarrow |L_1| = \sqrt{n}$

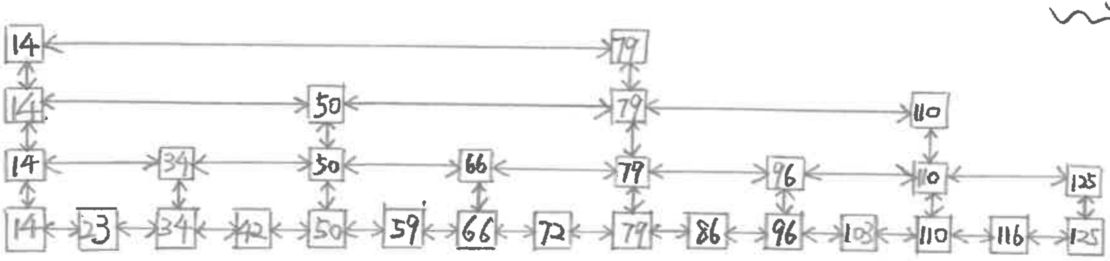
so search cost $\approx 2\sqrt{n}$



3 sorted linked list will cost $3\sqrt[3]{n}$ time for search

k ⋯⋯⋯ $k\sqrt[k]{n}$ ⋯⋯

$\log_2 n$ ⋯⋯⋯ $\log_2 n \cdot \sqrt[\log_2 n]{n} = \log_2 n \cdot n^{\frac{1}{\log_2 n}} = 2\log_2 n$ ⋯⋯ ☆



诀窍：设 $r = \frac{|L_m|}{|L_{m-1}|}$，共有 x 个 sorted linked list，则 $r^x = n$

若 $x = \log_2 n$，则 $r^{\log_2 n} = n \Rightarrow r = 2$

"like a tree!" "形式上不是树，但逻辑上有点类似"

skip lists maintainance roughly subjects to insert and delete

Insert (x)

- Search (x) to find where x fits in the bottom list
- insert x into the bottom list
- which other lists should store x ? (if there are $\log_2 n$ sorted linked lists)

flip a coin, if heads, then promote x to the next level up and flip again

每次都有50%的提升概率

Delete (x)

找到这个元素，把它从出现的链表中一路删除上去。

Theorem:

with high probability, every search in n elements skip lists costs $O(\lg n)$

"w.h.p."

在有$\log_2 n$级链表的情况下

*define event E occurs w.h.p. if for any $\alpha \geq 1$, there is a suitable choice of constants for which the event E occurs with probability $\geq 1 - O(\frac{1}{n^\alpha})$

失败概率

这里只给出一个引理，其余的证明过程略。

Lemma: w.h.p., #levels $= O(\lg n)$

proof: error probability for $\{\leq c\lg n \text{ levels}\}$

$= P\{> c\lg n \text{ levels}\}$

Boole inequality → $\leq n \cdot P\{x \text{ gets promoted} \geq c\lg n \text{ times}\}$

$= n \cdot (\frac{1}{2})^{c\lg n}$

$= n / n^c$

$= 1/n^{c-1} \underset{}{\overset{令c-1=\alpha}{=\!=}} 1/n^\alpha$ Q.E.D.

证明思路：(search backwards)
search starts [ends] at a node in the bottom list, at each node visited,
if the node wasn't promoted higher,
(抛的硬币抛出了反面), then go left,
if the node was promoted, then go up.
and stop [root] (or $-\infty$).

Lec 13 平摊分析、表的扩增、势能方法

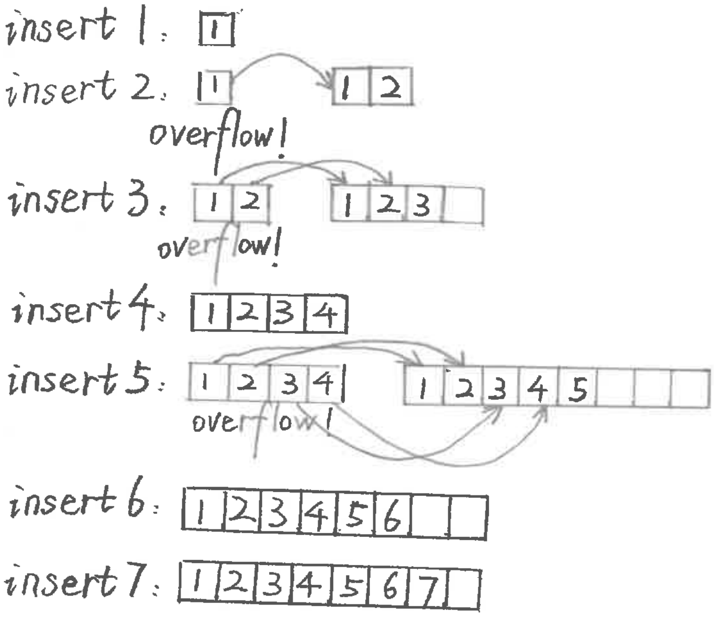
Amortized Analysis

analyze a sequence of operations to show that the average cost per operation is small, even though one or several operation(s) maybe expensive (no probability here, it's average performance in the worst case)

Example: dynamic tables

idea is that whenever the table gets too full (overflows), "grow" it

1. allocate a larger table
2. move the items from the old table to the new
3. free the old table



Analysis:

sequence of n insert operations,

the worst-case cost of 1 insert is $\theta(n)$,

but the worst-case cost of n inserts is NOT $n \cdot \theta(n) = \theta(n^2)$!

因为不是所有的项都是最坏情况。

n inserts take $\theta(n)$ time.

let c_i be the cost of the i-th insert, then $c_i = \begin{cases} i, & \text{if } i-1 = 2^x \\ 1, & \text{otherwise} \end{cases}$

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

so cost of n inserts $= \sum_{i=1}^{n} c_i = n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j \leq 3n = \theta(n)$

thus, average cost per insert is $\frac{\theta(n)}{n} = \theta(1)$

"尽管有时候要付出比较大的代价，但这个巨大的开销会被之前的操作平摊掉"

three types of amortized arguments

1. aggregate analysis (just saw，基本上就是要你分析 n 次操作一共花费多少时间)
2. accounting } more precise, because they allocate specific amortized costs to each operatio
3. potential }

Accounting Method

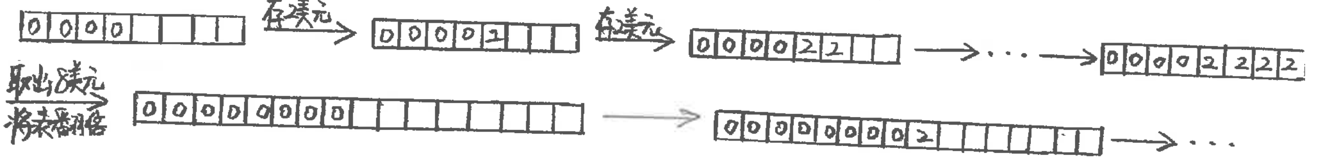
- charge the i-th operation a fictitous amortized cost $\hat{c_i}$ (虚构的)
  (\$1 pays for 1 unit of work)
- fee is consumed to perform the operation
- unused amount is stored in the bank for use by later operations
- bank balance must not go negative (不能借款)

must have $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c_i}$, $\forall n$

true cost

Example: dynamic table

- charge $\hat{c_i} = \$3$ for the i-th insert,
  \$1 pays for an immediate insert, so \$2 gets stored (预存，用于将表翻倍)
- when the table doubles.
  \$1 to move a rescent item. \$1 to move an old item



"我总是能偿付所有表扩增的开销"

$$\underset{3n}{\underset{\parallel}{\sum_{i=1}^{n} \hat{c_i}}} \geq \sum_{i=1}^{n} c_i$$

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
$\hat{c_i}$	2	3	3	3	3	3	3	3	3	3
$bank_i$	1	2	2	4	2	4	6	8	2	4

收了3美元，花了4美元用于复制旧项，又花了1美元插入新项，4+3−4−1=2

这里也可以收3美元，这样之后都会多出1美元

注：也可以每次收4美元，5、6、7、…都是可行的。
　　但不能每次收2美元，这样余额会变为负数。

Potential Method

"what do you aspire, to be a bookkeeper or to be a physicist?"

"bank account" viewed as potential energy of dynamic set

framework:

- start with data structure D_0
- operation i transforms D_{i-1} into D_i
- cost of operation i is C_i
- define the potential function $\phi: \{D_i\} \rightarrow R$ such that
  $\phi(D_0) = 0$ and $\phi(D_i) \geq 0 \ \forall i$
- define amortized cost $\hat{C_i}$, $\hat{C_i} = C_i + \underbrace{\phi(D_i) - \phi(D_{i-1})}_{\text{change in potential}}$. $\Delta\phi_i$

if $\Delta\phi_i > 0$, then $\hat{C_i} > C_i$.

// I charged more than it costs me to do the operation

operation i stores work in the data structure for later

if $\Delta\phi_i < 0$, then $\hat{C_i} < C_i$

data structure delivers up stored work to help pay for operation i

从势能的角度与从记帐法的角度比較,用记帐方法来看,会先决定一个平摊代价,然后再分析一下银行存款,确保它不为负值。在某种程度上,在势能法里,会说"我的存款是这样子的"然后再分析一下哪个平摊代价才合适。

total amortized cost of n operations is

$$\sum_{i=1}^{n}\hat{C_i} = \sum_{i=1}^{n}\left(C_i + \phi(D_i) - \phi(D_{i-1})\right)$$

$$= \sum_{i=1}^{n} C_i + \underbrace{\phi(D_n)}_{\geq 0} - \underbrace{\phi(D_0)}_{=0}$$

$$\geq \sum_{i=1}^{n} C_i \qquad \text{(telescope, 缩近)}$$

Example: table doubling

define $\phi(D_i) = 2i - 2^{\lceil \log_2 i \rceil}$, assume $\phi(D_0) = 0$, note $\phi(D_i) \geq 0 \ \forall i$

0	0	0	0	0	0		

$\phi(D_6) = 2 \times 6 - 2^{\lceil \log_2 6 \rceil} = 4$

amortized cost of the i-th insert

$$\hat{C_i} = C_i + \phi(D_i) - \phi(D_{i-1})$$

$$= \begin{cases} i, & \text{if } i-1 = 2^x \\ 1, & \text{otherwise} \end{cases} + 2i - 2^{\lceil \log_2 i \rceil} - 2(i-1) + 2^{\lceil \log_2 i-i}$$

$$= \begin{cases} i, & \text{if } i-1 = 2^x \\ 1, & \text{otherwise} \end{cases} + 2 - 2^{\lceil \log_2 i \rceil} + 2^{\lceil \log_2 (i-1) \rceil}$$

$$= \begin{cases} i + 2 - 2^{\lceil \log_2 i \rceil} + 2^{\lceil \log_2 (i-1) \rceil}, & \text{if } i-1 = 2^x \\ 1 + 2 - 2^{\lceil \log_2 i \rceil} + 2^{\lceil \log_2 (i-1) \rceil}, & \text{otherwise} \end{cases}$$

$$= \begin{cases} i + 2 - 2(i-1) + (i-1), & \text{if } i-1 = 2^x \\ 1 + 2, & \text{otherwise} \end{cases}$$

$$= \begin{cases} 3, & \text{if } i-1 = 2^x \\ 3, & \text{otherwise} \end{cases}$$

$$= 3$$

therefore, the amortized cost is 3, for each insert

so, n inserts costs $\Theta(n)$ in the worst case

平摊分析的结论.

① amortized costs provide a clean abstraction for data structure performance
简洁的 抽象

只要你不关注实时表现，只关注聚集行为，这将是一个相当好的性能抽象概念。

即使有的时候代价会很大，但也会被平摊掉.

② any method can be used, but each has situations where it is arguably simplest or most precise

③ different potential functions, or accounting costs, may yield different bounds
产生

Lec14 竞争性分析、自组织表

self-organizing lists

list L of n elements

- operation Access(x) costs rank(x) = distance of x from the head of the list
- L can be reordered by transposing adjacent elements, the cost is 1

Example:

$$L \rightarrow 12 \rightarrow 3 \rightarrow 50 \rightarrow 14 \rightarrow 17 \rightarrow 4$$

Access(14). cost = 4

Transpose(3, 50). cost = 1

Def

a sequence S of operations is provided one at a time,
for each operation, an online algorithm must execute the operation immediately.
offline algorithm may see all of S in advance

Goal:

to minimize the total cost $C_A(S)$

worst-case analysis

adversary always accesses tail element of L. $C_A(S) = \Omega(|S| \cdot n)$ if online

average-case analysis

suppose element x is accessed with probability $p(x)$,
$E[C_A(S)] = \sum_{x \in L} p(x) \cdot rank(x)$,
which is minimized when L is sorted in decreasing order with respect to p

Heuristic

keep count of the number of times each element is accessed,
and maintain the list in order of decreasing count

Practice

"move-to-front" heuristic
after accessing x. move x to head of list using transposes, cost = 2 × rank(x
respond well to locality in S

Competitive Analysis

Def: an on-line algorithm A is α-competitive if there exists a constant k, such that for any sequence S of operations, the cost of S using algorithm A

$C_A(S) \le \alpha \cdot \underbrace{C_{opt}}_{\text{"optimal off-line algorithm"}}(S) + k$

Theorem: Move-To-Front is 4-competitive for self-organizing lists

proof:

let L_i be MTF's list after the i-th access

let L_i^* be OPT's list after the i-th access

let c_i be MTF's cost for the i-th operation, equal $2 \times \text{rank}_{L_{i-1}}(x)$

let c_i^* be OPT's cost for the i-th operation, equal $\text{rank}_{L_{i-1}^*}(x) + \underbrace{t_i}_{t_i\text{次置换}}$

define the potential function $\phi : \{L_i\} \to R$ by:

$$\phi(L_i) = 2 \cdot \underbrace{|\{(x,y) : x <_{L_i} y \text{ and } y <_{L_i^*} x\}|}_{\text{"MTF表与OPT表的不同点"}}$$

$$= 2 \cdot \# \text{ inversions}$$

note: $\phi(L_i) \ge 0$, $\forall i$

$\phi(L_0) = 0$ if MTF and OPT start with same list

how much does ϕ change from one transpose?

a transpose creates or destroys one inversion

so $\Delta\phi = \pm 2$

以下证明略(看不懂)

Lec15 动态规划、最长公共子序列

Dynamic Programming

"any tabular method for accomplishing something"

a design technique, like Divide&Conquer, a way of solving a class of problems rather than a particular algorithm or something

Example:

Longest Common Subsequence Problem (LCS)

given two sequences $x[1...m]$ and $y[1...n]$,
to find a longest sequence which is common to both

x: A B C B D A B

y: B D C A B A

LCS's: BDAB、BCAB、BCBA

brute-force algorithm:

to check every subsequence of $x[1...m]$,
to see if it is also a subsequence of $y[1...n]$

there are 2^m subsequences of x, and each check takes $O(n)$,
so running time is $O(n \cdot 2^m)$

consider the more simple problem!

Simplification:

1. focus on the length of the longest common subsequence of x and y,
2. extend the algorithm to find the longest common subsequence itself

Notation: $|s|$ denotes the length of sequence s

Strategy: consider prefixes of x and y

Define $c[i,j] := |LCS(x[1...i], y[1...j])|$

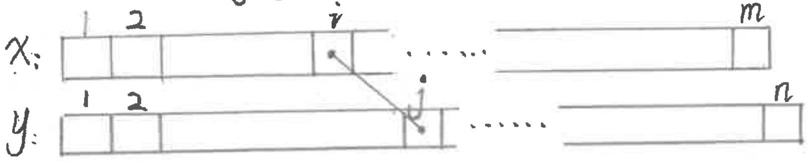
then $c[m,n] = |LCS(x,y)|$

Theorem:

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{, if } x[i]=y[j] \\ \max\{c[i-1,j], c[i,j-1]\} & \text{, otherwise} \end{cases}$$

Proof:

case $x[i] = y[j]$



let $z[1\ldots k] = LCS(x[1\ldots i], y[1\ldots j])$, where $c[i,j] = k$

then $z[k] = x[i]$ $(= y[j])$, or else z could be extended by appending on $x[i]$

thus $z[k-1]$ is a common sequence of $x[1\ldots i-1]$ and $y[1\ldots j-1]$

claim: $z[1\ldots k-1] = LCS(x[1\ldots i-1], y[1\ldots j-1])$

suppose w is a longer common sequence. that is $|w| > k-1$

cut & paste: $w + z[k]$ is a common sequence of $x[1\ldots i]$ and $y[1\ldots j]$, with its length $> k$

contradiction!

thus, $c[i-1, j-1] = k-1$, which implies that $c[i,j] = c[i-1, j-1] + 1$

case $x[i] \neq y[j]$

<similar>

Q.E.D.

Dynamic Programming Hallmarks #1
“特征”

optimal substructure

an optimal solution to a problem (instance) contains optional solutions to subproblems

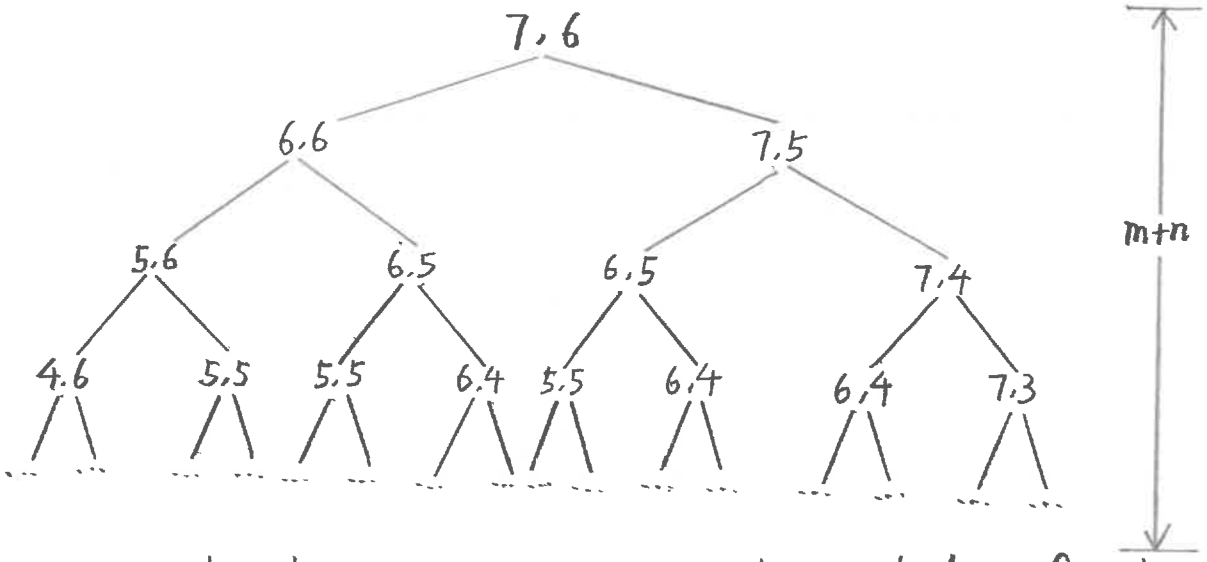
in LCS example, if $z = LCS(x, y)$, then any prefix of z is a longest common subsequence of a prefix of x and a prefix of y

Recursive Algorithm for LCS:

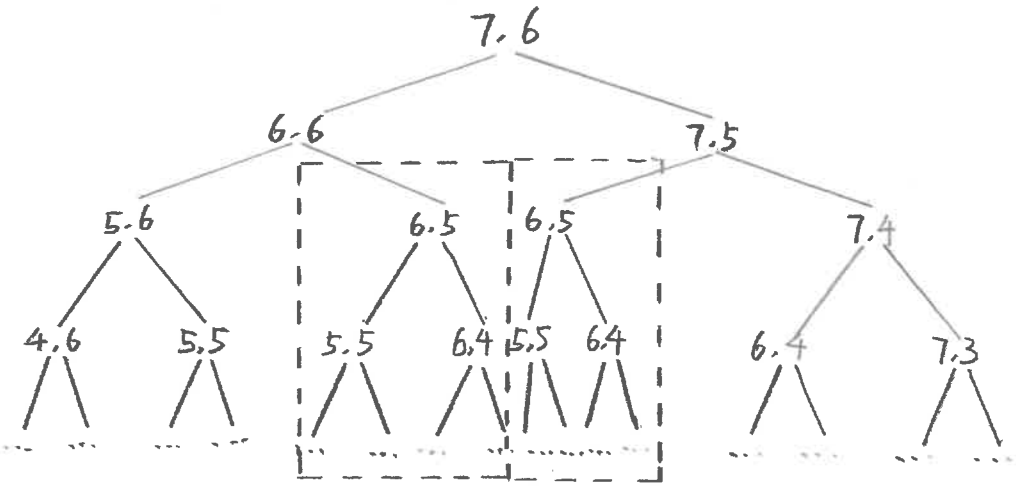
```
LCS(x, y, i, j): // ignoring base cases
    if x[i] = y[j]
        then c[i,j] <- LCS(x, y, i-1, j-1) + 1
    else
        c[i,j] <- max{LCS(x, y, i-1, j), LCS(x, y, i, j-1)}
    return c[i,j]
```

worst-case: $\forall i, j$, $x[i] \neq y[j]$

Recursion Tree (m=7 , n=6)



height = m+n implies the work is exponential , slow as the brute-force !



same subtree

⇒ same subproblem

⇒ repeated work

Dynamic Programming Hallmarks #2

overlapping subproblems

a recursive solution contains a "small" number of distinct subproblems repeated many times

与指数级比起来

in LCS example , subproblem space contains m×n distinct subproblems

so here is an improved algorithm !

it's an algorithm called " memo-ization algorithm "

"备忘录" , 不是 memorization

Memo-ization Algorithm

```
LCS(x, y, i, j):
    if c[i, j] = nil
        then if x[i] = y[j]
            then c[i, j] ← LCS(x, y, i-1, j-1) + 1
        else
            c[i, j] ← max{LCS(x, y, i-1, j), LCS(x, y, i, j-1)}
    return c[i, j]
```

time = $\Theta(mn)$

space = $\Theta(mn)$

there is another strategy for doing exactly the same calculation in a bottom-up way (真·动态规划)

the idea is to compute the table bottom-up

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0							
D	0							
C	0							
A	0							
B	0							
A	0							

→

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0							
C	0							
A	0							
B	0							
A	0							

填第一行

→

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0							
A	0							
B	0							
A	0							

填第二行

→

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0							
B	0							
A	0							

填第三行

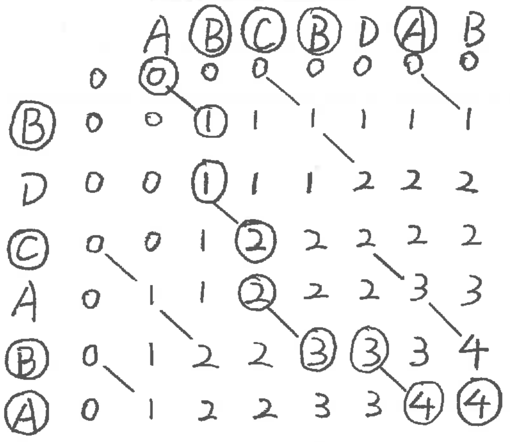
→ ... →

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	④

answer

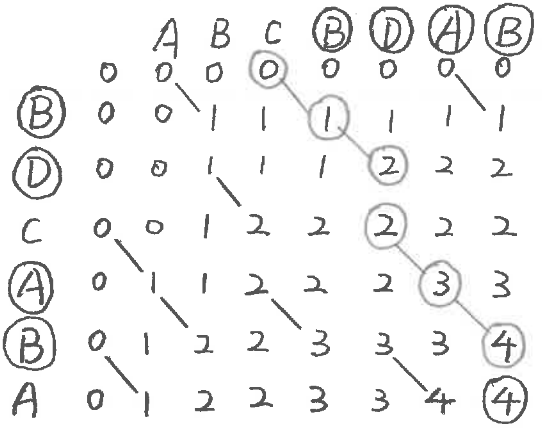
time = $\Theta(mn)$

I can reconstruct the longest common subsequence by tracing backwards



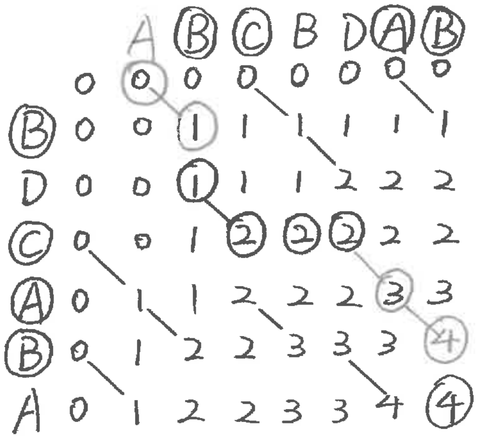
		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

"BCBA"



		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

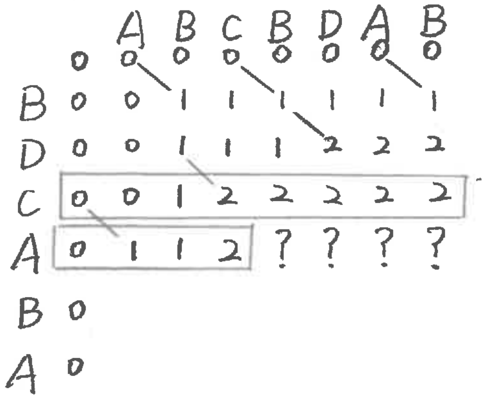
"BDAB"



		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

"BCAB"

space = $\Theta(mn)$,
actually, we can do $\Theta(\min(m,n))$
实际上，需要保留的只有一行。



		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	?	?	?	?
B	0							
A	0							

只有这些会影响？的计算，
之前的行都不参与计算

除了一行一行地填值，也可以一列一列地填值，这取决于 m 与 n 谁更小（省空间）

Lec16 贪婪算法、最小生成树

Graphs (review)

Digraph $G=(V,E)$

- set V of vertices (singular: vertex)
- set $E \subseteq V \times V$ of edges

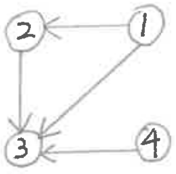
Undirected Graph: E contains unordered pairs

$|E| = O(|V|^2)$

if G is connected (连通的), $|E| > |V|-1$

Graph Representations

- adjacency matrix of $G=(V,E)$, where $V=\{1,2,\cdots,n\}$, is the $n \times n$ matrix A given by $A[i,j]=\begin{cases} 1 & ,\text{if } (i,j)\in E \\ 0 & ,\text{if } (i,j)\notin E \end{cases}$



	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

$\Theta(|V|^2)$ storage $\Longrightarrow$ dense representation

- adjacency list of $v \in V$ is the list $Adj[v]$ of vertices adjacent to v

$Adj[1] = \{2,3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\ \}$
$Adj[4] = \{3\}$

sparse!

$|Adj[v]| = \begin{cases} \text{degree}(v) & ,\text{ undirected} \\ \text{out-degree}(v) & ,\text{ directed} \end{cases}$

Handshaking Lemma (undirected graph)

$$\sum_{v \in V} \text{degree}(v) = 2|E|$$

结点也占据存储

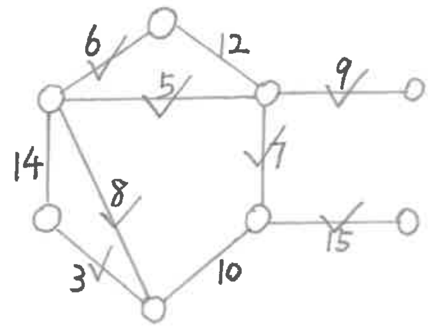
for undirected graphs $\Longrightarrow$ adjancy list representation uses $\Theta(|V|+|E|)$ storage, it is basically the same thing asymptotically for digraphs

Minimum Spanning Trees

input: connected undirected graph $G=(V,E)$ with weight function $w: E \to R$
for simplicity, assume all edge weights are distinct

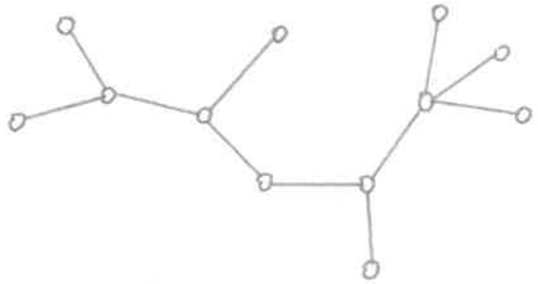
output: a spanning tree T (connects all the vertices) at minimum weight
$$W(T) = \sum_{(u,v)\in T} w(u,v)$$

Example:

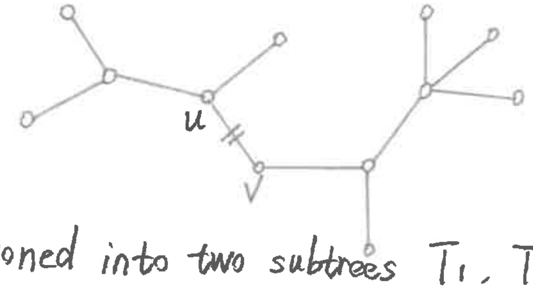


Optional Substructure

MST T:
(other edges in the graph will not be shown)



remove an arbitrary edge (u,v) in the MST T



then T is partitioned into two subtrees T_1, T_2

Theorem:

T_1 Is MST for $G_1 = (V_1, E_1)$,
G_1 is a subgraph of G induced by the vertices in T_1, i.e
V_1 = vertices in T_1, $E_1 = \{(x,y) \in E, x,y \in V_1\}$

similarly for T_2

proof: Cut & Paste

$w(T) = w(u,v) + w(T_1) + w(T_2)$

if T_1' is better than T_1 for G_1,

then $T' = \{(u,v)\} \cup T_1' \cup T_2$ would be better than T for G

conflication!

Overlapping Subproblems ?

YES!

Can we use Dynamic Programming?

YES!

but it turns out that MST exhibits a powerful property that, we call, the hallmark for greedy algorithms

Hallmark for Greedy Algorithms (这是可应用某算法的证据)

greedy-choice property:

a locally optimal choice is globally optimal

Theorem:

let T be MST of $G=(V,E)$, and let $A \subseteq V$

suppose $(u,v) \in E$ is the least weight edge connecting A to $V-A$

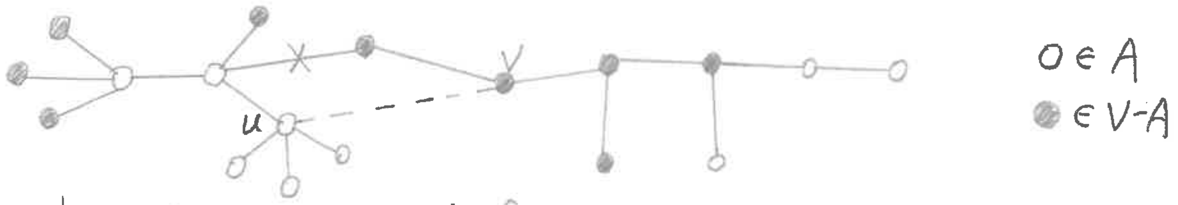
then, $(u,v) \in T$

Proof:

suppose $(u,v) \notin T$

Cut & Paste

T:



consider unique simple path from u to v in T,

swap (u,v) with the first edge on this path that connects a vertex in A to a vertex in $V-A$

图中的'x'边

a lower weight spanning tree than T, resulting a contradiction

Prim's Algorithm

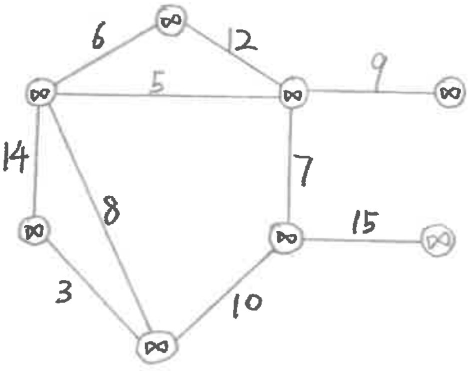
idea: to maintain V-A as a priority queue Q,
to key each vertex in Q with the weight of the least-weight edge
(vt.赋值)
connecting it to a vertex in A

pseudo-code:

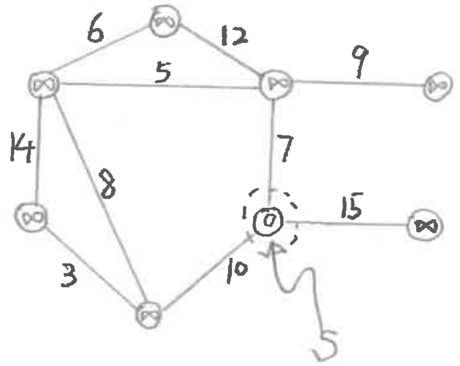
```
Q ← V    (A ← ∅)
v.key ← ∞ , ∀v ∈ V
s.key ← 0  for arbitrary s ∈ V

while Q ≠ ∅
      do u ← Extract-Min(Q)
      for each v ∈ Adj[u]
            do if v ∈ Q and w(u,v) < v.key
                  then v.key ← w(u,v)
                  T[v] ← u    (隐含了一个降低键值的操作)
```

at the end, $\{(v, \pi[v])\}$ forms MST

Example:

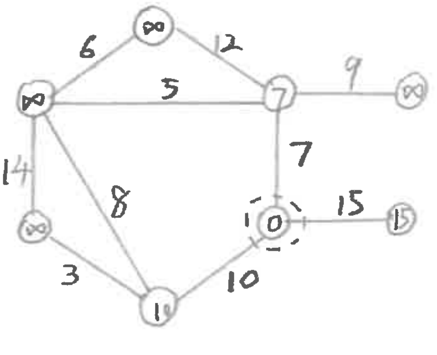


select S →

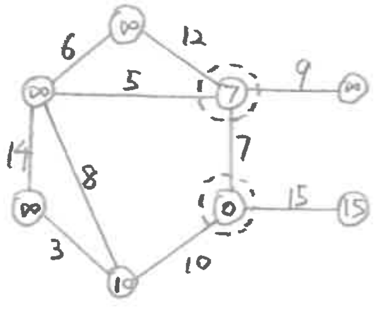


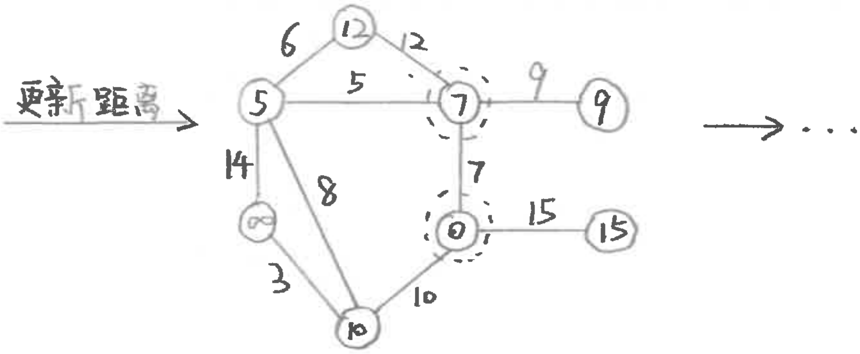
(⋯)表示已加入A

更新距离 →



取出"最便宜"的一个 →





Analysis

handshaking $\Rightarrow \Theta(E)$ implicit decrease keys

time = $\Theta(|V| \cdot T_{Exact\text{-}Min} + |E| \cdot T_{Decrease\text{-}Keys})$

$$= \begin{cases} O(|V|^2) , & \text{array} \leftarrow \text{dense 用这个} \\ O(|E|\lg|V|). & \text{binaryheap} \leftarrow \text{sparse 用这个} \end{cases}$$

Lec 17 shortest paths I：Dijkstra算法、广度优先搜索

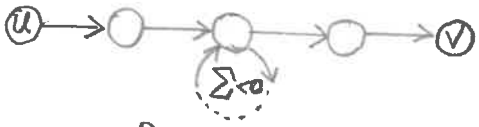
paths

- consider digraph $G=(V,E)$ with edge weights given by function $w: E\to R$
- path $p = v_1 \to v_2 \to \cdots \to v_k$ has weight $w(p)=\sum_{i=1}^{k-1} w(v_i, v_{i+1})$

shortest path from u to v is a path of minimum weight from u to v

shortest-path weight from u to v is denoted by $\delta(u,v) = \min\{w(p), p: \text{from } u \text{ to } v\}$

negative edge weights ⟹ some shortest paths may not exist



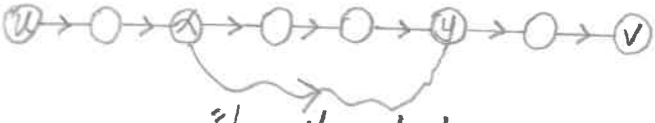
$\delta(u,v) = -\infty$

if no path from u to v, $\delta(u,v) = +\infty$

Optimal Substructure

a subpath of a shortest path is a shortest path

proof: (Cut & Paste)



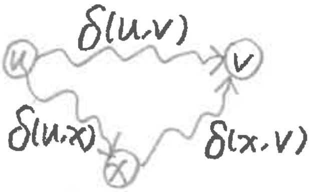
"hypothetical shorter path"

留作习题答案略，读者自证不难。

Triangle Inequality

for all vertices $u, v, x \in V$, $\delta(u,v) \le \delta(u,x) + \delta(x,v)$

proof:



留作习题答案略，读者自证不难。

single-source shorstest paths problem

from given source vertex $s\in V$, to find shortest-path weights $\delta(s,v)$ for all $v\in V$

today: assume $w(u,v)\ge 0$, $\forall u,v \in V$ ⟹ shortest paths exist if paths exist

⟹ $\delta(u,v) > -\infty$

idea: greedy

① maintain set S of vertices whose shortest-path distance from s is known ($s \in S$)

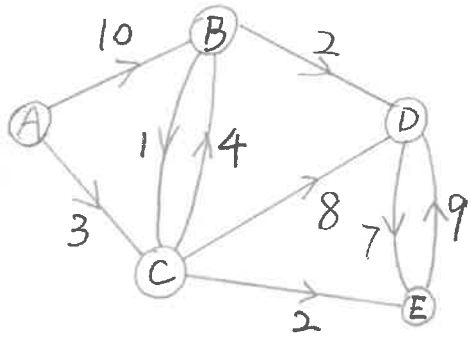
② at each step. add to S the vertex $v \in V-S$, whose estimated distance from s is minimum

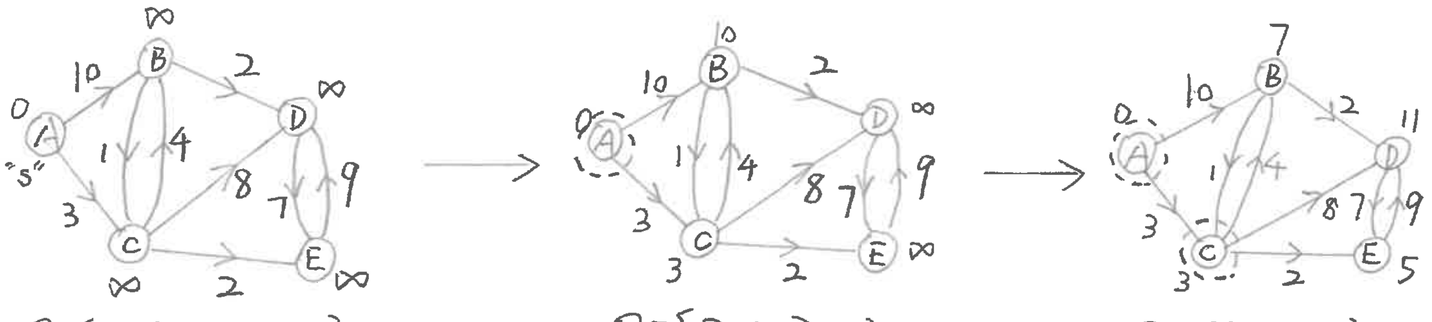
③ update distance estimates of vertices that are adjacent to v

Dijkstra's Algorithm

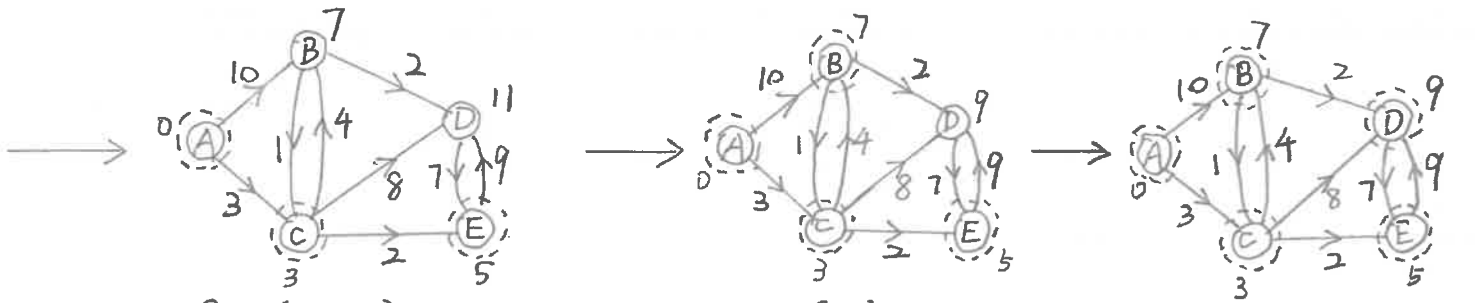
```
$s.distance \leftarrow 0$          // x.distance is the estimated distance from s to x
                                   $= \delta(s,x)$ when add x to S
for each $v \in V-\{s\}$
    do $v.distance \leftarrow \infty$
$S \leftarrow \phi$
$Q \leftarrow V$     // priority queue, keyed on distance

while $Q \neq \phi$
    do $u \leftarrow$ Extract-Min(Q)
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$
        do if $v.distance > u.distance + w(u,v)$
            then $v.distance \leftarrow \underbrace{u.distance}_{=\delta(s,u)} + w(u,v)$     } relaxation step
```

Example:





$Q = \{A, B, C, D, E\}$ $Q = \{B, C, D, E\}$ $Q = \{B, D, E\}$



$Q = \{B, D\}$ $Q = \{D\}$ $Q = \emptyset$

shortest-path tree

for each vertex v
  consider last edge (u,v) relaxed

Correctness I

invariant: $v.distance \geq \delta(s,v)$ for all $v \in V$

holds after initialization, and any sequence of relaxation steps

proof: (induction)

initially, $s.distance = 0$ and $v.distance = \infty \ \forall v \in V - \{s\}$

$\delta(s,s) = 0$ and $\delta(s,v) \leq \infty$

suppose for contradiction that invariant is violated

consider first violation $v.distance < \delta(s,v)$ is caused by

relaxation $v.distance \leftarrow u.distance + w(u,v)$

so $u.distance + w(u,v) < \delta(s,v)$

but $u.distance + w(u,v) \geq \delta(s,u) + w(u,v) \geq \delta(s,u) + \delta(u,v) \geq \delta(s,v)$

contradiction! Q.E.D.

Correctness Lemma

suppose $s \rightarrow \cdots \rightarrow u \rightarrow v$ is a shortest path from s to v,

if $u.distance = \delta(s,u)$ and we relax that edge (u,v),

then $v.distance = \delta(s,v)$ after relaxation

proof

$\delta(s,v) = w(s \rightarrow \cdots \rightarrow u) + w(u,v) = \delta(s,u) + w(u,v)$

Correctness I $\Rightarrow v.distance \geq \delta(s,v)$

either $v.distance = \delta(s,v)$ before relaxation $\Rightarrow$ done

or $v.distance > \delta(s,v) = u.distance + w(u,v)$ before relaxation

$\Rightarrow$ we relax and set $v.distance \leftarrow u.distance + w(u,v) = \delta(s,v)$

Q.E.D.

Correctness II

when Dijkstra terminates, $v.distance = \delta(s,v)$ for $\forall v \in V$

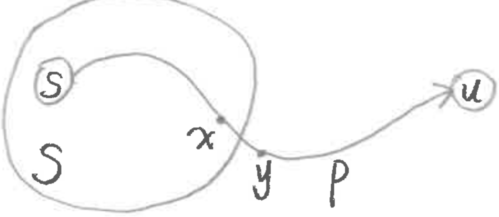
proof:

v.distance doesn't change once v is added to S

$\Rightarrow$ it suffices to prove that $v.distance = \delta(s,v)$ when v is added to S

suppose for contradiction that u is the first vertex (about to be) added to S, for which $u.distance \neq \delta(s,u) \Rightarrow u.distance > \delta(s,u)$

let p be the shortest path from s to u $\Rightarrow w(p) = \delta(s,u)$



consider first edge (x,y) where p exits S

$\because$ first violation

$\therefore$ $x.distance = \delta(s,x)$

when we add x to S, we relax (x,y)

by Correctness Lemma, $y.distance = \delta(s,y) \leq \delta(s,u)$

but when we are about to add u to S, that means $u.distance \leq y.distance$

so $u.distance \leq y.distance \leq \delta(s,u)$

contradiction! Q.E.D.

Unweighted Graphs

BFS!

use FIFO queue instead of priority queue

relax if $v.distance = \infty$ then $v.distance \leftarrow u.distance + 1$

add v to the end of queue

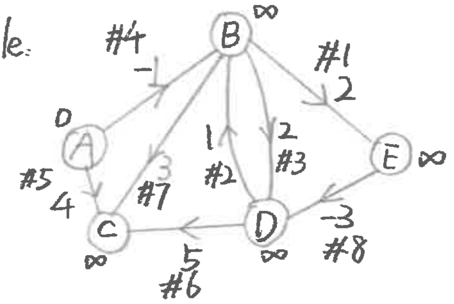
Lec 18 shortest path Ⅱ Bellman和差分约束系统

Bellman-Ford Algorithm

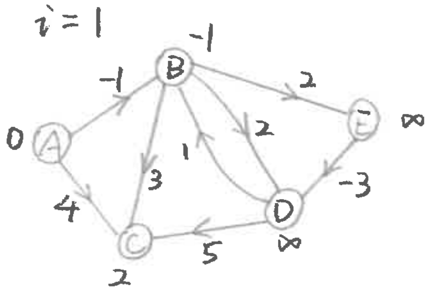
- computes shortest-path weights $\delta(s,v)$ from source vertex $s \in V$ to all vertices $v \in V$
- OR reports that a negative-weight cycle exists

```
Bellman-Ford
    s.distance ← 0
    for each v ∈ V-{s}
        do v.distance ← ∞
    for i ← 1 to |V|-1
        do for each edge (u,v) ∈ E
            do if v.distance > u.distance + w(u,v)
                then v.distance ← u.distance + w(u,v)
    for each edge (u,v) ∈ E
        do if v.distance > u.distance + w(u,v)
            then report that a negative-weight cycle exists
        else
            v.distance = δ(s,v)
```

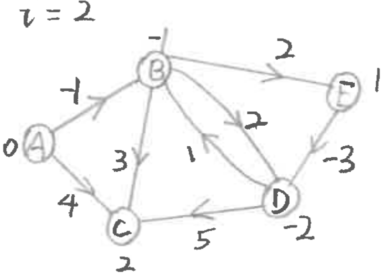
Example:



i=1



i=2



i=3

no change

i=4

no change

Correctness

if $G=(V,E)$ has no negative-weight cycle

then Bellman-Ford terminates with $v.distance = \delta(s,v)$ for all $v \in V$

proof:

consider any $v \in V$

by monotonicity of distance values, and Correctness I ($v.distance \geq \delta(s,v)$)

only need to show that $v.distance = \delta(s,v)$ at some time

let $p = \underset{\underset{s}{\parallel}}{v_0} \to v_1 \to v_2 \to \cdots \to \underset{\underset{v}{\parallel}}{v_k}$ be the shortest path from s to v

with minimum number of edges 可能有零权环

$\Longrightarrow$ p is a simple path

$\delta(s,v_i) = \delta(s,v_{i-1}) + w(v_{i-1},v_i)$ by optimal substructure

$v_0.distance = 0$ initially

$\delta(s,v_0)$ has to be 0 or there exists a negative-weight cycle

so $v_0.distance = \delta(s,v_0)$. base case check!

assume by induction that $v_j.distance = \delta(s,v_j)$ after j rounds, $j<i$

after $i-1$ rounds, $v_{i-1}.distance = \delta(s,v_{i-1})$

during the i-th round, relax (v_{i-1},v_i)

Correctness Lemma $\Longrightarrow$ $v_i.distance = \delta(s,v_i)$

after k rounds, $v_k.distance = \delta(s,v_k)$, $k \leq |V|-1$ because p is simple

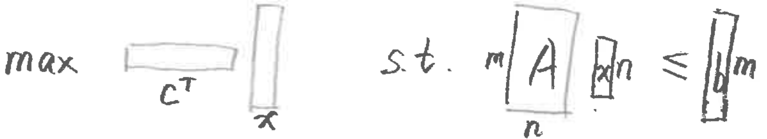
Corollary

if Bellman-Ford fails to converge after $|V|-1$ rounds,

then there has to be a negative-weight cycle

Linear Programming

given $m \times n$ matrix A, m-vector b, n-vector c,
to find n-vector x that maximizes $c^T x$ subject to $Ax \leq b$,
or no such x exists.



many efficient algorithms to solve LPs,

① simplex algorithm 单-算法
② ellipsoid algorithm 椭球算法
③ interior point algorithm 内点法
④ random sampling 随机抽样

Linear Feasibility Problem

no objective c, just find x s.t. $Ax \leq b$

Difference Constraints

linear feasibility problem where each row of matrix A has one 1 and one -1
rest is 0

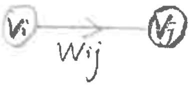
each constraint is of form $x_j - x_i \leq w_{ij}$

Example:

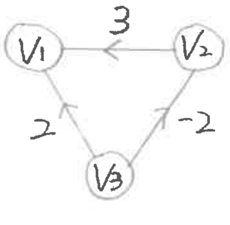
$x_1 - x_2 \leq 3$
$x_2 - x_3 \leq -2$
$x_1 - x_3 \leq 2$

Constraint Graph

$x_j - x_i \leq w_{ij}$ $\Longrightarrow$



$|V| = n$
$|E| = m$



$x_j - x_i \le W_{ij}$

$\Rightarrow x_j \le x_i + W_{ij}$

$\rightarrow j.\text{distance} \le i.\text{distance} + W_{ij}$

Theorem:

if constraint graph has a negative-weight cycle,

then difference constraints are unsatisfiable

proof:

$V_1 \rightarrow V_2 \rightarrow \cdots \rightarrow V_k \rightarrow V_1$ is a negative-weight cycle

$$\begin{aligned}
\because\ & x_2 - x_1 \le W_{12} \\
& x_3 - x_2 \le W_{23} \\
& \cdots \quad \cdots \quad \cdots \\
& x_k - x_{k-1} \le W_{k-1,k} \\
& x_1 - x_k \le W_{k,1}
\end{aligned}$$

$\therefore \quad 0 \le W_{1,2} + W_{2,3} + \cdots + W_{k-1,k} + W_{k,1} = W(\text{cycle}) < 0$

“这些约束是矛盾的！没有这样一组 $\{x_1, x_2, \cdots, x_k\}$ 可以满足所有的约束！”

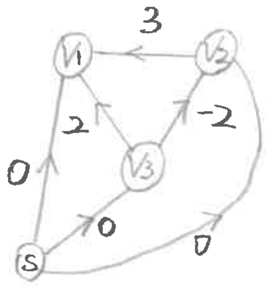
Theorem

if no negative-weight cycle in constraint graph G,

then difference constraints are satisfiable

proof:

add to G a new vertex s with a weight-0 edge from s to all $v \in V$



modified graph has no negative-weight cycles and has paths from s

$\Rightarrow$ has shortest paths from s

assign $x_i = \delta(s, v_i)$,

$x_j - x_i \le W_{ij} \iff \delta(s, v_j) - \delta(s, v_i) \le W_{ij}$

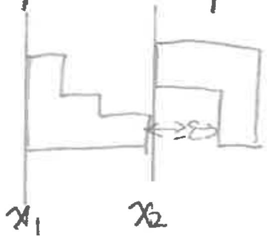
$\iff \delta(s, v_j) \le \delta(s, v_i) + W_{ij}$ Q.E.D ???

Corollary:

Bellman-Ford solves a system of m difference constraints on n variables in $O(mn)$ time

VLSI layout

place IC features without putting any two of them too close to each other



$x_2 - x_1 \geq \text{distance} + \varepsilon$

Bellman-Ford solves these constraints and minimizes the spread

"compactness"

Lec 19 shortest paths Ⅲ 点的最短路径

All-Pairs Shortest Paths

- unweighted graph: $|V| \times$ BFS $= O(|V||E|)$
- non-negative edge weights: $|V| \times$ Dijkstra $= O(|V||E| + |V|\lg|V|)$
- general:

  $|V| \times$ Bellman-Ford $= O(|V|^2 \cdot |E|)$

  three algorithms today

Problem

input: digraph $G=(V, E)$, say $V=\{1, 2, \cdots, n\}$, edge-weight function $W: E \Rightarrow R$

output: $n \times n$ matrix of shortest-path weights. $\delta(i,j)$ for $\forall i,j \in V$

① dynamic programming $O(n^4)$

matrix multiplication $O(n^3)$

② Floyd-Warshall Algorithm

③ Johnson's Algorithm